UNIDAD 1: INTRODUCCIÓN

¿Qué significa renderizar una escena? ¿Qué cosas incluye la descripción/el modelado de una escena para ser procesada por la GPU? Detalle cómo se define o representa cada una de esas cosas.

El **renderizado** es una (gigantesca) serie de procesos en la GPU que transforma una escena tridimensional modelada en la CPU en una imagen; de modo tal que esa imagen, puesta delante del ojo, muestre la misma escena (realista o no) que obtendríamos con una cámara. Tiene bastantes requerimientos tanto en software como en hardware para lograr las exigencias de calidad y eficiencia en la actualidad.

El rendering puede ser fotorrealista (simula un modelo físico) o no fotorrealista (más abstracto). También se clasifica por realizarse en tiempo real (resultado instantáneo, como los videojuegos, idealmente 60 fps como mínimo) o tiempo por lotes (películas, por ejemplo, que toman mucho tiempo para renderizar).

La descripción de una escena que debe darle la CPU a la GPU para que pueda ser procesada implica la posición de la cámara y de las luces, y el conjunto de objetos a mostrar. Cada objeto tiene su forma (descripción geométrica: sus primitivas o conjunto de vértices representadas mediante coordenadas), color (representadas mediante ternas R, rojo, G, verde, B, azul) y/o texturas (imágenes que posteriormente serán mapeadas donde corresponda).

¿Qué diferencias hay entre vector y ráster?

El **mundo vectorial** está representado por funciones matemáticas analíticas. Si uno hace zoom a una curva definida por funciones matemáticas, los contornos se verán suaves y continuos. Es mejor para trabajar como entrada, pero después se le hace su transición a ráster para mostrarse por pantalla.

El **mundo ráster** está representado por píxeles, esto es, una matriz cuyos elementos son colores (tres luces RGB tal que todos los colores representables por la pantalla se harán como una combinación de éstas). Al hacer zoom a una imagen ráster se ve pixelada.

¿Qué etapas componen el pipeline gráfico? Describir su funcionamiento.

La etapa de **aplicación** es la única correspondiente a la CPU en el pipeline gráfico descrito. En esta etapa, la CPU arma la descripción de la escena, genera los triángulos (las ternas de vértices con alguna información adicional como el color o el material) y se los manda a la GPU para su procesamiento.

Pasando al trabajo de la GPU, en la etapa de **procesamiento de vértices** se realizan las transformaciones necesarias para acomodar el triángulo a las coordenadas de la ventana. Le pasa las primitivas transformadas a la próxima etapa.

En la etapa de **rasterización** se pasa del mundo vectorial al mundo ráster. A partir de los tres vértices ubicados en la pantalla, determino qué píxeles cubriría el triángulo y, a cada uno de ellos, interpolarle los valores asociados (la mezcla de colores, la profundidad antes de ser aplastado con proyección). Le pasa los fragmentos a la próxima etapa.

En la etapa de **procesamiento de fragmentos**, generalmente la más cara del pipeline ya que implica el cálculo de iluminación y aplicación de texturas, se define si un fragmento sobrevive o no. Esto es, si pasará a ser un píxel de la pantalla o no, mediante distintos tests.

¿Qué tipos de modelado distinto hay?

- **CSG-Tree:** Consiste en utilizar un conjunto reducido de cuerpos simples y analíticos y realizar operaciones booleanas entre ellos (uniones, intersecciones, diferencias). Con esto, desde abajo hacia arriba, se construye un árbol binario que representa un objeto complejo.
- **B-REP:** Método más común. Se utiliza una malla de polígonos sencillos, en general triángulos, que representa la superficie aproximada.
- **LEVEL-SETS:** Permite definir superficies de nivel, útil para situaciones en las que se requiere almacenar contenido tridimensional. En un dominio espacial, se cuenta con un campo escalar: en cada punto del espacio se asigna un valor de alguna variable (temperatura, altura).

UNIDAD 2: INTERPOLACIÓN

¿Qué es la interpolación? ¿Qué tipos de interpolaciones hay? ¿Qué usos, ventajas, desventajas, límites tiene cada una?

La **interpolación** es un proceso por el cual se define un valor aproximado para un punto cualquiera a partir de los valores conocidos en algunos puntos dados. En general, tenemos un conjunto finito de datos {x,v} tal que se conoce la x y queremos estimar el valor v. Existen distintos tipos de interpolaciones según qué puntos voy a tomar para la aproximación:

Empezando por categorizaciones simples: una **interpolación convexa** es aquella en la que los pesos no son negativos. Una **interpolación lineal** es aquella cuyos pesos son funciones lineales de las coordenadas, aunque en general se le llama interpolación lineal a la que es afín y lineal.

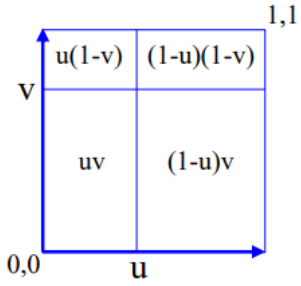
Una **interpolación afín** es un promedio ponderado o combinación lineal con coeficientes que suman uno. Dichos coeficientes coinciden con los de la combinación afín, pero tratándose como "pesos", tal que se termina calculando las variables del mismo modo que las coordenadas de una combinación afín. $x = \sum a^i x_i\,,\ \sum a^i = 1 \rightarrow\ v = \sum a^i v_i.$

El caso más simple es la interpolación afín de dos puntos: $x = (1-a)\cdot x_0 + a \cdot x_1 \rightarrow v = (1-a)\cdot v_0 + a \cdot v_1$. Puedo extrapolar con esta fórmula (parámetros fuera de [0,1]), pero la combinación deja de ser convexa y no puedo salirme de la recta que definen los puntos.

Una interpolación afín se utiliza, generalmente, para la interpolación de coordenadas de textura y de color, para obtener valores como el clima, presión, temperatura en distintos nodos.

Una **interpolación bilineal** es una forma particular de expresar los pesos de la combinación convexa de cuatro puntos que forman un cuadrángulo (en el plano o en el espacio) pero usando sólo dos parámetros: coordenadas u y v de un cuadrado unitario en el espacio de parámetros. Los cuatro parámetros de la combinación afín resultante suman uno, se obtiene una interpolación afín convexa pero no-lineal sino bilineal.

$$\mathbf{x}_{01} = (1-u)\,\mathbf{x}_0 + u\,\mathbf{x}_1$$
$$\mathbf{x}_{32} = (1-u)\,\mathbf{x}_3 + u\,\mathbf{x}_2$$
$$\mathbf{x} = (1-v)\,\mathbf{x}_{01} + v\,\mathbf{x}_{32} = \overbrace{(1-v)\,(1-u)}^{\alpha_0}\,\mathbf{x}_0 + \overbrace{(1-v)\,u}^{\alpha_1}\,\mathbf{x}_1 + \overbrace{v\,u}^{\alpha_2}\,\mathbf{x}_2 + \overbrace{v\,(1-u)}^{\alpha_3}\,\mathbf{x}_3$$



Entre sus usos, se encuentra el mapeo de texturas o la interpolación de colores en cuadriláteros, ya que el resultado es más suave que el obtenido dividiendo en triángulos.

Una **interpolación hiperbólica** es una interpolación en el espacio proyectivo, una interpolación lineal en coordenadas homogéneas con una dimensión más. Aquí, se trata la w como un peso adicional en el cálculo del promedio ponderado. En general, se utiliza al rasterizar, para asignar colores o pegar las texturas correctamente.

Hagamos un par de ejemplos. Tenemos dos puntos en 3D: $\mathbf{x}_0 = \{0,0,0\}$ y $\mathbf{x}_1 = \{1,0,0\}$; los representamos en coordenadas homogéneas con el mismo w=1: $\mathbf{x}_0 = \{0,0,0,1\}$ y $\mathbf{x}_1 = \{1,0,0,1\}$. Con pesos $\alpha^0 = \alpha^1 = 1/2$ el punto interpolado es: $\mathbf{x} = \{0,0,0,1\}/2 + \{1,0,0,1\}/2 = \{1/2,0,0,1\}$ que equivale en 3D a $\{1/2,0,0\}$; en este caso, ambos w valen uno y el resultado no es sorprendente. Ahora, si aumenta el w del segundo punto: $\mathbf{x}_1 = \{2,0,0,2\}$ (el mismo punto 3D) el nuevo resultado es $\mathbf{x} = \{0,0,0,1\}/2 + \{2,0,0,2\}/2 = \{1,0,0,3/2\}$ que al pasar a 3D queda $\{2/3,0,0\}$, más cerca del segundo punto, que "pesa" el doble que el primero.

¿Qué son las coordenadas baricéntricas? ¿Cómo se calculan?

La **inversión de la interpolación** utiliza una variable que se interpola linealmente (longitud, área o volumen) para encontrar los pesos que logran este resultado. Para esto, se utilizan coordenadas baricéntricas: pesos lineales de un conjunto de nodos con independencia afín.

Si tengo dos puntos y quiero obtener un punto sobre el segmento que éstos definen, el procedimiento para calcularlas implica plantear una interpolación afín de las longitudes de los extremos al punto a interpolar. Con esto, se despeja el alfa de la ecuación y se obtienen ambos pesos (se obtiene un peso y el otro es su complemento).

En el caso de tener extrapolaciones, se debe tener longitudes con signo y para eso se establece un orden de los puntos. Luego, $a^1 = (\boldsymbol{x} - \boldsymbol{x_0}) \cdot \boldsymbol{l}/l^2$ y $a^0 = (\boldsymbol{x_1} - \boldsymbol{x}) \cdot \boldsymbol{l}/l^2$.

Para tres puntos el procedimiento es similar, pero calculando las áreas, también respetando un orden para que, cuando el punto está en el interior del triángulo, apunten al mismo lado del plano que el vector área total.

Si aumento el número de puntos interpolantes, necesito volúmenes de tetraedros y se hacen productos triples (obtengo paralelepípedos).

