UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño
y programación de videojuegos

UNLVIRTUAL

MANIPULACION DE OBJETOS EN 2D

Unidad 3: Manipulación de buffers

Docente
Walter Sotil

Tutores
Emmanuel Rojas Fredini, Cristian Yones

CONTENIDOS

Introducción

OpenGL opera con diversos buffers: el color-buffer, el depth-buffer, el stencil-buffer y el accumulation buffer. Internamente, no son más que espacios de memoria con valores asignados a los píxeles. Los buffers en su conjunto conforman el frame-buffer.

En esta unidad estudiaremos algunos métodos de trabajo que producen resultados visuales a través del manejo de la información gráfica *per-píxel,* en lugar de la 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*.

En el campo de la computación gráfica 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[1] 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 (y encima pocos píxeles), en los llamados *bitmaps,* que eran imágenes de puntos blancos o negros.

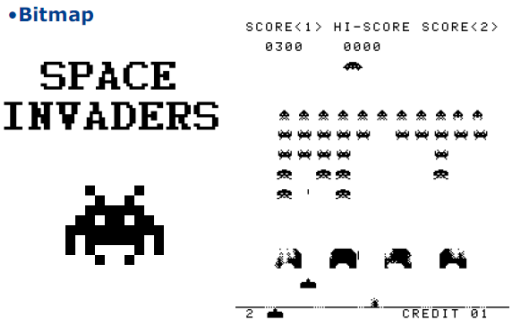


Figura 1. Bitmap.

Luego, 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. El problema residía en el elevado precio de la memoria.

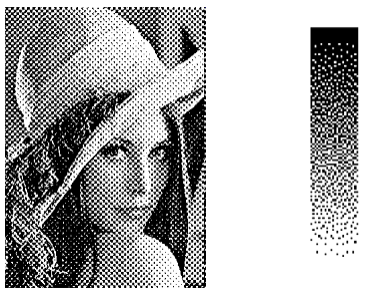


Figura 2. Dithering o puntillado con un bit por píxel.

[1] Rasterización es el proceso de representar un objeto mediante un conjunto discreto de puntos.

Del blanco y negro (¡o peor, ámbar o verde!), se pasó a 16 y de inmediato a 256 colores, organizados en una *paleta (palette).* Actualmente se utilizan 24 bits de profundidad para RGB (o 32 para RGBA).

Figura 3. Imagen representada con 256 colores.

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 casi no se usan para nada.

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

Inicialmente analizaremos el frame-buffer y luego iremos estudiando los buffers gráficos.

3.1. El frame-buffer

Frame-buffer
Es un conjunto de buffers gráficos con los que trabaja OpenGL.

OpenGL trabaja con varios buffers, entre ellos el *color-buffer,* que es 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 frame-buffer. 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 buffers: front-left, front-right, back-left, back-right y algunos otros auxiliary color buffers.
- Depth buffer.
- Stencil buffer.
- Accumulation buffer.

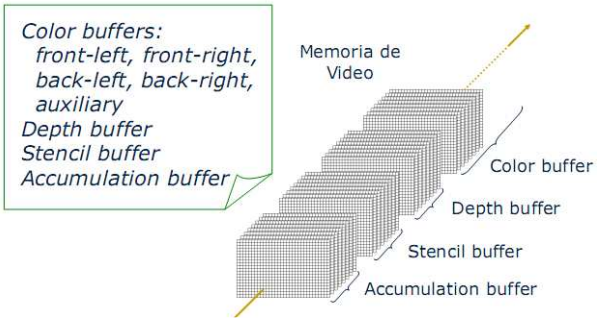


Figura 4. El frame-buffer.

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) cuántos 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 importante es saber que los datos pueden manipularse, logrando una miríada de efectos útiles y trabajando 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.

Conviene saber también, por cuestiones de eficiencia, que la disposición interna de los píxeles en la memoria es por *scan-lines* o líneas de *y* constante: a la línea *y=0* le sigue la línea *y=1* y así sucesivamente.

Además, las placas gráficas suelen utilizar *words* contiguos para almacenar distintos *buffers*. Es decir, conviene leer *r, g, b, a, z...* por píxel, en lugar de leer primero todos los r, después todos los g, etc. De cualquier manera, a los píxeles se accede de acuerdo a su posición *x, y;* la aritmética de punteros la hace siempre la placa gráfica. En general, sólo debemos pensar en las *scan-lines* para leer o escribir por filas, en lugar de hacerlo por columnas, al menos cuando se pueda elegir.

