Ej1: Z-Buffer

Comparado con algoritmo del pintor (+ intersecciones si las primitivas pueden atravesarse entre sí):

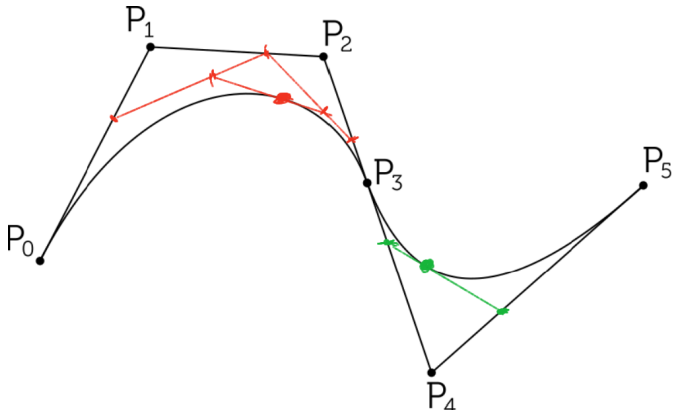
- Ventaja: No requiere ordenar ni calcular las intersecciones
- Desventajas: Consume más memoria (el buffer) y obliga a definir límites artificiales para lo que "se ve" en z (z-near y z-far) [1].

Problemas:

- Transparencias: que lo que está detrás de una transparencia no se vea (se descarta si se rasteriza después), o que dos transparencias se mezclen mal.
    - Si no se superponen, renderizo primero los objetos opacos, luego los transparentes, y funciona bien.
    - Si se superponen ("múltilpes"), renderizar lo transparente sin actualizar el buffer da un aproximación (se mezclan los colores que deben) pero no es del todo correcta (los pesos de la mezcla pueden estar mal).
- Precisión (z-fighting): cuando el resultado de comparar el z de dos fragmentos se ve alterado por errores numéricos (dos triángulos superpuestos en el mismo plano, o casi).
    - Hay que ajustar los planos z-near y z-far lo mejor posible a la escena (especialmente near, por la no-linealidad del z)
    - Si puedo predecir qué dos objetos van a "pelear" y se cual quiero que gane, perturbo el z de uno de ellos (polygon offset).
    - Codificar el valor de z de forma que el número tenga más precisión cerca del ojo (donde un error es más notorio) [2].

Ej2: Curvas

a) Como las Bezier interpolan los extremos, tiene que ser en P_3 donde termina un tramo (digamos $C_1(t)$) y comienza el otro ($C_2(t)$). Entonces un $t \in [0; 0.5]$ para $S(t)$ debe ser mapeado a un $t \in [0; 1]$ para $C_1(t)$ y un $t \in [0.5; 1]$ para $S(t)$ debe ser mapeado a un $t \in [0; 1]$ para $C_2(t)$ [3]. O sea que $S(\frac{1}{3})$ se encuentra evaluando $C_1(\frac{2}{3})$ (rojo en la figura); y $S(\frac{2}{3})$ se encuentra evaluando $C_2(\frac{1}{3})$ (verde en la figura):



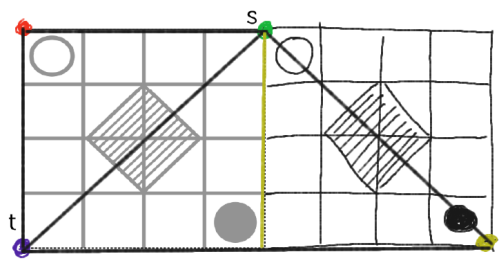
b) P_2, P_3 y P_4 están alineados así que mínimamente hay continuidad geométrica. Pero parece haber paramétrica también: para que esto ocurra las derivadas deben ser iguales también en magnitud. Las derivadas en ese punto salen de los segmentos de los polígonos de control (el último de la 1ra curva, el 1ro de la 2do) multiplicado por el grado. El detalle es que por la cantidad de puntos, la primera curva debe ser de grado 3 y la segunda de grado 2, entonces los segmentos no deben ser iguales, hay continuidad paramétrica si $3 * (P_3 - P_2) = 2 * (P_4 - P_3)$.