¿Qué son las isolíneas o líneas isoparamétricas? ¿Para qué sirven?

Las **isolíneas o líneas isoparamétricas** son las líneas que, paralelas a un nodo, tienen una coordenada de área respectiva constante. Para un tetraedro, las isosuperficies son los planos paralelos a la cara opuesta de un nodo.

Entre sus aplicaciones, se encuentra una manera gráfica de observar los pesos de cada nodo interpolante en distintas áreas. Esto es especialmente útil cuando tenemos transformaciones afines y queremos ver si una transformación es correcta verificando que se mantenga la combinación afín con las coordenadas baricéntricas calculadas en la primitiva original.

¿Qué es el convex-hull de un conjunto de puntos fijos? ¿Para qué nos sirve? ¿Cómo reduzco el ancho del convex-hull?

El **convex-hull** es un envoltorio convexo del conjunto de puntos, definido por el menor convexo que lo contiene. Se podría decir, es como tomar un elástico y envolver a todos los puntos. Determina el límite de las posiciones que pueden obtenerse haciendo combinaciones afines de todos los puntos fijos, con pesos no negativos ($\alpha >= 0$).

Una buena aplicación del convex-hull es para encontrar intersecciones. Teniendo, por ejemplo, una curva de Bezier de grado 3 y orden 4 (esto es, con 4 puntos de control), si achico sucesivamente su convex-hull hasta una cierta tolerancia dependiente del tamaño original, puedo usar ese convex-hull para calcular intersecciones con la curva. Otro ejemplo es que si tengo dos curvas C1 y C2, achicando su convex-hull puedo ir viendo que se intersequen hasta encontrar el punto de intersección.

Con el algoritmo de subdivisión, puedo reducir el ancho del convex-hull cuadráticamente con cada subdivisión. Esto me sirve para encontrar la intersección de la curva con una línea u otra curva, para rasterizar la curva o, también, para encontrar el punto de la curva más cercano a un punto dado. Dado que la curva está dentro del convex-hull, el ancho acota el error y se utiliza para decidir que un tramo es suficientemente recto, detener el algoritmo y trazar el tramo como un segmento rectilíneo, ya sea para rasterizar o interceptar.

UNIDAD 3: TRANSFORMACIONES

¿Qué es una transformación? ¿Qué tipos de transformaciones hay? ¿Qué matrices los representan?

Una **transformación** es una función que hace corresponder a cada punto O del espacio con otro punto Ô del mismo espacio: Ô = T(O). En computación gráfica, se las aplicamos a los objetos para moverlos o deformarlos, y a las direcciones para alterarlas.

Las **transformaciones lineales** de los puntos son aquellas que preservan los coeficientes de toda combinación lineal: L es lineal sí y sólo si L(ax + by) = aL(x) + b(Ly). En una transformación lineal basta conocer cómo se transforma la base para poder calcular cómo se transforman todos los demás vectores. Una matriz de una transformación lineal tiene la forma: $\begin{bmatrix} x & x & 0 \\ x & x & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

Las **transformaciones afines** de los puntos son aquellas que preservan los coeficientes de las combinaciones afines: P = suma(a_i*x_i), suma(a_i) = 1. Son transformaciones lineales del vector posición y una reubicación del origen o traslación. Una matriz de una transformación afín tiene la forma: $\begin{bmatrix} x & x & x \\ x & x & x \\ 0 & 0 & 1 \end{bmatrix}$ donde las dos primeras columnas (R^2) son los vectores base transformados menos el origen transformado y la tercera columna es la traslación del origen. En una transformación afín, las rectas paralelas se mantienen paralelas.

Las **transformaciones proyectivas** transforman todos los puntos del espacio tridimensional en puntos de un mismo plano, el plano de la imagen. En el proceso se pierde una transformación, por eso son transformaciones de rango 2, no invertibles y no se puede recuperar la posición 3D de un punto de la imagen 2D. Una matriz de una transformación proyectiva tiene la forma: $\begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \end{bmatrix}$
Esto es, cualquiera de sus elementos es variable a diferencia de las demás transformaciones mencionadas.

¿Qué es el espacio proyectivo? ¿Cuáles son las coordenadas homogéneas?

El **espacio proyectivo** P^2 tiene como puntos (elementos) a las rectas por el origen en R^3, cuya dirección está definida por cualquier vector de R^3 (excepto el nulo). {x,y,w} y {ax,ay,aw} representan la misma recta por el origen y, para identificarla mediante un único vector, la cortamos por el plano w = 1 (divido por w). Las rectas horizontales que están en el plano w = 0 no cortan al plano w = 1 en ningún punto, por lo que se suelen llamar “puntos ideales” o “puntos en el infinito” de P^2. El plano ideal es aquél con w = 0 y el plano proyectivo es el de w = 1.

El vector {x,y,w} ≠ 0 de R^3 define las **coordenadas homogéneas** de un punto P = {x,y} en R^2. Si w ≠ 0 son las del punto finito {x,y} = {X/w, Y/w}. Si w = 0, son las de un punto en el infinito, en dirección a {X,Y}. Podemos decir que con w ≠ 0 tenemos puntos (finitos) y con w = 0 direcciones hacia puntos en el infinito.

¿Qué diferencias hay entre proyección perspectiva y proyección ortogonal? ¿Puede decirse que una contiene a la otra como caso particular? ¿Por qué? ¿En qué casos utilizaría cada una?

La **proyección perspectiva** consiste en interceptar, en el plano de la imagen, todos los rayos que van desde cada punto de la escena al ojo del observador. Queda definida por medio de la posición del ojo, punto de vista o centro y un vector desde el ojo, en dirección a la visual y perpendicular al plano de la imagen, su longitud se denomina distancia focal y define la ubicación del plano.

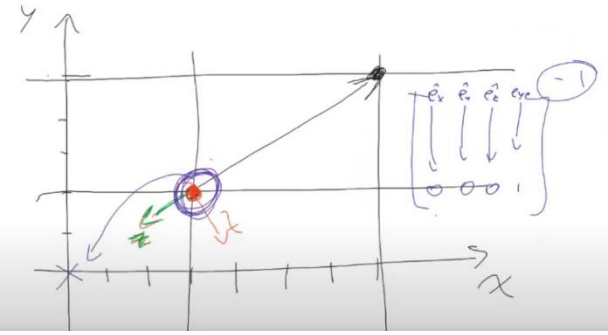
Proyectar en x = 1 implica que las coordenadas finales de un punto cualquiera {x,y,z} sean {1,y/x,z/x}. No es lineal en R^3 pero sí en homogéneas: {x,y,z,1} -> {x,y,z,x} = {1,y/x,z/x}.

La **proyección ortogonal** aplasta al espacio como un acordeón. Se puede decir que es un caso particular de la perspectiva porque es como si lleváramos al infinito el ojo de la proyección perspectiva: es como hacer una proyección perspectiva en x = 0, por eso es al infinito porque vamos a estar dividiendo por 0 tecnicamente.

Proyectar en x = 0 consiste en asignar a cualquier {x,y,z} el punto {0,y,z}. Es una transformación lineal aún en R^3 pero expresado en homogéneas es {0,y,z,1}.

Dado este look at: glLookAt({3,2,0}, {7,5,0}, {0,0,1}), ¿cuál es la matriz que produce? ¿De qué espacio a qué otro espacio transforma? ¿En cuál de las matrices de OpenGL se carga?

LookAt me lleva del espacio del mundo al espacio del ojo o de la cámara. Se carga en la transformación de model view (la que simultáneamente ubicaba los objetos en el modelo y el modelo respecto a la cámara: transforma de las world coordinates a las del ojo o de la cámara). La cámara siempre está en el (0,0) mirando al z negativo.



Considere las matrices model, view y projection que se utilizan habitualmente, ¿de qué espacio a qué otro espacio transforma cada una de ellas?

El espacio del modelo incluye vértices como modelos, geometrías, mallas. El espacio de la cámara es aquél que está en {0,0,0} y mira al -z. El espacio del mundo implica objetos renderizables, luces, cámaras. Las posiciones de los vértices se transforman aplicando, en orden, una model matrix, una view matrix y una projection matrix.

La matriz **model** transforma vértices del espacio del modelo al espacio del mundo. Se le pueden aplicar transformaciones y transforma mi objeto. La matriz **view** transforma del espacio del mundo al espacio del ojo o de la cámara. La matriz **projection** se aplica a los vértices en el espacio de la cámara para transformarlos al espacio de la imagen (con coordenadas homogéneas).

UNIDAD 4: RASTERIZACIÓN

¿Cuáles son los requisitos para un buen algoritmo de rasterización?

En los algoritmos de rasterización se dice “pintar un píxel” al correcto “generar un fragmento en la posición correspondiente a un píxel y asignarle las variables interpoladas de los vértices (color, textura, etcétera)”.

- **Forma adecuada de la curva:** mi representación raster de la curva o recta en su forma vectorial debe asimilar a la curva.
- **Contiguidad:** El máximo entre la diferencia de columnas y de filas debe ser 1: se exigirá que las curvas de tendencia horizontal tengan un pixel de alto y las de tendencia vertical uno de ancho. Esto es, no deben haber cuadraditos sin pintar.
- **Confiabilidad y eficiencia:** El algoritmo se utilizará reiteradas veces, por lo que debe ser confiable y no debe tardar mucho en procesarse.

¿Qué métodos de rasterización de segmentos hay?

- **DDA para segmentos:** Como las rectas tienen pendiente constante, empieza el algoritmo definiendo la diferencia en x y la diferencia en y. Si el valor absoluto de la diferencia en x es mayor, entonces la recta tiene una tendencia horizontal. En ese caso, la curva debe avanzar de a 1 en su coordenada x y en relación a la pendiente en su coordenada y. Lo mismo sucede al revés, si la diferencia en y es mayor (tendencia vertical) se avanza de a 1 en y, y la otra coordenada avanza según la pendiente.
  Entre sus ventajas: va sumando una pendiente fija. Tiene una sola división (por dx o dy) al inicializar el lazo, evitando fácilmente la división por cero y no hay ninguna división ni multiplicación en el interior del lazo. El redondeo de la variable que se incrementa lo hace solo al principio.
- **Bresenham:** Es el algoritmo DDA con una mejora que consiste en usar aritmética de enteros. Pinta al menos un punto y no pinta el último como desventaja.