3.2. El color-buffer

Color-buffer
Es la porción de memoria que almacena los WxH píxeles de cada componente de color.

La imagen que observamos se arma en 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). Este es el buffer que se pinta o actualiza cuando un fragmento es visible.

Aquí es importante desarrollar el concepto de profundidad de color o bits por píxel. El mismo hace referencia a la cantidad de bits de información necesarios para representar el color de un píxel en una imagen digital o en el frame-buffer. Debido a la naturaleza del sistema binario de numeración, una profundidad de bits de *n* implica que cada píxel de la imagen puede tener 2^n posibles valores y por lo tanto, representar 2^n colores distintos. Dicho de otra manera, como cada bit puede ser sólo 0 o 1, la fórmula matemática para hallar el número de colores posible es la de elevar 2 a la potencia del número de bits por píxel que tengamos. De este modo, con una profundidad de 1 bit es posible representar solamente 2 colores y con una profundidad de 8 bits es posible representar 256 colores.

Primeramente, con sólo un bit por píxel, se utilizó el blanco y negro debido a restricciones dadas por el elevado costo de la memoria. Luego, se avanzó a 16 colores; posteriormente a 256, y actualmente se utilizan 24 bits de profundidad para RGB (o 32 para RGBA). Más adelante veremos el exceso de llamar *true color* a esta paleta muy densa. Con la profundidad de color de 24 bits por píxel, se dedica un octeto entero a representar la intensidad luminosa de cada uno de los tres tonos primarios de rojo, verde y azul, lo cual permite que cada píxel pueda tomar 2^{24} = 256x256x256 = 16.777.216 colores distintos.

Es decir, con 24 bits de color podemos manipular ocho 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.

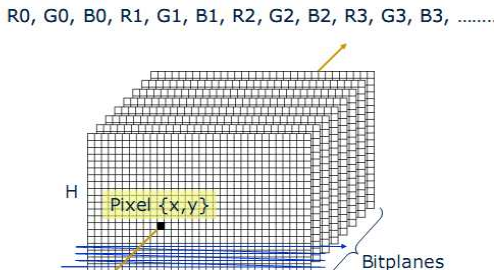


Figura 5. El color-buffer.

Existe 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),* evitando de esta forma el parpadeo *(flickering)* que se puede observar 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. Los buffers de color *(aux)* restantes se utilizan como *layers* o capas, o también 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 del objeto 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, de modo que leyendo el color del píxel se pueda conocer el objeto debajo del cursor.

3.3. El z-buffer

Al renderizar varios objetos 3D, algunos tapan visualmente a otros.

Hay varios métodos geométricos (vector) para resolver los problemas de oclusión visual, que consisten en calcular intersecciones y dividir los objetos, para luego renderizar en forma ordenada, de acuerdo a la distancia del observador.

El problema con los métodos geométricos 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* para resolver las oclusiones visuales. 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.

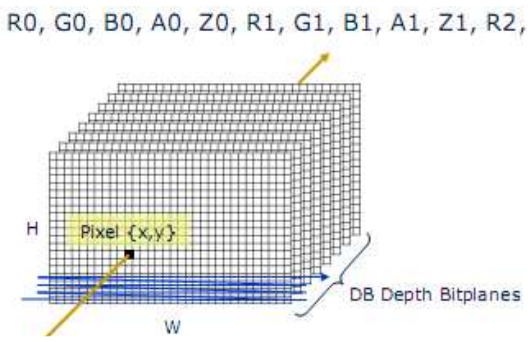


Figura 6. El z-buffer o depth-buffer.

Depth-buffer
Consiste en WxH valores de punto flotante, normalmente de 24 o 32 bits de precisión. Se encarga de resolver las conclusiones visuales.

Dijimos anteriormente que 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). Este es el buffer que se pinta o actualiza cuando un fragmento es visible.

Ahora bien, el depth-buffer se mantiene en paralelo y consiste en WXH valores de punto flotante, normalmente de 24 o 32 bits de precisión. 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 (estudiadas en la unidad 2) sirven para cuantizar el rango de profundidades, ya que no hay precisión infinita para almacenar los valores de *z*.

