Tarea 4 – Comparando Line Sweeping y Barycentric

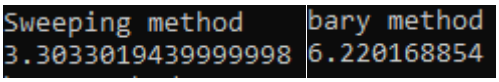
Gráficas por Computadora

Gabriel Brolo Tobar, 15105

---

1. ¿Sin ninguna modificación, cuál algoritmo fue capaz de renderizar su modelo en menos tiempo?

Fue Line Sweeping.



2. ¿En porcentaje, cuanto más rápido fue un algoritmo sobre el otro?

Fue un 53.05% más rápido.

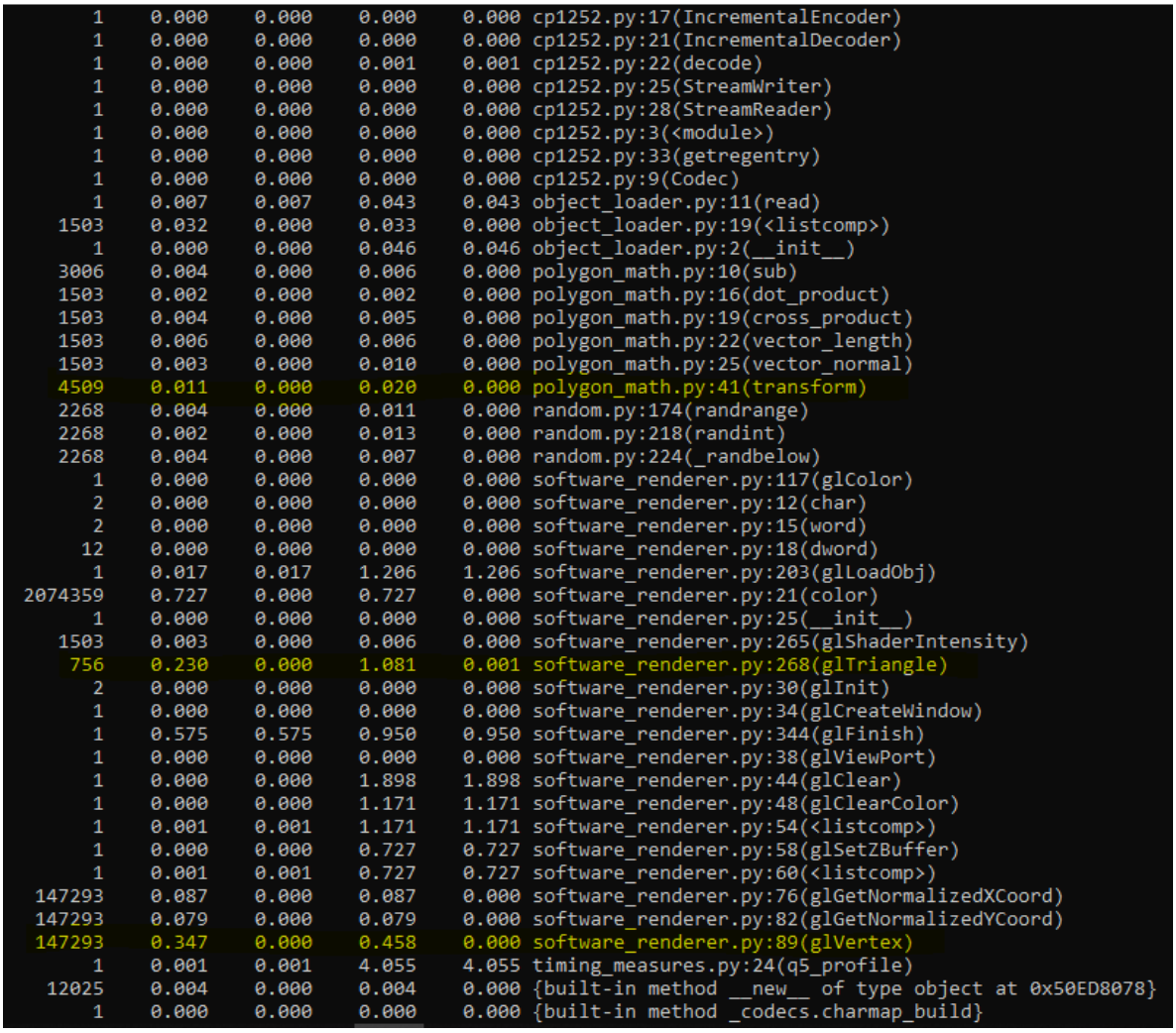
3. ¿En qué manera es superior el algoritmo de line sweeping al barycentric?