¿Qué métodos de rasterización para curvas hay?

- **Subdivisión:** Dada una curva, la subdivide en segmentos rectos hasta que la distancia del punto de subdivisión a la curva real sea menor o igual a medio píxel, para luego poder aplicarle un método de rasterización para rectas a cada segmento resultante (Bresenham). Para el correcto funcionamiento de ésta, es necesario realizar n*2 (n: grado de la curva) subdivisiones iniciales: lo malo es que no siempre vamos a saber el grado de la curva, pero se pueden hacer 100 divisiones iniciales rápidamente. Esto es para evitar posibles cortes de la subdivisión en curvas como y = x^3, donde un punto en el centro cortaría la recursividad, pero la segmentación resultante no es adecuada a la curva.

- **DDA para curvas:** Consiste en hacer saltos finitos del parámetro t (que va de t0 a t1) que generen fragmentos continuos. Podemos suponer que la curva es prácticamente recta en un píxel, entonces podemos aproximar incrementos mediante diferenciales (aproximación de Taylor de primer orden). Desde un t = t0, si la curva tiene tendencia horizontal (diferencia en x mayor que diferencia en y), sumo t += 1/|dy(t)|, sino sumo t += 1/|dy(t)| hasta que t >= t1.
  Existen dos problemas con este método: no es sencillo garantizar que no se divida por cero. No siempre se pude saber si la curva es lo suficientemente suave por lo tanto también hay que garantizar que no haya dejado píxeles discontiguos o pintado dos veces. Se emplea cuando se sabe la ecuación y las derivadas en forma paramétrica de la curva.

¿Qué método utilizamos para rasterizar figuras planas cerradas o para rellenar?

El método utilizado es el de **scanlines**: se tiran rayos desde un punto hasta otro punto que estén fuera de la caja que encierra la figura (bounding box), luego el rayo va a estar dentro de la figura cuando haya cortado (intersección del rayo con la curva) un número impar de veces a la curva y ahí es donde debe pintar. El buffer de memoria de la imagen se almacena como una sucesión de línea horizontales, por lo tanto, las scanlines deben ser horizontales para optimizar el uso de la memoria cache.

El problema está en las tangencias y las auto intersecciones, lo cual afecta a la efectividad del método. Por ejemplo, si estoy justo en un bordecito y empiezo a pintar en un píxel de borde, pero después ya no tengo más píxeles, sigue pintando, aunque esté fuera de la curva y nunca para. Cuando el rayo atraviesa una frontera, tengo que asegurarme de que no cambie la paridad por cada píxel de la frontera. Hay soluciones para todos estos problemas.

¿Cómo interpolar las variables de los fragmentos?

Cuando rasterizo un triángulo no sólo determino qué píxeles cubre para generarle fragmentos, que eventualmente podrán llegar a ser píxeles, sino que en cada fragmento hay que interpolarle sus propiedades: su color, coordenadas de textura, normales, z para saber quién está adelante y quién está atrás para cuando dos triángulos caigan en un mismo píxel.

Puedo interpolar linealmente la z porque es la única variable que varía linealmente, pero para el resto de las variables (color, textura), a partir de ese z, las interpolo con hiperbólica (el triángulo que rasterícé en 2D es la versión proyectada de un triángulo en 3D, si algún vértice estaba más lejos de la cámara, las cosas varían distinto en el mismo fragmento y deja de ser lineal).

UNIDAD 5: COLOR

¿Qué es un color? ¿Qué terna de color se utiliza más en computación gráfica? ¿Por qué esos tres colores?

Si bien el **color** se puede pensar como una distribución espectral de longitudes de onda electromagnéticas visibles (una función continua I(λ), no un valor), también es cierto que, dado el contexto (demostrado mediante las famosas ilusiones ópticas), podemos percibir un mismo color físico como si fuera otro color completamente distinto. Luego, podría decirse también que es lo que se percibe en el cerebro comparando con patrones aprendidos y en un contexto dado (el color percibido por cada uno).

La terna de colores que más se utiliza en computación gráfica es la de rojo, verde y azul (**RGB**). Esta terna representa los colores aditivos, llamados así ya que las pantallas (monitores, televisores) tienen una matriz de ternas de luces con estos tres colores. El monitor apagado es negro y, si queremos tener un píxel de cierto color, debemos emitir luz (prender en cierto grado una combinación de las tres luces) para que se pueda ver en la pantalla como queremos.

Una posible respuesta a por qué se utilizan RGB es que el ser humano tiene unas células especializadas en la retina: los conos y los bastones. Los bastones están encargados de la intensidad de la luz, de ver formas en la oscuridad, pero no colores. Los conos son aquellos que nos permiten ver colores y hay tres tipos de conos que filtran las longitudes de onda de distinto modo: que no dejan pasar toda la luz en la misma medida. Un cono es más sensible a la luz en la zona de longitudes de onda cortas (azul), otro en la zona media (verde) y otro en las largas (rojo). Con esos tres tipos de conos, desde cada pequeña zona del ojo se envía al cerebro una terna de valores que definen el color percibido.

¿Qué es el gamut? ¿Qué es la corrección gamma?

El **gamut** es la zona limitada que determina el conjunto de colores que puede captar una cámara o reproducir un monitor, por ejemplo. Cualquier representación de esta que veamos en un monitor o impreso en una hoja es falsa, porque no puede reproducir todos los colores identificables por el ojo, la figura es meramente indicativa.

Una determinada terna de colores (por ejemplo, las tres luces RGB) puede formar cualquier color que se encuentre en el interior del triángulo que definen esos tres colores, como puntos, en el diagrama. Pero hay toda una gama de colores que me queda afuera y que no se puede representar, por lo que es falso decir que usamos RGB porque puede representar todos los colores.

La **corrección gamma** existe ya que nuestra percepción no es lineal: aplicar la misma cantidad de R, G y B no va a dar blanco (no está en el centro del triángulo sRGB en el gamut). Para convertir una imagen en colores a escala de grises, primero se deshace la corrección gamma y luego se calcula la luminancia: el verde influye mucho más que el rojo en la percepción de luminosidad y el azul no aporta casi nada.

Las impresoras usan CMYK, ¿por qué sólo 4 colores? ¿Por qué justo esos 4 colores? Si se parte de una imagen RGB, ¿cómo se obtienen los valores de C, M e Y?

Las impresoras usan, en un principio, las tintas **cian (C), magenta (M) y amarillo (Y)**, terna que representa los colores sustractivos, porque se parte de una hoja blanca y se debe restar color para absorber la luz. Estos colores son los complementos de los colores aditivos: C = 1 – R, M = 1 – G, Y = 1 – B. Luego, si quiero representar el color rojo, debo utilizar sólo magenta y amarillo porque el cian representa la ausencia del rojo.

En cuanto al **negro (K)**, si bien se podría representar como la suma de las otras tres tintas: C + M + Y = K (si sumo todas las tintas a la hoja blanca, absorben todo el color), se implementó la tinta negra ya que es mucho más barata y se usa solo una tinta para representar el mismo efecto. Hasta se implementa para realizar efectos de profundidad con la menor cantidad de tinta de color posible para abaratar costos.

¿Cuál es el sistema HSL? ¿En qué casos se usa?

El **sistema HSL** está dado por un matiz (hue), saturación y luminosidad. El matiz indica el color de base o puro del espectro, la saturación es cero para el gris y uno para el color puro. La luminancia es cero para el negro y uno para el blanco. Este sistema es más intuitivo para los usuarios porque se especifica directamente con el matiz el color que queremos y no hay que pensar en las combinaciones de los colores RGB.

¿Qué es el sombreado o shading? ¿Qué tipos de sombreado hay?

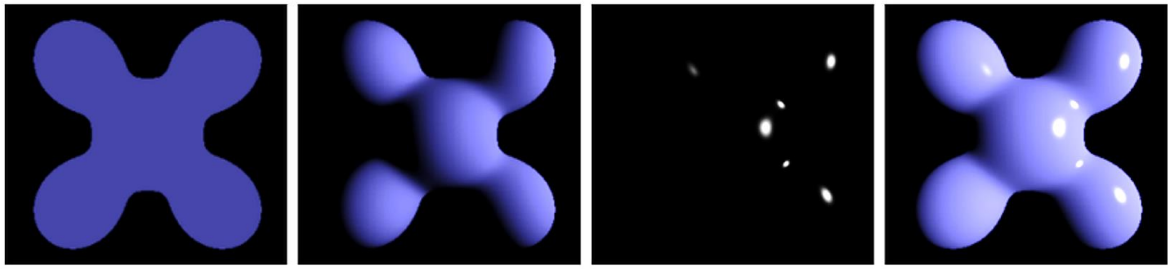
Se entiende por **sombreado** (shading) al degradé de color que vemos en un objeto por el hecho de no recibir luz en forma uniforme. El sombreado de objetos opacos se modela mediante una función BDRF (bidirectional reflectance distribution function) que indica, para un punto de la superficie del objeto, la proporción de luz reflejada, en función de la longitud de onda, de la dirección de incidencia y de la dirección hacia el ojo u observador.

El sombreado es la primera etapa programable de la GPU y no es lo mismo que proyección de sombras, que es cuando tengo otro objeto y me proyecta una sombra sobre mi objeto.

Respecto a la luz incidente, existen los modelos locales que solamente consideran la luz proveniente de un número reducido de fuentes puntuales. Los modelos globales, por otro lado, calculan la luz proveniente de fuentes extensas o del reflejo en otros objetos del entorno y admiten, además, objetos traslúcidos que refractan luz: producen efectos realistas, pero implican muchos cálculos y por esto no se pueden usar en real-time, sino que se hace off-line (se prepara antes de renderizar).