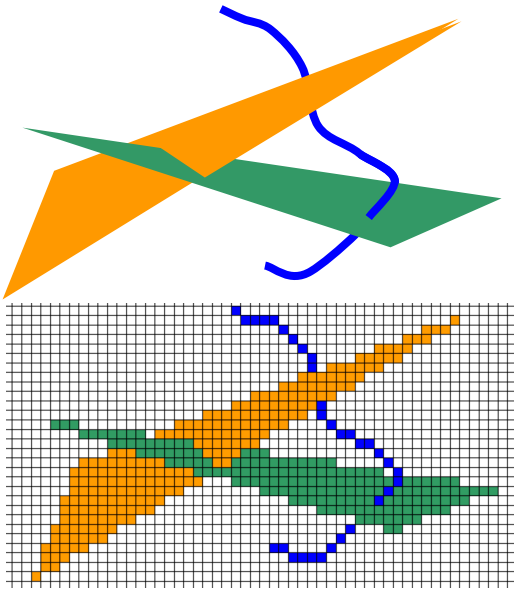


Figura 7. Algoritmo basado en el z-buffer o depth-buffer: se encarga de resolver las conclusiones visuales.

El algoritmo fue desarrollado por Edwin Catmull (uno de los fundadores de Pixar) en 1974, y es más o menos 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)

El algoritmo no almacena exactamente *z* entre znear y zfar, sino que previamente convierte ese valor en uno que se encuentra entre cero y uno, normalmente con precisión de 24 bits.

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.*

No obstante, 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 semitransparente, debe combinarse su color con el color actual del píxel. 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, el resultado depende del orden de rasterización. Una técnica utilizada

consiste en renderizar primero los objetos opacos y luego los semitransparentes, pero ordenados desde los más lejanos a los más cercanos al ojo (ó cámara).

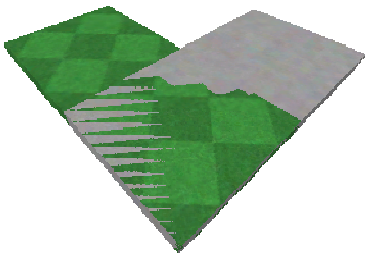


Figura 8. Visualización de z-fighting.

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)/$(2^{24}-1)$[2].
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. 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, este algoritmo 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.

4. Algunas aplicaciones de los buffers

Manipulando el z-buffer se pueden lograr algunos trucos útiles. Por ejemplo, la determinación de siluetas buscando altos gradientes de *z* entre píxeles. También es posible 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 de cierta manera las líneas invisibles en un paso y las visibles en el siguiente paso, pero 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*, que analizaremos en detalle más adelante, se utiliza para muchísimos 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 por 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 para renderizar la vista trasera.

[2] 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

También se utiliza para realizar reflejos (bastante simple) y sombras (complicado a muy complicado); hay ejemplos e instrucciones en la web, pero primero debemos aprender algo más sobre transformaciones.

Otro uso muy interesante y complicado consiste en renderizar sólidos armados mediante las operaciones booleanas de un *CSG tree*.

El buffer de acumulación, como su nombre lo indica, 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; para *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 de luz extensas, no-puntuales. El buffer de acumulación no puede manipularse píxel a píxel como el resto, sino que se le pasa (varias veces) un buffer de color entero y el resultado de las operaciones se transfiere completo al buffer de color. Este buffer suele tener más bits disponibles que el de color, por lo que las operaciones de acumulación se pueden hacer con más precisión y se *redondean* al transferir. Actualmente, el buffer de acumulación ha quedado obsoleto y no es soportado por algunas placas gráficas, siendo reemplazada su funcionalidad por la programación directa de *shaders* en la GPU.

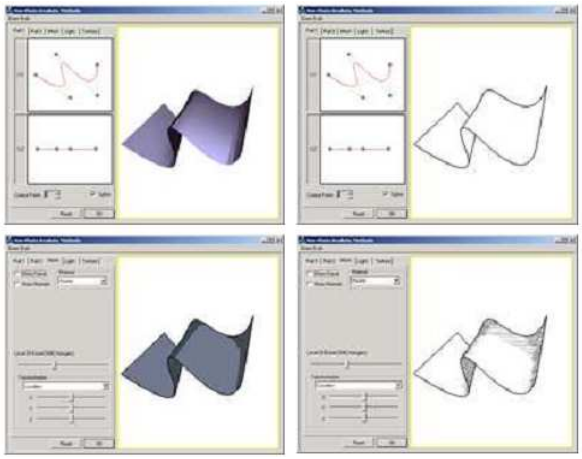


Figura 9. Aplicaciones de los buffers: determinación de siluetas.