Que es más rápido dibujar líneas dentro de los triángulos realizados, al no tener que computar coordenadas baricéntricas para cada pixel.

4. ¿En qué manera es superior el algoritmo de barycentric al de line sweeping?

Es más fácil encontrar un *bounding box*; no hay mucho problema en verificar si un punto determinado pertenece a un triángulo 2D (o cualquier polígono convexo), solo se necesita computar las coordenadas baricéntricas para cada pixel y si tiene al menos una componente negativa, el pixel está fuera del triángulo, lo que garantiza que todo pixel dentro del bounding box será pintado, a diferencia del otro algoritmo en donde podríamos tener la eventualidad que algún pixel no sea pintado.

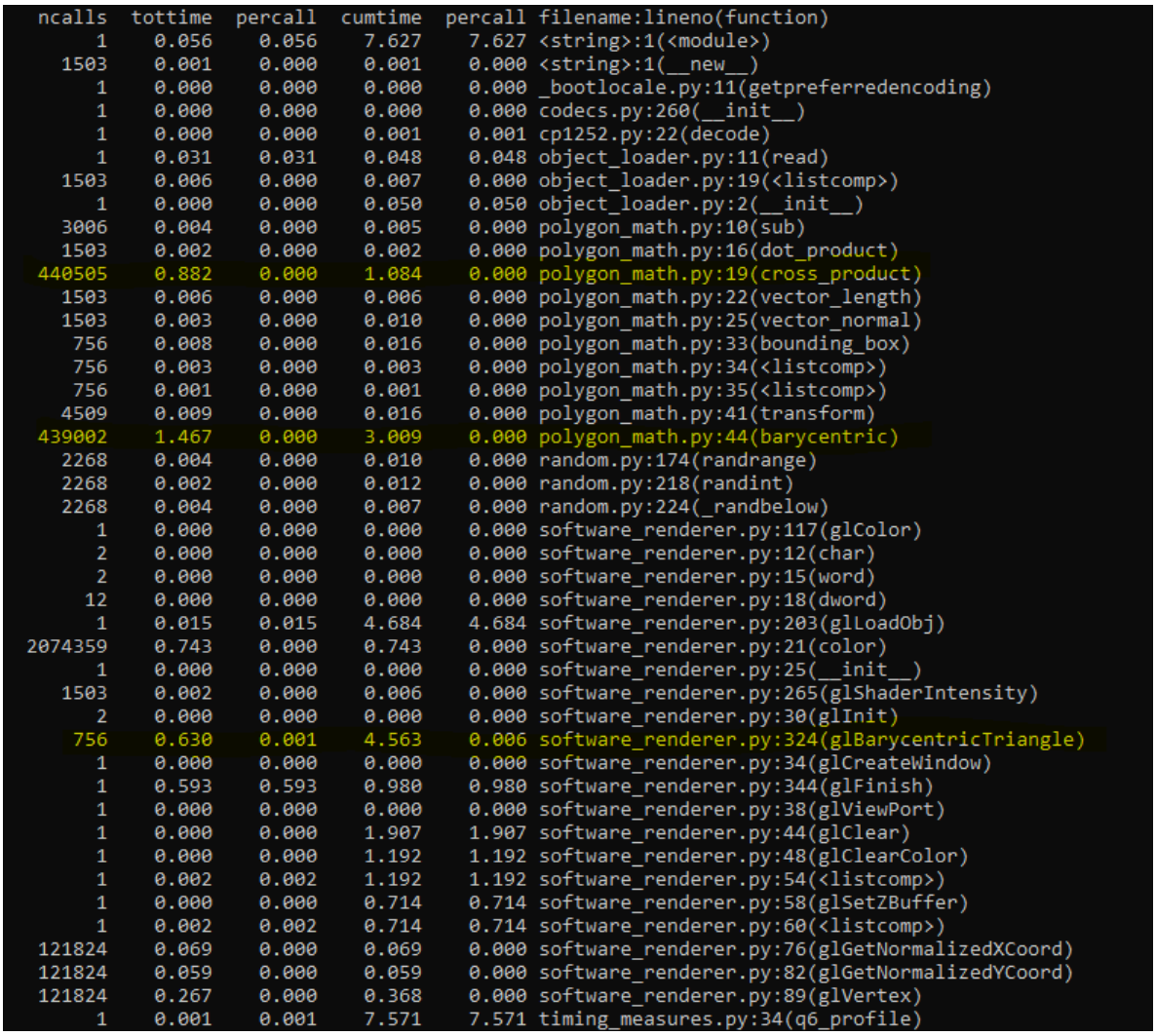
5. ¿Cuál es la operación más costosa en el algoritmo de line sweeping?



```
      1    0.000    0.000    0.000    0.000 cp1252.py:17(IncrementalEncoder)
      1    0.000    0.000    0.000    0.000 cp1252.py:21(IncrementalDecoder)
      1    0.000    0.000    0.001    0.001 cp1252.py:22(decode)
      1    0.000    0.000    0.000    0.000 cp1252.py:25(StreamWriter)
      1    0.000    0.000    0.000    0.000 cp1252.py:28(StreamReader)
      1    0.000    0.000    0.000    0.000 cp1252.py:3(<module>)
      1    0.000    0.000    0.000    0.000 cp1252.py:33(getregentry)
      1    0.000    0.000    0.000    0.000 cp1252.py:9(Codec)
      1    0.007    0.007    0.043    0.043 object_loader.py:11(read)
   1503    0.032    0.000    0.033    0.000 object_loader.py:19(<listcomp>)
      1    0.000    0.000    0.046    0.046 object_loader.py:2(__init__)
   3006    0.004    0.000    0.006    0.000 polygon_math.py:10(sub)
   1503    0.002    0.000    0.002    0.000 polygon_math.py:16(dot_product)
   1503    0.004    0.000    0.005    0.000 polygon_math.py:19(cross_product)
   1503    0.006    0.000    0.006    0.000 polygon_math.py:22(vector_length)
   1503    0.003    0.000    0.010    0.000 polygon_math.py:25(vector_normal)
   4509    0.011    0.000    0.020    0.000 polygon_math.py:41(transform)
   2268    0.004    0.000    0.011    0.000 random.py:174(randrange)
   2268    0.002    0.000    0.013    0.000 random.py:218(randint)
   2268    0.004    0.000    0.007    0.000 random.py:224(_randbelow)