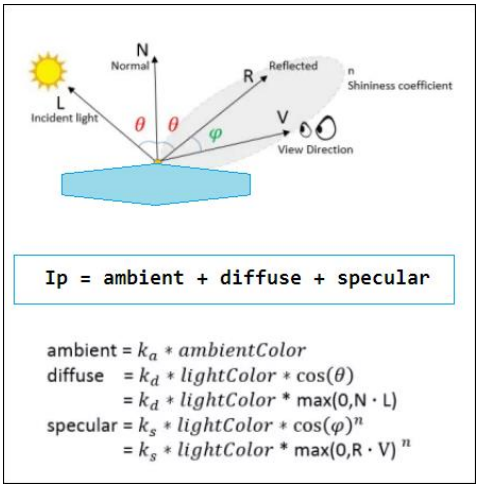
¿Qué es el modelo de Phong? ¿Qué tipo de modelo de iluminación es? Explique sus componentes. Mencione dos efectos que se observen en la realidad pero que este modelo no represente.

El **modelo de Phong** es un modelo BDRF local y lineal, que dice cómo rebota la luz a partir de dos direcciones (donde está la luz y dónde apunta la cámara). Suma tres mecanismos de reflexión de la luz de las fuentes: ambiente, difusa ideal y especular no ideal. OpenGL le agrega una componente más, la emisiva.



$$I = K_a I_{ag} + \Sigma_j [K_a I_{aj} + K_d I_{dj} n \cdot l_j + K_s I_{sj} (n \cdot h_j)^q] + K_e$$

La **reflexión difusa ideal** asume que la luz cuando pega en un punto se distribuye uniformemente para todas las direcciones. Luego, el vector del observador no influye en esta ecuación: lo mires por donde lo mires se va a ver igual. Lo que sí influye es el ángulo en el que le llega la luz. Si la luz le llega al objeto de frente, la componente difusa va a ser máxima. En cambio, si la luz le llega de perfil se va a ver muy poco iluminada (la luz rebota menos).



La **reflexión ambiente** representa la iluminación de origen incierto, provista por el ambiente. Es una componente constante, con una proporción reducida del color del objeto, para que las zonas que no reciben luz directa no queden negras. Se consideran incidentes en cualquier punto con igual intensidad.

La **reflexión especular no ideal** tiene que ver con cuánto de la luz reflejada llega a la cámara. Es no ideal porque no se va a tener en cuenta sólo el vector que se refleja en la dirección exacta sino también los que se reflejan en direcciones cercanas. Como particularidad, tiene una componente de brillo (q): el punto brillante es tanto más pequeño cuanto mayor es el exponente q del coseno del ángulo de desvío.

La **componente emisiva** representa la luz que emite el objeto por sí mismo, aun cuando todas las luces están apagadas. Define una intensidad de luz emitida en cada punto del objeto y por igual en cualquier dirección. Se utiliza para modelar fuego, luciérnagas, luces de neón, objetos que tienen luz propia.

Primero se tiene la componente ambiente general y la emisiva, luego por cada fuente de luz se suman sus respectivas componentes difusa y especular. Las componentes K determinan la fracción de luz reflejada por el material (suelen estar en {1,1,1}), las componentes I definen la intensidad de la luz de cada reflexión.

En cuanto a los efectos que se observan en la realidad pero que este modelo no representa, está la posibilidad de tener dos objetos uno al lado del otro. Aunque uno tenga una componente emisiva, no va a iluminar al otro objeto porque es un modelo local. Otro efecto que no se representa por la naturaleza del método es el de la refracción de la luz por parte de los objetos traslúcidos.

¿Cómo se simula la niebla? ¿Para qué otra cosa sirve este efecto?

La aplicación de **niebla** se realiza mezclando el color f de la niebla (normalmente blanco) con el color de cada fragmento, mediante una interpolación lineal (promedio ponderado). El peso del color del fragmento es inversamente proporcional a la distancia del fragmento al ojo y se define mediante una función y sus parámetros. Si es lineal, entonces mientras más lejos esté el ojo, menos peso va a tener el color del fragmento y más el color blanco de la niebla, logrando el efecto de niebla en las lejanías.

También se puede usar para simular una escena bajo el agua: cuanto mayor es la distancia, menos peso tiene el fragmento y más tiene el del agua; y se usa para evitar tener que renderizar fondos muy complejos, simulando una visibilidad limitada para aumentar el sentido de profundidad.

¿Cuáles son las diferencias entre los sombreados flat (o facetado), Gouraud (o suave) y Phong? ¿En qué situaciones se hacen más/menos evidentes?

En el sombreado **flat** (izquierda), mediante la normal a la superficie y las direcciones hacia el ojo y hacia la luz, se calcula la iluminación en un punto (un vértice) y se asigna el color resultante a la cara (fragmento) a la que pertenece, por eso se ve facetado. Sólo es aceptable para poliedros simples o cuando se quiere resaltar la subdivisión en primitivas. También es valioso como alternativa rápida, cuando la imagen es intermedia y no final.

En el método de **Gouraud** (medio), se calcula la iluminación en cada vértice y los colores resultantes se interpolan en los fragmentos interiores. Es un poco más costoso que el facetado, pero tiene un mejor resultado y sin un costo excesivo, ya que sólo son tres números más que tiene que interpolar la GPU. Sin embargo, tiene defectos visibles, sobre todo en la especular cuando las primitivas son grandes en relación a la curvatura del objeto.

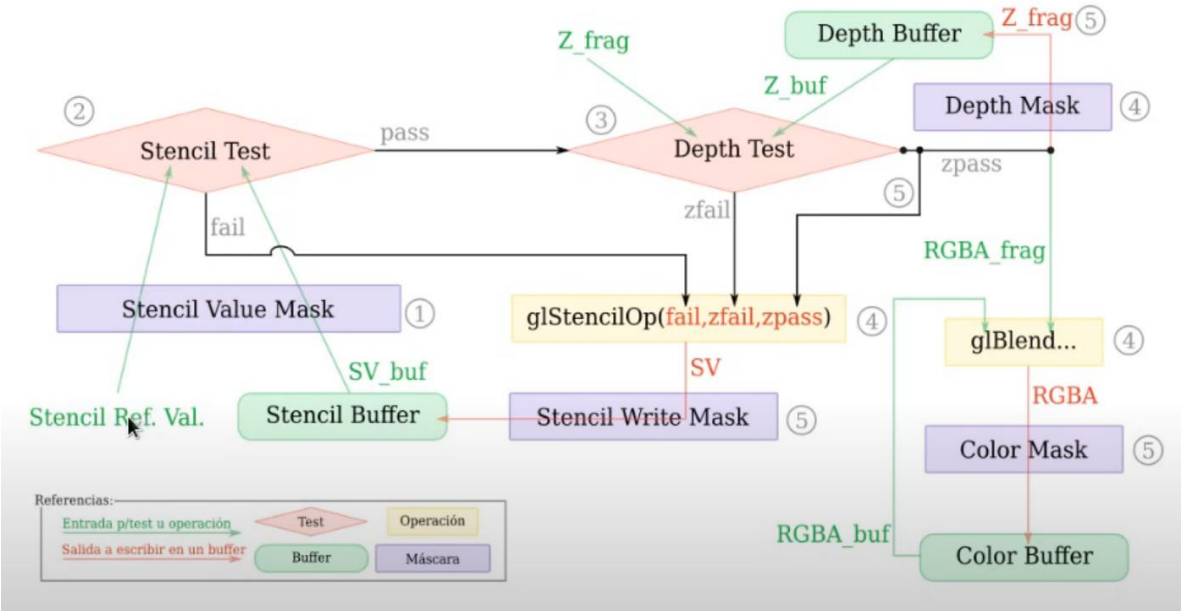
El mejor método es el de **Phong** (derecha), que consiste en interpolar las normales y calcular la iluminación en cada fragmento. Suele resultar muy costoso en tiempo y entorpecer el trabajo interactivo cuando la malla es muy densa, pero da el mejor resultado.

UNIDAD 6: BUFFERS

¿Qué diferencia hay entre fragmento y píxel?

Pixel es 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, la rasterización muestrea valores de color, coordenadas de textura, entre 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 píxel, mezclando o no sus valores con los que tenía el pixel. Ambos contienen valores asociados a una determinada posición elemental de una imagen de WxH píxeles.

Describa el funcionamiento conjunto del stencil-buffer y el z-buffer. ¿Qué se compara con qué en cada uno? ¿Cómo y cuándo se graba un dato en cada uno?



El **stencil buffer** actúa como una máscara a la hora de dibujar algo en pantalla. Este buffer almacena números enteros por cada bit que después serán interpretados para lograr distintos efectos dependiendo del valor del mismo. Para poder modificar el buffer se establecen condiciones antes de dibujar: se establece con qué condición pasa el stencil test o no, y luego definir qué hacer con los fragmentos que no pasaron el test.

Para poder enmascarar una primera figura, a modo de ejemplo simple, se puede establecer que el test falle siempre con un valor de referencia 1 (el valor de referencia que se comparará con el valor almacenado en el buffer) y luego que, cuando el test falle, se actualice el valor del stencil buffer. De esta forma, lo que se mande a dibujar no se mostrará en pantalla (porque falla el test y no llega al buffer de color), pero quedará en el stencil con valor 1. Así, si luego se quiere mostrar algo en pantalla con la forma establecida previamente en el stencil, sólo se debe hacer que el stencil pase sólo donde hay un 1 en el buffer, con una máscara en ~0 (de manera que no influya).

También se pueden configurar máscaras que indiquen qué bits serán actualizados en una operación que modifique al stencil buffer, para lograr distintos efectos. Si tengo dos superficies A y B, y quiero diferenciar las partes en las que se ve la A sola, la B sola y la intersección entre ambas, puedo setear la máscara en 01 para cuando mande a fallar el test con A, 10 para la B y pueda dibujar la intersección con la máscara en 11.