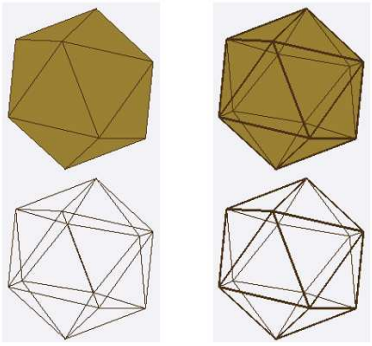


Figura 10. Aplicaciones de los buffers: discriminación de líneas ocultas.

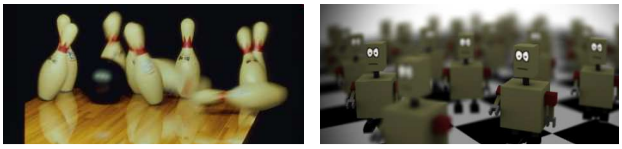


Figura 11. Aplicaciones de los buffers: reflejos, motion blur, depth of field.

5. Operaciones sobre fragmentos

Después de rasterizar, OpenGL puede realizar varias operaciones de manipulación de los fragmentos, como tests y alteración.

Después de rasterizar, OpenGL puede realizar varias operaciones de manipulación de los fragmentos antes de actualizar un píxel. Los detalles de las funciones se encuentran en el *Red Book* y las mismas pueden dividirse en dos tipos: *tests* y *alteración.* Por una cuestión de eficiencia, los tests se hacen primero. Como vimos en el caso 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 pase uno de la serie, inmediatamente se descarta.

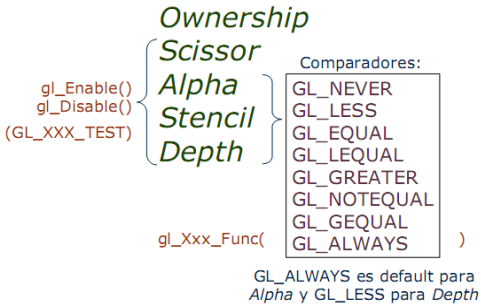


Figura 12. Test sobre fragmentos.

El píxel ownership test es un test que depende de la placa gráfica, la versión de OpenGL y el sistema operativo; se refiere a si el píxel está o no tapado por otra ventana. Si la ventana está tapada, lo razonable es que no perdamos tiempo haciendo cálculos cuyos resultados no se verán.

Ahora bien, supongamos que después de un largo proceso en batch, donde el procesador trabaja mucho desatendido, cuando el trabajo termina, el programa lee los píxeles del color buffer y graba una imagen. En tal caso, hay que leer sobre este test (o experimentar), pues puede ocurrir que no dibuje, no trabaje ni lea bien los píxeles que no son visibles por estar tapados por otro programa. (Este test en particular no se encuentra explicado en el *Red Book*; una fuente de consulta es http://www.opengl.org/resources/faq/technical/rasterization.htm).

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

El *scissor test* (tijeras) es extremadamente simple. Se define un rectángulo donde se dibuja sólo por dentro. Este test se usa para renderizar por secciones.

Todos los otros tests utilizan el mismo conjunto de comparadores entre valores: GL_ALWAYS, GL_NEVER, GL_EQUAL, GL_NOTEQUAL, GL_LESS, GL_LEQUAL, GL_GREATER y GL_GEQUAL. Los defaults son GL_ALWAYS para alpha y GL_LESS para depth; para stencil no hay default porque sólo se usa si se define. Para alpha y stencil se compara con el valor de referencia que se determina junto con el test; para depth, se compara con el valor almacenado en el depth-buffer.

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 se usa para comparar con el valor de referencia y realizar efectos de difuminado o invisibilidad de ciertas partes de la imagen.

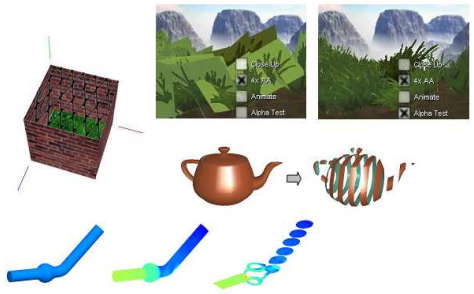


Figura 13. Test sobre fragmentos: alpha test.

Las operaciones de alteración son las que rellenan un píxel y las mismas pueden realizarse 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.