      1    0.000    0.000    0.000    0.000 software_renderer.py:117(glColor)
      2    0.000    0.000    0.000    0.000 software_renderer.py:12(char)
      2    0.000    0.000    0.000    0.000 software_renderer.py:15(word)
     12    0.000    0.000    0.000    0.000 software_renderer.py:18(dword)
      1    0.017    0.017    1.206    1.206 software_renderer.py:203(glLoadObj)
2074359    0.727    0.000    0.727    0.000 software_renderer.py:21(color)
      1    0.000    0.000    0.000    0.000 software_renderer.py:25(__init__)
   1503    0.003    0.000    0.006    0.000 software_renderer.py:265(glShaderIntensity)
    756    0.230    0.000    1.081    0.001 software_renderer.py:268(glTriangle)
      2    0.000    0.000    0.000    0.000 software_renderer.py:30(glInit)
      1    0.000    0.000    0.000    0.000 software_renderer.py:34(glCreateWindow)
      1    0.575    0.575    0.950    0.950 software_renderer.py:344(glFinish)
      1    0.000    0.000    0.000    0.000 software_renderer.py:38(glViewPort)
      1    0.000    0.000    1.898    1.898 software_renderer.py:44(glClear)
      1    0.000    0.000    1.171    1.171 software_renderer.py:48(glClearColor)
      1    0.001    0.001    1.171    1.171 software_renderer.py:54(<listcomp>)
      1    0.000    0.000    0.727    0.727 software_renderer.py:58(glSetZBuffer)
      1    0.001    0.001    0.727    0.727 software_renderer.py:60(<listcomp>)
 147293    0.087    0.000    0.087    0.000 software_renderer.py:76(glGetNormalizedXCoord)
 147293    0.079    0.000    0.079    0.000 software_renderer.py:82(glGetNormalizedYCoord)
 147293    0.347    0.000    0.458    0.000 software_renderer.py:89(glVertex)
      1    0.001    0.001    4.055    4.055 timing_measures.py:24(q5_profile)
  12025    0.004    0.000    0.004    0.000 {built-in method __new__ of type object at 0x50ED8078}
      1    0.000    0.000    0.000    0.000 {built-in method _codecs.charmap_build}
```

En sí lo que más tiempo le lleva es dibujar los puntos con glVertex (0.347s, 147293 llamadas), sin embargo, el mero algoritmo está en glTriangle (0.230s, 756 llamadas), en dónde se encuentran los cálculos para realizar los triángulos, pero son los mismos que usa el barycentric. Aún así, esta es la parte más costosa del algoritmo por que las otras funciones son funciones generales.