Si un fragmento pasa el stencil buffer, llega al **z-buffer** o **depth-buffer**. Este sirve para que, si un fragmento está detrás de otro, sea descartado ya que no sería visible. Esto permite dibujar los fragmentos sin necesidad de ordenarlos. Se asocia a todos los fragmentos un valor de z que está entre 0 y 1, siendo 0 lo más cercano (z-near) a la cámara y 1 lo más lejano (z-far). Inicialmente el buffer tiene valor del z-far y si el fragmento a rasterizar es menor al valor almacenado en el buffer, pasa al color buffer y se actualiza el valor del z-buffer con el valor de z del mismo.

Este algoritmo tiene dos problemas:

- **Transparencias múltiples:** Si tengo dos objetos opacos, puedo mandar a rasterizarlos en cualquier orden sin problema. Si tengo objetos opacos y uno transparente, debo enviar a rasterizar primero los opacos y recién después el transparente. Pero si tengo muchos objetos transparentes, tengo que saber cuál va primero. Para esto se puede realizar un ordenamiento como el BSP y usar el algoritmo del pintor: voy en la recursividad eligiendo el hijo que está más lejos del a cámara (si tengo la ecuación del plano, meto las coordenadas de la cámara y si me da negativo, voy por ese).
- **Z-fighting:** Supongamos que el z-buffer tiene una precisión de 2 bits, cuando guardo el z de un fragmento, se redondea a una de las 4 combinaciones de 2 bits: 00, 01, 10, 11. Si están muy cerca, pueden caer en el mismo valor por tener tan poca definición en el z-buffer.
  Para arreglar esto, si se quién va adelante, puedo hacer un polygonOffset que me falsea el valor de z. Si no sé quién va adelante, tengo que reacomodar el z-near y el z-far para no dejar tanto espacio entre ellos. Acerco los planos para que me funcionen las cuatro divisiones bien.

Explique cómo y para qué utilizaría el test de alpha de OpenGL. ¿En qué casos aventaja al Blending, cuando este último podría dar el mismo resultado? ¿Y desventajas?

El **alpha test** compara el alpha que trae el fragmento (junto con el color: RGBA) con un valor de referencia que se define al habilitar el test. Si quiero pasen el test los fragmentos opacos, mi valor de referencia será alpha = 1 y la condición GL_EQUAL. Si quiero que pasen los fragmentos transparentes, mi valor de referencia será alpha = 0 con la misma condición. Generalmente se utiliza en ese orden, aunque primero haciendo un ordenamiento de los fragmentos transparentes según cuál está más cerca de la cámara.

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 ventaja del alpha test ante el blending es que me va a descartar el fragmento. Por ejemplo, si descarto los fragmentos con alpha = 0 no se va a ver en pantalla y no va a causar problemas en el z-buffer porque no pasó. Si no tenés alpha test, dejás que pase un fragmento con alpha = 0, no se va

a ver en pantalla porque el blending va a mezclar 0 de ese fragmento contra lo que había. Pero como llegó al final, me arruinó el z-buffer y actualizó el valor de z a pesar de que no se ve.

La desventaja del alpha ante blending es que el alpha es un test binario: sí o no. ¿Qué pasa con un fragmento que vale 0,5, por ejemplo? Si quiero lograr efectos de transparencia, sí o sí me lo hace el blending.

¿Para qué sirve el accumulation buffer? Algoritmo que genere una imagen con sensación de movimiento rápido (motion-blur)

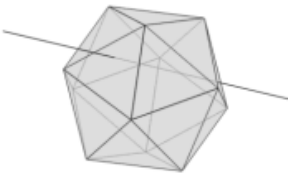
El **accumulation buffer** se utiliza para superponer imágenes, como si fuesen distintas manos de pintura con efecto acumulado sobre una misma superficie. No puede manipularse píxel a píxel como el resto, se le pasa (varias veces) un buffer de color entero (atenuado o no) y el resultado de las operaciones se transfiere entero al buffer de color. 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 la transferir.

Sirve para hacer el borroneado de movimiento (motion-blur) dando la impresión de velocidad: se promedia 5 o 6 fotogramas con la cámara en posiciones ligeramente distintas, por ejemplo, los fotogramas anteriores si se trata de un auto en movimiento. También se puede utilizar para hacer antialiasing, para simular la profundidad de campo fotográfico o para hacer sombras blandas.

¿Qué es el color-buffer?

El **color-buffer** almacena WxH píxeles de color, es el buffer que se actualiza cuando un fragmento es visible. Esto es, en este buffer se arma la imagen que vemos.

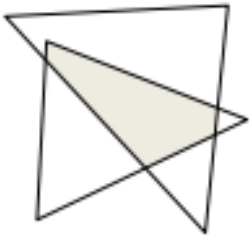
¿Cómo se logra el efecto de la imagen?



Cuando empiezo tengo todo en z-far y todo está adelante. Mando a dibujar el relleno, enmascarando color y obtengo un valor de referencia.

```
glPolyMode(FILL) – Sólo el relleno.
ColorMask(F,F,F) – Pongo false para que no pase de la color mask.
Dibujar() – Obtengo mi valor de referencia.
ColorMask(T,T,T) – Vuelvo a activar la máscara de color.
glDepthFunction(GL_GREATER) – Pasan solo las cosas que estén atrás.
glPolyMode(GL_FRONT_AND_BACK,GL_LINES) – Pongo para que pasen los bordes.
glWidth(1) – Línea más fina para las de atrás
Dibujar() – Dibujo lo de atrás.
glPolygonOffset – Offset para dibujar lo de adelante, sino queda en el mismo z
que el relleno y se va a ver mal.
glWidth(2) – Lo de adelante tiene la línea más gruesa.
glDepthFunction(GL_EQUAL) – Cuando es igual al valor de referencia del relleno.
Dibujar() – Dibujo lo de adelante.
```

Un programa permite que el usuario mueva los vértices de dos triángulos y, mientras lo hace, muestra las aristas y la intersección rellena. ¿Cómo logra ese efecto mediante una técnica raster?



Puedo lograr este efecto de dos maneras haciendo uso del test de stencil. Por un lado, usando la StencilMask: si dibujo un triángulo y le digo al stencil buffer que se actualice en el segundo bit (10), y luego dibujo el otro triángulo y le digo al stencil buffer que se actualice en el primer bit (01), la intersección que definen va a actualizar los dos primeros bits (11).

Por otro lado, puedo hacer lo siguiente:

```
StencilFunct(GL_NEVER,1,-0);
Dibujar(triangulo_1);
StencilFunct(GL_EQUAL,1,-0);
PolygonMode(GL_FRONT_AND_BACK, GL_FILL);
Dibujar(triangulo_2); -- dibujo el segundo triángulo sobre el stencil del
primero, con relleno.
PolygonMode(GL_FRONT_AND_BACK,GL_LINES);
StencilFunct(GL_ALWAYS,1,-0); -- dibujo los dos triángulos sin relleno y sin el
stencil.
Dibujar(triangulo_1)
Dibujar(triangulo_2)
```

UNIDAD 7: CURVAS

¿Qué es una curva? ¿De qué formas se puede representar una curva?

Una **curva** es la imagen de una función continua desde un intervalo real al espacio euclídeo. Resulta un conjunto continuo y unidimensional de puntos, cuyas coordenadas (dos o tres) varían como funciones continuas de un parámetro real.

Las formas de representarlas son:

- **Explícita:** y = f(x). Fácil de graficar, pero no todas son representables de esta manera.
- **Implícita:** f(x,y) = 0. Difícil de graficar porque no se obtienen los puntos de manera directa.
- **Paramétrica:** x = f(t), y = g(t). Fácil de graficar y puedo representar cualquier curva con el mapeo directo de cada t a un punto de ésta.

¿Qué es una curva de Bézier? ¿Qué propiedades tiene? ¿Cómo encuentro un parámetro t en la curva? ¿Cómo obtengo la derivada?

Una **curva de Bézier** es un método de definición de una curva en serie de potencias. Consiste en definir algunos puntos de control, a partir de los cuales se calculan los puntos de la curva.

Como propiedades, una curva de Bézier cumple con: tiene control global, todas sus derivadas existen y son continuas; no puede oscilar más que el polígono de control (variation diminishing, dado por la forma de su derivada); la curva no puede salirse del convex-hull porque los polinomios de Bernstein (pesos) suman uno; y la curva debe comenzar y terminar tangente al polígono de control. Además, tiene invariancia afín porque son combinaciones afines de los puntos de control, luego, para hacer una transformación afín de la curva solo hay que transformar sus puntos de control y construirla en el espacio transformado.

Se puede obtener un parámetro dado de una curva de Bézier haciendo el algoritmo de De Casteljau. Tomando el t, por ejemplo, t = 1/3, interpolo en cada segmento: $P(u) = (1-u) \cdot P_0 + u \cdot P_1$ donde P_i son los puntos de control y u ∈ [0,1]. A partir de este algoritmo, también puedo recuperar los polinomios de Bernstein que actúan como los pesos variables en función de u de cada punto de control.

La derivada de una curva de Bézier de grado n es una curva de Bézier de grado n-1 definida mediante puntos de control que son n veces las diferencias entre pares de puntos sucesivos. El último segmento de De Casteljau me da la derivada. Si estoy en superficies de Bézier, el último segmento de De Casteljau en un parámetro me da la derivada parcial en ese y, el que está a la misma altura en el otro parámetro me da la otra derivada parcial.

¿Qué métodos para interpolar puntos con splines hay?