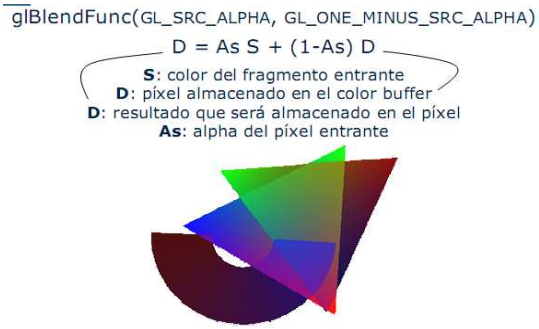


Figura 14. Blending: transparencias.

La operación de mezcla clásica, cuando entra un fragmento semitransparente, es D = As S + (1-As) D. En esta 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 resultado 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*):

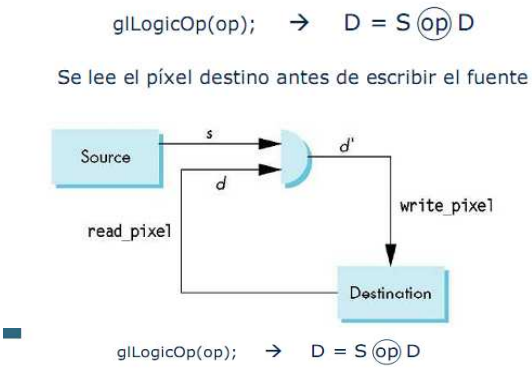
Constant	Relevant Factor	Computed Blend Factor
GL_ZERO	source or destination	(0, 0, 0, 0)
GL_ONE	source or destination	(1, 1, 1, 1)
GL_DST_COLOR	source	(R_d, G_d, B_d, A_d)
GL_SRC_COLOR	destination	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_DST_COLOR	source	(1, 1, 1, 1)-(R_d, G_d, B_d, A_d)
GL_ONE_MINUS_SRC_COLOR	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 *Red Book* o en otras fuentes bibliográficas es posible encontrar 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*[3] (exclusive or) para cursores o líneas temporarias, como por ejemplo los rectángulos de selección. La idea es que a^b^b=a, es decir, que si el píxel era de color *a* y lo pinto con color *b* aplicando *xor,* el color cambia; si luego vuelvo a pintar el resultado con color *b* (y *xor*), aparece nuevamente el color *a* original.



glLogicOp(op); → D = S op D

Parameter	Operation	Parameter	Operation
GL_CLEAR	0	GL_AND	s&d
GL_COPY	**s**	GL_OR	s|d
GL_NOOP	d	GL_NAND	~(s&d)
GL_SET	1	GL_NOR	~(s&d)
GL_COPY_INVERTED	~s	**GL_XOR**	**s^d**
GL_INVERT	~d	GL_EQUIV	~(s^d)
GL_AND_REVERSE	s&~d	GL_AND_INVERTED	~s&d
GL_OR_REVERSE	s|~d	GL_OR_INVERTED	~s|d

Figura 15. Alteración de fragmentos mediante Logic Operation.

Si se usan varios tests, es importante conocer el orden: fragment → ownership → texturing4 → fog4 → scissor → alpha → stencil → depth → blending → dithering → logic op → masking → píxel.

Una operación de alteración muy costosa es *glClear(),* que borra todo el contenido del o de los buffers solicitados, porque recorre 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.

OpenGL posee un pipeline 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.

Existen máscaras de escritura: *gl<Buffer>Mask()* para color, depth y stencil. Para color y depth, las máscaras son *booleanas* (un solo bit) e indican si el buffer se puede escribir o no.

[3] La operación lógica *xor* es aquella que da por resultado uno cuando los valores en las entradas son distintos. Por ejemplo: 1 y 0, 0 y 1 dan por resultado un uno.

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

Para el stencil, la máscara tiene el mismo tamaño en bits e indica cuáles son los bits que pueden ser alterados.

Hay que tener especial cuidado al borrar el contenido del buffer *(glClear)* , debido a que implica escribir el *clear-value* en todos los píxeles; y también se aplican las máscaras 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.

6. El stencil-buffer

El *stencil-buffer* consta de 8 bits y sirve para colocar banderas (flags o números enteros que se interpretan bit a bit) en distintas regiones, logrando así diferentes efectos según 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, de acuerdo al 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*, se descarta, esto es, no sigue hacia el color-buffer; pero aún así, el stencil-buffer se actualiza, sólo que de distinto modo, en función de que tests pasó o no el fragmento.