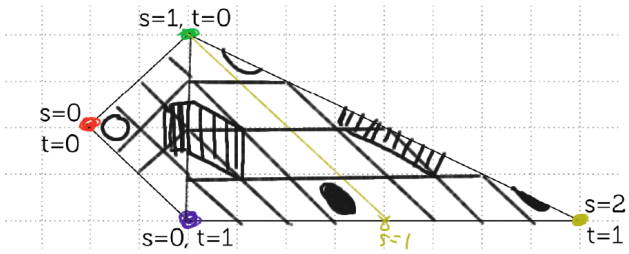
c) P_0, P_1 y P_5 pueden moverse libremente sin perder continuidad en la spline. Los 3 puntos que deben mantener una relación son P_2, P_3 y P_4, siendo la relación la fórmula que se menciona en b). De esa fórmula puedo despejar P_2 o P_4 para recalcularlo cuando alguno de los otros 2 se mueve y recuperar la continuidad. Cuando se mueve P_3 puedo hacer algo más simple que es aplicarle el mismo desplazamiento a P_2 y P_4.

d) No con esos puntos de control, porque son tramos de diferente grado. Sí podría encontrar una definición de 3er grado para la 2da curva (algoritmo de "elevación de grado") y entonces plantear una B-Spline con un vector de knots { 0, 0, 0, 0.5, 0.5, 0.5, 1, 1, 1 } para que coincida en forma y parametrización.

Ej3: Mapeo de Textura

Es un cuadrilátero, así que hay que dividirlo en dos triángulos para rasterizarlo (el pipeline rasteriza por triángulo, y además adentro interpola linealmente [4]). Queda mejor si divido por la diagonal vertical (ver ejercicio 5). Luego ubico las coord. de textura de los vértices sobre la imagen de la textura y dibujo ahí los triángulos para tener referencia de qué parte debo mapear en cada uno (dado que hay una coordenada s=2, para hacer más simple el repeat dibujo la repetición en el espacio de la textura). Y luego simplemente "copio" lo que hay en cada triángulo de la derecha sobre los de la izquierda, teniendo cuidado de mantener proporciones dentro de cada uno (y de que como es lineal, las rectas deben seguir siendo rectas). Como ayuda también puedo marcar sobre el objeto el borde derecho de la textura (linea s=1) para tener más referencia.





Ej4: Ordenamiento Espacial

- Partición (ej: quad-tree, kd-tree, bsp-tree) vs no-partición (ej: bvh): cuando es partición (en particular cuando las subáreas no se solapan) es más fácil la búsqueda (especialmente de puntos, ya que un punto está en una sola hoja del árbol); pero es más difícil/costoso de actualizar. Si admito solapamiento, cuando algo se mueve puedo no actualizar la estructura (el árbol queda "menos óptimo", pero sigue siendo consistente/útil). El solapamiento también facilita poner cosas con área/volumen dentro del árbol sin tener que partirlas.

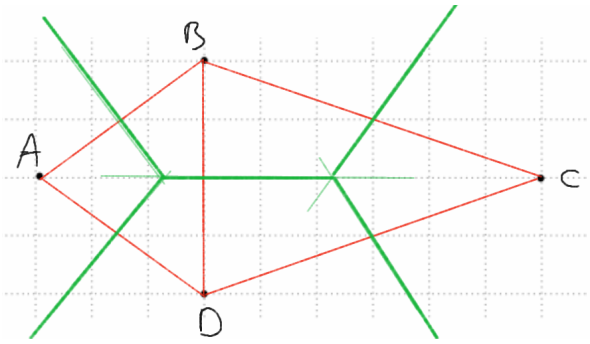
- Partir por la mitad (quadtree/octree) o no (bsp, kd): partir el espacio a la mitad es lo más rápido (para construir el árbol) y simple (ni siquiera necesito guardar el/los planos que dividen, es trivial recalcularlos durante el recorrido); pero no es lo mejor pensando en el balanceo del árbol (puede haber nodos vacíos con con "hermanos" muy poblados/subdivididos). Usar algo mejor (por ej, la mediana, como se hace habitualmente en el kd-tree) genera un árbol más balanceado (óptimo si es la mediana), pero también hace que sea más caro construirlo.
- Binario (BSP, KD, usualmente BVH) o no (quad/octree): no es una diferencia importante en cuanto a complejidad cuando en los no binarios igual el nro de hijos está fijo (como quad/octree). Dividir en más de 2 hace que las subáreas se reduzcan más rápido y eventualmente necesitemos menos niveles.. pero también los datos podrían estar mucho más "desparramados" en una dirección que en otra y convenir dividir solo esa.
- Planos alineados con los ejes (quadtree, kdtree) o no (bsp): En general es más simple pensar en planos alineados (luego verificar de qué lado está un pto es tan simple como comparar una sola coordenada); pero a veces tiene sentido usar una orientación que nos dan los propios datos para la división (por ej, si un objeto representa una pared, es probable que sea un buen plano para dividir el resto de los objetos).