Catmull-Rom, que entre cada par de puntos Pi y Pi+1 se inventan dos puntos de control intermedios que garanticen la continuidad paramétrica. El parámetro u se hace variar entre 0 y m, tomando el valor i cuando la curva pasa por Pi, es decir que P(i) = Pi. La velocidad local se define como la velocidad media de los tramos que se unen. Luego, cada punto dista de Pi un tercio de la derivada recién calculado. Lo malo de este método es el overshooting.

$$\mathbf{v}_i = \frac{dP(i)}{du} = \frac{\Delta P}{\Delta u} = \frac{P_{i+1} - P_{i-1}}{(i+1)-(i-2)} = \frac{P_{i+1} - P_{i-1}}{2} \qquad P_i^- = P_i - \mathbf{v}_i/3 \qquad P_i^+ = P_i + \mathbf{v}_i/3$$

Overhausser, que resigna la continuidad paramétrica y obtiene continuidad geométrica. El parámetro en el punto i no es i sino la suma de las longitudes de los segmentos hasta ese punto, de modo que la velocidad media en cada tramo es de módulo unitario. Para definir la dirección común, se realiza un promedio ponderado de las velocidades medias anterior y posterior, donde pese el punto más cercano. Los puntos ±1 van a distar de vi/3 la longitud del punto anterior al Pi.

$$\mathbf{v}_i^- = \frac{P_i - P_{i-1}}{|P_i - P_{i-1}|} \qquad \mathbf{v}_i^+ = \frac{P_{i+1} - P_i}{|P_{i+1} - P_i|} \qquad \mathbf{v}_i = \frac{|P_{i+1} - P_i|\mathbf{v}_i^- + |P_i - P_{i-1}|\mathbf{v}_i^+}{|P_i - P_{i-1}| + |P_{i+1} - P_i|}$$

$$P_i^- = P_i - |P_i - P_{i-1}|\mathbf{v}_i/3 \qquad P_i^+ = P_i + |P_{i+1} - P_i|\mathbf{v}_i/3$$

En ambos métodos se implica un tratamiento distinto en los extremos porque sólo se tiene un solo punto. Una posible respuesta es no interpolar los puntos inicial y final, pero la curva va desde el segundo hasta el ante-último. Otra opción es inventar los tramos iniciales y finales de la curva en base a un solo punto adicional.

¿Qué es la continuidad paramétrica y geométrica?

En el interior de cada tramo la continuidad es C(inf), se analiza la continuidad de las splines de Bézier en los puntos de unión entre cada curva.

La **continuidad paramétrica** implica que cada par sucesivo tenga una misma recta tangente, opuesta, en el punto de unión. La derivada respecto del parámetro unificado es continua. Si importa la velocidad de la curva, se requiere C1.

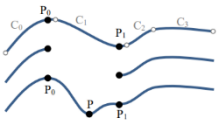
La **continuidad geométrica** implica que, para cada par sucesivo, las derivadas sean proporcionales y en el mismo sentido (proporción positiva). Se ignora el módulo de la velocidad de la curva. Es la continuidad visual, por decirle así.

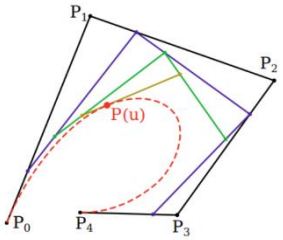
¿Qué son las curvas de Bézier racionales? ¿Qué son las B-splines? ¿Y las NURBS?

Una **curva de Bézier racional** está definida igual que antes pero mediante coordenadas homogéneas {x,y,u,w}. La transformación proyectiva de una curva de Bézier se hace por medio de la transformación lineal de sus puntos de contorl en R+1 y para graficarse se divide por w. La w actúa como un peso.

Las **B-splines** son splines en las que se utiliza una base polinómica, son curvas no-interpolantes, unidas con continuidad C2 y cuyos puntos de control poseen control local. Se definen con un vector de knots. Las **NURBS** son un tipo de B-splines, racional y no uniforme (pesos pueden repetirse o saltarse), proyectadas en el espacio homogéneo. En los casos en los que se repite una componente, se pierde un grado de la derivada. Una NURBS de grado 3, por ejemplo, tiene continuidad hasta la segunda derivada. Si repito un knot en un punto, perdió la continuidad de la segunda derivada.

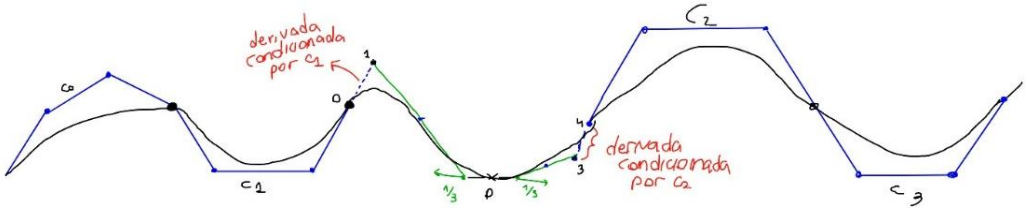
Describa cómo determina en que curva C_i y el parámetro t_i en la misma, donde clickea el usuario. ¿Cómo recorta la curva? ¿Cómo genera el nuevo tramo?





Le calculo el convex-hull a cada curva y veo si cae en ese, se pueden superponer los convex-hull, entonces subdivido/refino hasta una tolerancia mínima para encontrar t. Sino aproximo la curva con segmentitos y calculo la distancia de los puntos. Para cortar la curva, hago De Casteljau con el t encontrado y corto donde encuentra el punto:

Las dos primeras derivadas vienen dadas por las curvas adyacentes. Después me tengo que inventar las otras derivadas. Las puedo calcular con derivada promedio. Plantear los peros (contras, qué más puedo usar):



¿Cómo parametrizamos una superficie? ¿Qué tipos de superficies hay?

Como las curvas, las superficies también debemos parametrizarlas. Se hace definiéndola como una función de dos parámetros (u,v). En general, se tiene una grilla regular de cuadriláteros, por lo cual va a ser útil la interpolación bilineal: en cada cuadradito se interpola bilinealmente con (u,v) formando nuevos cuadraditos, donde se repite el proceso en forma recursiva hasta llegar al punto. La última superficie bilineal es tangente a la superficie, por lo que la normal se puede calcular con el producto cruz de los segmentos u = cte y v = cte de la última interpolación bilineal. Si fijo un u, puedo obtener una curva isoparamétrica P(v) con v variable (y viceversa) y también la puedo calcular con De Casteljau en ambos parámetros: el último segmento me va a dar la derivada parcial y después les hago producto cruz.

Las superficies que hay son:

- **Bilineales:** definidas por cuatro puntos, que son los vértices.
- **Reglada:** Se puede recorrer toda la superficie en una recta móvil. Las dos curvas son distintas, pero están parametrizadas.
- **Cono:** Las rectas se cortan todas en un punto.
- **Cilíndrica:** Si la generatriz se traslada paralela a sí misma.
- **Superficie de revolución:** Si un conjunto de curvas isoparamétricas (u o v) son círculos de eje común. Los puntos de control se copian girados sobre una circunferencia.
- **Coons:** Cuatro curvas unidas en un circuito cerrado y se genera una superficie interior sin definir puntos de control internos. Se suman las regladas de cada par opuesto y se resta la bilineal de los cuatro vértices.

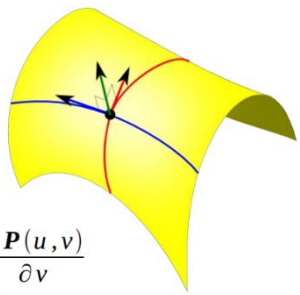
¿Cómo se rasteriza una superficie una superficie NURBS? ¿Cómo encuentro un punto? ¿Cómo encuentro la normal en ese punto?

Una **superficie NURBS** se rasteriza subdividiendo la grilla que va de 0 a 1 en u y t, en pequeños quads: hago un for para la u, un for para la v, armo una grilla y armo quads.

Para encontrar un punto en una superficie NURBS puedo hacer interpolación bilineal ya que tengo los dos parámetros u y v: en la grilla, divido recursivamente las grillas más pequeñas que me van quedado hasta encontrar el punto (en vez de un punto de control, en cada grilla pierdo una fila y una columna). Sino puedo hacer De Casteljau en ambas direcciones: primero hago De Casteljau con u hasta encontrar el punto y luego, sobre el polígono de control que se formó con los puntos encontrados con ese parámetro, hago de Casteljau con v y encuentro el punto.

Para encontrar la normal en el punto, puedo hacer producto cruz de las dos derivadas parciales, que las encuentro con los dos últimos segmentos de De Casteljau multiplicados por el grado de Bezier.

Tangentes y Normales:



$$\hat{\boldsymbol{N}}(u,v)=\frac{\partial \boldsymbol{P}(u,v)}{\partial u}\times\frac{\partial \boldsymbol{P}(u,v)}{\partial v}$$

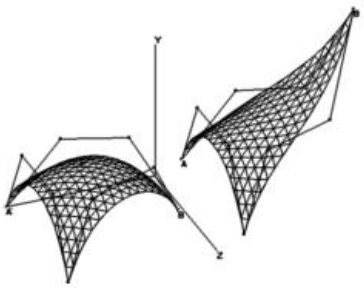
$$\boldsymbol{N}(u,v)=\frac{\hat{\boldsymbol{N}}(u,v)}{\left|\hat{\boldsymbol{N}}(u,v)\right|}$$

¿Qué son las trimmed nurbs? ¿Qué son las subdivision surfaces?

Las **trimmed NURBS** permiten recortar una superficie NURBS por medio de splines definidas en el espacio de parámetros. Lo malo de esto es que al armar la triangulación algunos cuadriláteros quedan recortados, como triangulitos aplastados y no está bueno para iluminación. Además, lo que dibujo en el corte de abajo no sale igual al corte de arriba, es difícil traducir eso y hacer el corte que de verdad quiero.

Las **subdivision surfaces** consisten en tener una forma muy simple y refinarla en un proceso recursivo hasta obtener la superficie buscada. Para esto se usan el método de Doo-Sabin y Catmull-Clark.

El método de Coons sirve para definir una superficie bi-paramétrica mediante cuatro curvas paramétricas unidas en un circuito que define el borde de la superficie. Consiste en sumar las superficies regladas de pares opuestos y restar la bilineal de los cuatro vértices. ¿Cómo haría un método del mismo estilo con tres curvas unidas en circuito?