Hay una función *(glStencilFunc)* que define cómo se hará el filtrado; determina un valor de referencia *(Stencil Ref),* una máscara *(Stencil 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 (*fail*), el fragmento se descarta; si pasa, se somete al test de profundidad.

Como explicamos anteriormente, en el *depth-test*, el valor *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; caso contrario, sigue su camino al *color-buffer*, mientras que se actualiza el *depth-buffer*, aunque con previo paso por la máscara de escritura *Depth Mask*, que indica si en el buffer se puede escribir o no.

El *stencil-buffer* se actualiza de distinto modo según los tests que fallen o pasen. Para ello se provee la función *glStencilOp(),* la cual indica qué hacer en cada uno de los tres casos: falló el *stencil test,* pasó el *stencil test*, pero falló 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 ocurrir que se borre, se incremente, se invierta, se escriba el valor de referencia, etc. Pero, para darle más versatilidad, primero se enmascara lo que queremos escribir con una máscara de escritura *(Stencil Write Mask),* la cual se define aparte mediante la función *glStencilMask().* Esto significa que hace lo que debe hacer, pero sólo donde la máscara de escritura tenga un bit en uno.

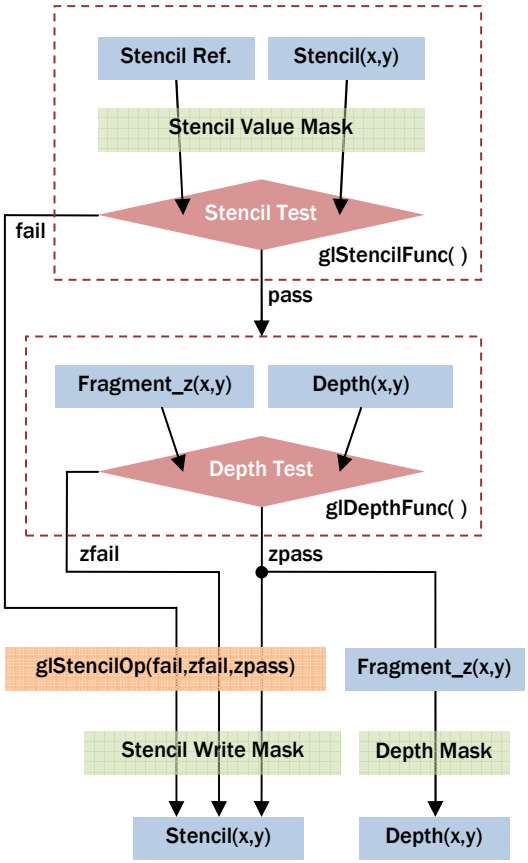


Figura 16: Escritura en el Stencil-buffer.

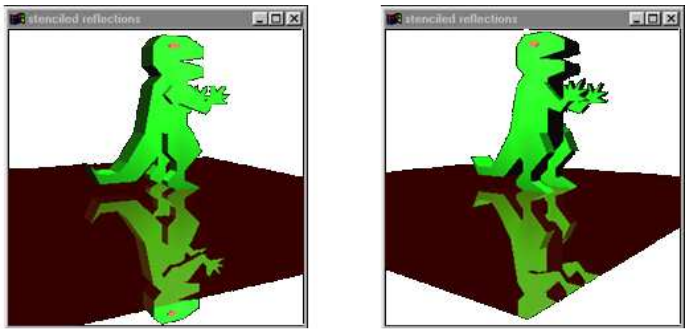


Figura 17: Ejemplo de utilización de Stencil-buffer para "recortar" el reflejo sobre el suelo.

BIBLIOGRAFÍA

Hearn, D.; Baker, P. *Computer Graphics, C version.* Second Edition. Pearson Prentice Hall, 1997.

Kilgard, M. J. "The OpenGL Utiliy Toolkit (GLUT) Programming Interface". Silicon Graphics, Inc, 1996 [en línea] [OpenGL]
http://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf

Neider J.;Davis,T.; Woo M. "OpenGL Programming Guide: The Official Guide to Learning OpenGL". Addison-Wesley Publishing Company, 1997 [en línea] [OpenGL]
http://www.opengl.org/documentation/red_book/

OpenGL [en línea] http://www.opengl.org

The Blue Book: The OpenGL Reference manual [en línea] [OpenGL]
http://www.opengl.org/documentation/blue_book/