Ej5: Delaunay/Voronoi

Voronoi: Un diagrama donde cada celda representa a todos los pts del espacio que están más cerca de un cierto nodo que de cualquier otro. Entonces los límites entre celdas son pts equidistantes a 2 (aristas) o más (3 en los vértices) nodos. La triangulación Delaunay es dual del diagrama de Voronoi. Esto es que hay ciertas relaciones que permiten pasar de uno a otro:

- Cada nodo de Voronoi es un vértice en Delaunay.
- Dos nodos de celdas vecinas en Voronoi se conectan formando una arista en Delaunay; eso define los triángulos.
    - Una arista de Voronoi está contenida en la recta equidistante a dos vértices de una arista Delaunay.
- Si por cada triángulo defino una circunferencia, los centros son los vértices de Voronoi.

Teniendo la triangulación (rojo), por cada arista trazo un recta perpendicular que pase por su centro (pts equidistante a ambos vétices de la arista). Estas rectas se van a juntar de a 3 en los vértices del diagrama (verde).



Para elegir la diagonal correcta con la que partir este quad tengo dos justificaciones posibles:

- buscar "maximizar el mínimo ángulo" (el menor del quad es BCD, por eso evito partirlo),
- o que las circunferencias de los triángulos generados no contengan a otro vértice.

Ej6: Proyector

- El rectángulo que forman los Q_i es efectivamente un rectángulo perfecto: es plano porque el enunciado dice que serían los vértices si proyectáramos "sobre un plano", y es un rectángulo (lados paralelos, ángulos de 90) porque el enunciado dice que ese plano es "perfectamente perpendicular" a la dirección en que el cañón proyecta la imagen (no hay deformación por la perspectiva).
- Bajo estas condiciones podemos encontrar la intersección de un un rayo de luz que salga del proyector (de *P*) hacia un punto del objeto (*F*) contra ese plano. Si el objeto está de frente al proyector (misma dirección para un vector de *P* a *F* que para uno de *P* al centro de la proyección) y la intersección cae "dentro" del rectángulo, se ilumina con el color que envíe el proyector; si no, no.
- Para saber si cae dentro (el pto intersección calculado en el paso anterior) calculamos las coordenadas de texturas y vemos si están entre 0 y 1. Para calcular las coord. de textura podemos pensar en partirlo (al rectángulo) en dos triángulos y usar interpolación afín, o usar bilineal (es un caso fácil, lados paraleos). Por ser un rectángulo perfecto, ambas soluciones dan el mismo mapeo. Más aún, si uso el triangulo Q_0, Q_1, Q_2, el peso (coord. baricéntricas) de Q_0 es la coord. de textura *s* y el peso de Q_2 es *t*, aún cuando la intersección esté en el otro triángulo (el peso que no va a estar entre 0 y 1 va a ser el de Q_1... pueden dibujar las isolineas para convencerse de esto).
- Resumiendo:

1. ver si está frente al proyector [5]:
   - $C = \sum_i^4 \frac{1}{4} Q_i$
   - ¿$(C - P) \cdot (F - P)$>0?
2. encontrar la intersección:
   - $n = (Q_0 - Q_1) \times (Q_2 - Q_1)$
   - $a_P = (P - Q1) \cdot n$
   - $a_F = (F - Q1) \cdot n$
   - $\alpha = \frac{a_p}{a_p - a_f}$
   - $I = (1 - \alpha)P + \alpha F$
3. Calcular *s* y *t*
   - $s = \frac{((Q_1 - Q_2) \times (I - Q_2)) \cdot n}{n^2}$
   - $t = \frac{((Q_0 - Q_1) \times (I - Q_1)) \cdot n}{n^2}$
   - ¿$s \in [0, 1] \wedge t \in [0, 1]$?

---

1. menciono esta segundo por completitud, pero en el parcial con la 1ra ya les daba todo el puntaje]. ↩

2. menciono esto por completitud, pero en el parcial con las 2 anteriores ya les daba todo el puntaje, ya que esto no es algo que solemos programar, sino que ya hace la GPU. ↩

3. hay un ambigüedad en $t = 0.5$, pero da lo mismo con cual la evalúe. Si molesta, podría definirse el primer tramo como $t \in [0; 0.5)$ o el 2do como $t \in (0.5; 1]$. ↩

4. correctamente en un caso general debería ser "hiperbólica" la interpolación, pero este enunciado aclara que el objeto se ve de frente para poder hacerlo simplemente linea. ↩

5. se podría haber hecho también con el α del paso 2, preguntando si es positivo ↩