UNIDAD 8: TEXTURAS

¿Qué es una textura? ¿Qué diferencia hay con imagen?

Una **textura** es una imagen de una textura compleja que se aplica sobre una superficie simple: una matriz de colores que sirve para darle detalle a los modelos sin tener que hacer 5000 triangulitos más. La diferencia con imagen depende del uso que le doy.

¿Cómo se tratan las coordenadas fuera del rango? (wrapping)

Hay distintos métodos de **wrapping**: clamping y repeat. **Clamping** toma todo valor menor que cero como cero y todo valor mayor que uno, luego, las últimas filas y columnas de la textura se van a repetir hasta el final de la superficie. **Repeat** toma la textura y la repite en los espacios donde $0 < s < 1$ y $0 < t < 1$: $s = s - \text{int}(s)$.

¿Qué métodos de mapeo hay?

- **Mapeo manual:** Se le asigna las coordenadas de textura a cada vértice de cada primitiva. Puedo asignar coordenadas negativas si quiero que quede dentro del objeto y que no encuadre todo el objeto.
- **Mapeo automático:**
    - **Mapeo plano:** Consiste en asignar un plano a cada coordenada de textura, que se define en función de la distancia del punto al plano. Ax+By+Cz+D = 0, donde {A,B,C} es el vector normal al plano y la D es la función lineal de la distancia del plano al origen: indica la posición del plano y, con ello, la de la textura. A mayor D, más chica se verá la textura.
    - **Mapeo en dos partes:** Consiste en mapear la textura sobre una superficie intermedia y luego cada punto de esa superficie intermedia con cada punto del objeto (o-mapping). Por ejemplo: cilíndrica, esférica (cambio las coordenadas cartesianas por las cilíndricas/esféricas), cúbica (ambiente, seis fotos del entorno y se usa rayos reflectados).

¿Qué es el filtrado?

El **filtrado** de una textura es lo que se hace cuando queremos que las transiciones sean más suaves y agradables. En lugar de elegir el color del téxel (píxel de la imagen de textura) donde cae el punto (s,t), se realiza un promedio ponderado de vecinos. En el texel en donde cae el punto, busco los vecinos más cercanos y hago mi promedio con la interpolación bilineal ya que estoy tratando con grillas de cuadraditos.

En cuanto a las técnicas para ver qué texel asignar: nearest (uso el texel que cae justo en el centro, da un serruchado espantoso que se disimula si no hago zoom), interpolado (interpolo los colores que rodean al píxel.), linear (pondera unos cuantos alrededor del pixel), mipmaps (versiones reducidas de la misma textura, busco un texel que se corresponda con el tamaño del pixel entre los distintos y después aplico linear). Cuando no se comporta igual en todas las direcciones (píxel no está en un cuadrado lindo de téxeles), hago un filtrado isotrópico.

Magnification: cada téxel cubre más de un píxel. Tengo que hacer una técnica de filtrado. Puedo usar mipmaps, promediando de a cuatro píxeles y, cuando voy a renderizar, elijo la que tenga una medida de píxel igual (o lo más parecida posible) a la del téxel. También puedo promediar dos mipmaps. Luego, uso nearest o linear que van a andar bien porque tengo toda la información necesaria acumulada.

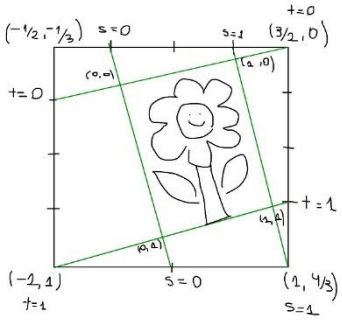
Minification: cada píxel cubre más de un téxel. Generalmente se reduce la imagen original varias veces a la mitad promediando vecinos y finalmente promediar promedios hasta llegar al tamaño adecuado.

¿Qué es la mezcla?

Puedo mezclar los colores de la textura con los del fragmento de distintas formas:

- **Replace** (sustitución, queda el color de la textura).
- **Decal** (mezcla con transparencia, donde la textura tiene α -79h-= 0 quedan den color del fragmento).
- **Modulate** (se multiplica el color y el alfa del fragmento por el de la textura, así se atenúa).
- **Blend** (mezclo utilizando un tercer color fijo, los valores de la textura se usan como parámetro para interpolar en cada canal).

Los vértices de un cuadrado tienen las coordenadas de textura que indica la figura. La imagen de la flor se utiliza como textura mediante clamping (no se repite, aunque el tallo llega al borde inferior). Dibuje el resultado (indique el marco y esquematice la posición).



Voy buscando los s = 0, s = 1, t = 0, t = 1 interpolando linealmente en los bordes de la figura. Armo las diagonales y encontré el resultado.

¿En qué consiste el mapeo de texturas en 2 partes? Desarrolle un método (ecuaciones incluidas) para aplicar un mapeo en 2 partes utilizando un cilindro.

El mapeo de texturas en 2 partes consiste en mapear manual o algorítmicamente (UV) la textura sobre una superficie intermediaria muy simple: un plano, esfera, cubo o cilindro; para luego identificar de forma sencilla cada punto del objeto con algún punto de la superficie intermedia. Tiene un primer mapeo sobre la superficie intermediaria, el S-mapping, y el segundo es el O-mapping, sobre el objeto.

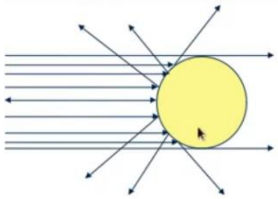
Para el cilindro, como es una superficie biparamétrica P(u,v), se mapea la coordenada s en el rango de u y la t en el rango de v. En una esfera, se puede considerar que s es la longitud $\theta = \text{atan2}(y,x) \in [0,2\pi]$ y t la latitud $\varphi = \text{atan2}(z,\text{sqrt}(x2+y2)) \in [-\pi,\pi]$ => $s = \theta/(2\pi)$; $t = ½+\varphi/\pi$. Para un cilindro, s es la altura en el punto (x,y) y t es el ángulo resultante de hacer atan2(y,x). Hay que recordar dividir s por la altura total y t por 360 para poder tener s,t

¿Cómo funciona el o-mapping cúbico?

Tengo un objeto y tengo el cubo, dependo de donde está la cámara. Alguna luz viene de una dirección (pasa por un punto de la textura), se refleja en el objeto y llega al ojo. El punto de la textura del que viene es el que se va a ver reflejado en el punto en donde se reflejó. Tengo que poder tirar rayos desde la cámara a los vértices, que esos rayos reboten y ver con qué cortan:

La técnica del **mapeo cúbico** del ambiente tiene la gran ventaja de que el objeto intermediario se realiza con seis imágenes muy fáciles de obtener, pero no está exento de algunos problemas técnicos, pues las coordenadas de textura se deben asignar de acuerdo al rayo reflejado “en cada fragmento”, no se pueden interpolar las coordenadas de textura calculando el reflejo sólo en los vértices de una primitiva pues puede suceder que cada vértice caiga en una cara distinta del cubo, generando problemas con la asignación de dos coordenadas en el cubo desarrollado sobre un plano (figura derecha); aun si se resuelve eso, se pierde la suavidad al pasar de una cara a otra.

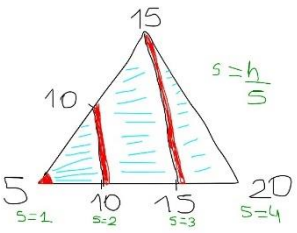
¿Cómo realiza, a partir de fotos, el O-mapping de un cielo sobre una semiesfera? Suponga que ya tiene la posición de un fragmento y la dirección del rayo reflejado ¿Cómo calcula en qué téxel cae y, si no hizo una única imagen, en qué foto hay que buscar ese téxel?



El **mapeo de entorno esférico** es un truco para simular un mapeo de entorno por rayo reflejado. Utiliza una textura consistente en una imagen del entorno como si fuese una foto de una esfera espejada. Esa imagen se puede construir por medio de un algoritmo que la genere a partir de las seis caras de un cubo, también se puede generar falsa, pero creíblemente, deformando una sola foto. Contando con esa imagen y suponiendo al observador en el infinito, basta conocer la normal del objeto en el espacio del ojo, para definir en forma analítica las coordenadas de textura. La obvia desventaja es que solo funciona con cámara fija, si se quiere mostrar el objeto girando queda muy bien, pero si se quiere mover la cámara en una escena fija el objeto siempre refleja lo mismo en forma independiente del punto de vista, como si el universo detrás de la cámara se moviese con la cámara.

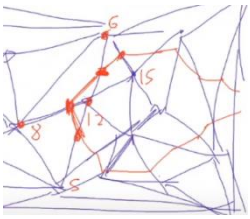
Se tiene un conjunto de mediciones de alturas en N puntos de un terreno. Se quiere graficar iso-lineas/isocurvas de altura constante. Describa un método basado en texturas para hacerlo. Describa un método vectorial para generarlas como splines. ¿Qué ventajas/desventajas presenta cada método?

Para resolver este problema con texturas, puedo utilizar una textura 1D como la de la izquierda. Puedo elegir coordenadas de textura en cada vértice de forma tal que me quede pintada una línea al inicio de cada isolínea.



La textura está en repeat, cuando llega a s = 1, s= 2, s = 3 y así, se vuelve a repetir de s = 0 a s = 1 y por eso se logra ese efecto.

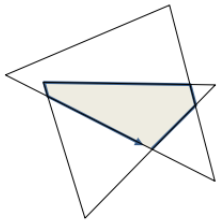
Si quiero resolverlo de forma vectorial, para interpolar la altura en cualquier punto: armo una triangulación con los puntos de los cuales tengo el dato de su nivel/altura, para cada punto donde yo quiera la altura veo en qué triangulo cae y, en ese triangulito, uso las coordenadas baricéntricas.



UNIDAD 9: ORDENAMIENTO

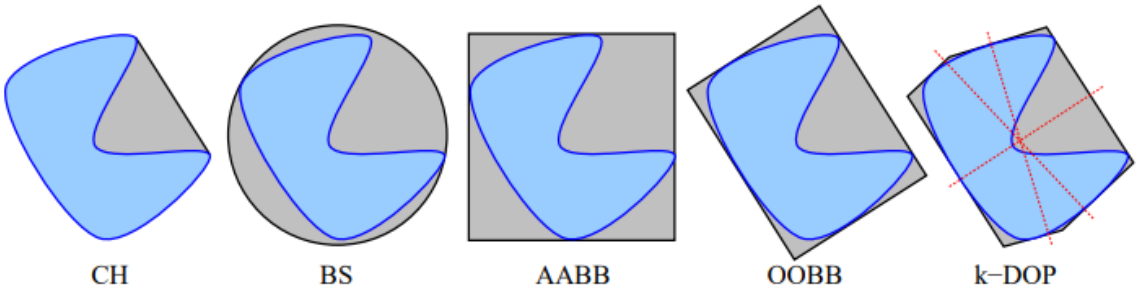
¿Cómo se calcula la intersección entre dos triángulos en 2D de modo que quede definida su frontera poligonal ordenada, con el interior a la izquierda?

Para encontrar la intersección entre dos triángulos en 2D debo hacer la intersección segmento a segmento con cada par de aristas de los triángulos. Por cada punto de intersección que encuentre voy a ir guardándolo y voy a moverme hacia el punto del cual no vengo, hasta llegar al primer punto que guardé (corto, ya definí el polígono).

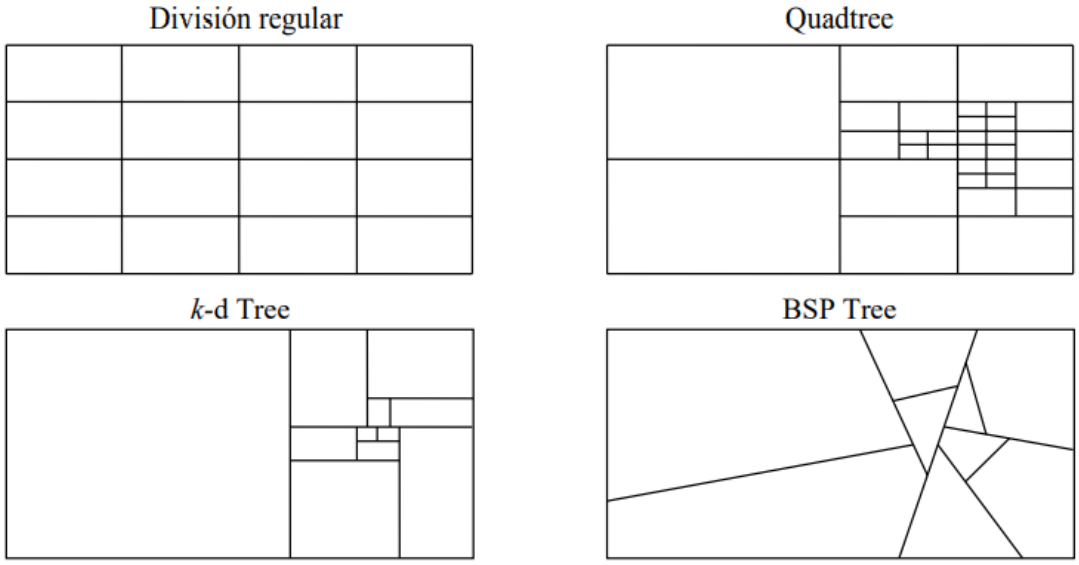


¿Qué es el bounding box de un objeto?

Existen distintos tipos de **bounding box** (envoltorios convexos simples) que se pueden darle a los objetos complicados. El convex-hull es el mínimo envoltorio convexo; la esfera envolvente mínima; el axis-aligned bounding-box (caras paralelas a los planos de coordenadas); oriented bounding-box; k-discrete oriented polytope que tiene k pares de caras en direcciones dadas.



¿Qué métodos de ordenamiento espacial hay?



La **división regular** refina siempre en partes iguales. Problema: espacios vacíos, puede tener todos los puntos en uno solo. Ventajas: Fácil de implementar, sencillo y óptimo al buscar vecinos, genial si tengo los puntos equiespaciados.

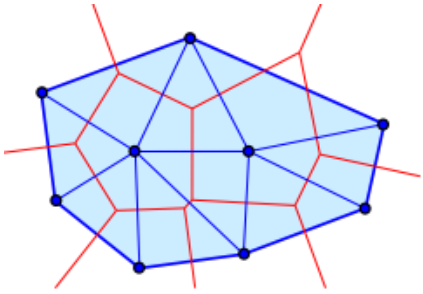
Los **quadtree** (y **octrees** en 3D) refinan solo en las cajas que contienen o interceptan más de determinada cantidad de objetos. Cada división es O(1) pero termina siendo O(n). El árbol tiene bastante profundidad.

Los **k-d tree** dividen de a uno, donde haya algún elemento. Calcular la mediana es de orden O(n). Como ventaja, es más sencillo de trabajar y produce árboles balanceados: menos subdivisiones y menor profundidad total del árbol. Se usan para cosas fijas porque es caro construir el árbol.

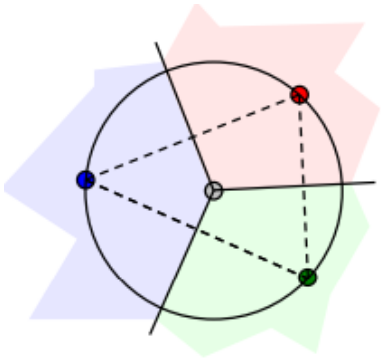
Los **BSP tree** dividen con la dirección que se le cante, o sólo alineado con las cajas. Uso cuando los datos ya me dan una división precocida pero es costoso de construir. Con este árbol, si tengo la posición de la cámara, puedo utilizar el algoritmo del pintor para recorrerlo: voy en la recursividad eligiendo el hijo que está más lejos de la cámara y lo anoto, la lista resultante es mi orden de rasterización con garantía de no dibujar algo que está tapado. Se usa para cosas fijas: si me cambian de lugar un nodo, tengo que reconstruir medio árbol y es costoso.

BVH no es partición como los otros, sumo las partes y puede no darme el todo o puede repetirsen partes. Hago cajas que contienen a los objetos, pudiendo solaparse. Puedo seguir dividiendo pero tengo que tener un criterio para hacerlo y está bueno cuando tengo objetos móviles (no fijos) porque si se mueve un punto, puedo agrandar o achicar la cajita y listo. Árbol es fácil de armar, pero la ineficiencia viene de recorrer el árbol ya que puede haber alguna superposición y tengo que recorrer los dos árboles.

Voronoi consiste en dividir el espacio en celdas, una para cada objeto, de modo que cada celda contiene los puntos del espacio que están más cerca de su objeto que de cualquier otro. Las aristas del diagrama separan dos celdas y equidistan de sus nodos. Los vértices, donde se juntan tres mediatrices, son los centros de una circunferencia que inscribe al triángulo formado por los tres nodos.



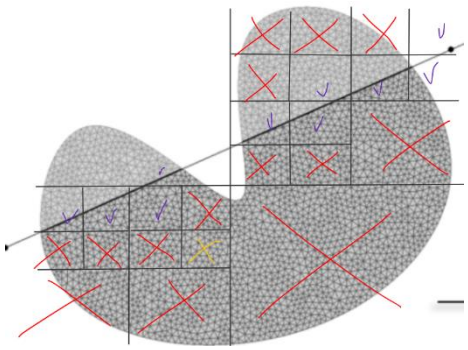
La triangulación **Delaunay** tiene como vértice al centro de la circunferencia que equidista de los nodos y define el triángulo. Para saber si una triangulación es Delaunay o no, me fijo si los círculos contienen nodos en el interior: en Delaunay NO contienen nodos en el interior.



Como ventajas, podemos averiguar en forma muy eficiente cuales son los nodos cercanos a un determinado punto, averiguando a qué triángulo pertenece. Tiene aplicación en ejercicios de problemas, ejemplo: farmacia, comisarías.

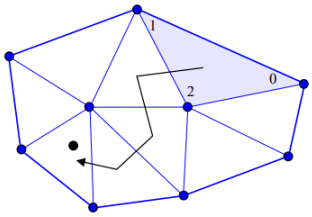
Si tengo que encontrar la intersección triángulo-segmento y tengo muchísimos triángulos que forman una triangulación, ¿cómo ahorro cálculos y hago el programa más eficiente?

Para ahorrar cálculos y no tener que hacer los cálculos de intersección con todos los triángulos de la triangulación, puedo aplicar un método de ordenamiento. Por ejemplo, usando quadtrees, puedo ir haciendo los quads y descartando aquellos que no contienen a la recta. Una vez que ya no puedo dividir más o tras una cierta tolerancia en la recursividad, puedo hacer la intersección triángulo-segmento sólo con los que están dentro de los quads que me quedaron.



¿Qué es la búsqueda lineal?

Tenemos una triangulación de un conjunto de nodos y queremos consultar a qué triángulo pertenece un punto del plano. El procedimineto consiste en partir de un triángulo cualquiera y movernos por vecindades, de un modo recursivo que nos acerque cada vez más al punto.



Para ello, debemos constar con una estructura de datos que identifique, para cada triángulo, cuáles son sus vecinos. Ordenamos la lista de modo que el vecino i-ésimo sea opuesto al i-ésimo nodo del triángulo. Ejemplo: {0, 1, 2}. Partimos del triángulo pintado y calculamos las coordenadas baricéntricas en el punto (sin dividir por el área total). La más negativa nos indica a qué vecino pasar. Cuando son todas positivas, se encontró el triángulo que lo contiene. Para acelerar esta búsqueda, se precalcula con un ordenamiento espacial para el conjunto de nodos para así poder partir de un triángulo cercano.