6. ¿Cuál es la operación más costosa en el algoritmo barycentric?



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.056	0.056	7.627	7.627	<string>:1(<module>)
1503	0.001	0.000	0.001	0.000	<string>:1(__new__)
1	0.000	0.000	0.000	0.000	_bootlocale.py:11(getpreferredencoding)
1	0.000	0.000	0.000	0.000	codecs.py:260(__init__)
1	0.000	0.000	0.001	0.001	cp1252.py:22(decode)
1	0.031	0.031	0.048	0.048	object_loader.py:11(read)
1503	0.006	0.000	0.007	0.000	object_loader.py:19(<listcomp>)
1	0.000	0.000	0.050	0.050	object_loader.py:2(__init__)
3006	0.004	0.000	0.005	0.000	polygon_math.py:10(sub)
1503	0.002	0.000	0.002	0.000	polygon_math.py:16(dot_product)
440505	0.882	0.000	1.084	0.000	polygon_math.py:19(cross_product)
1503	0.006	0.000	0.006	0.000	polygon_math.py:22(vector_length)
1503	0.003	0.000	0.010	0.000	polygon_math.py:25(vector_normal)
756	0.008	0.000	0.016	0.000	polygon_math.py:33(bounding_box)
756	0.003	0.000	0.003	0.000	polygon_math.py:34(<listcomp>)
756	0.001	0.000	0.001	0.000	polygon_math.py:35(<listcomp>)
4509	0.009	0.000	0.016	0.000	polygon_math.py:41(transform)
439002	1.467	0.000	3.009	0.000	polygon_math.py:44(barycentric)
2268	0.004	0.000	0.010	0.000	random.py:174(randrange)
2268	0.002	0.000	0.012	0.000	random.py:218(randint)
2268	0.004	0.000	0.007	0.000	random.py:224(_randbelow)
1	0.000	0.000	0.000	0.000	software_renderer.py:117(glColor)
2	0.000	0.000	0.000	0.000	software_renderer.py:12(char)
2	0.000	0.000	0.000	0.000	software_renderer.py:15(word)
12	0.000	0.000	0.000	0.000	software_renderer.py:18(dword)
1	0.015	0.015	4.684	4.684	software_renderer.py:203(glLoadObj)
2074359	0.743	0.000	0.743	0.000	software_renderer.py:21(color)
1	0.000	0.000	0.000	0.000	software_renderer.py:25(__init__)
1503	0.002	0.000	0.006	0.000	software_renderer.py:265(glShaderIntensity)
2	0.000	0.000	0.000	0.000	software_renderer.py:30(glInit)
756	0.630	0.001	4.563	0.006	software_renderer.py:324(glBarycentricTriangle)
1	0.000	0.000	0.000	0.000	software_renderer.py:34(glCreateWindow)
1	0.593	0.593	0.980	0.980	software_renderer.py:344(glFinish)
1	0.000	0.000	0.000	0.000	software_renderer.py:38(glViewPort)
1	0.000	0.000	1.907	1.907	software_renderer.py:44(glClear)
1	0.000	0.000	1.192	1.192	software_renderer.py:48(glClearColor)
1	0.002	0.002	1.192	1.192	software_renderer.py:54(<listcomp>)
1	0.000	0.000	0.714	0.714	software_renderer.py:58(glSetZBuffer)
1	0.002	0.002	0.714	0.714	software_renderer.py:60(<listcomp>)
121824	0.069	0.000	0.069	0.000	software_renderer.py:76(glGetNormalizedXCoord)
121824	0.059	0.000	0.059	0.000	software_renderer.py:82(glGetNormalizedYCoord)
121824	0.267	0.000	0.368	0.000	software_renderer.py:89(glVertex)
1	0.001	0.001	7.571	7.571	timing_measures.py:34(q6_profile)

La función para calcular las baricéntricas es el que más tiempo toma (1.467s), asimismo tiene bastantes llamadas (439002) debido a que se calculan para cada pixel. De hecho, se puede observar también que la función para calcular el producto cruz toma un tiempo 0.882s y también tiene un gran número de llamadas (440505).

7. Las computadoras modernas tienen muchos más recursos que las computadoras antiguas, lo que les permite ejecutar varios threads a la vez. ¿Cuál de estos algoritmos es más amigable a ejecutarse en múltiples threads y por qué? (incluyan cómo implementarían múltiples threads en su respuesta)

Considero que el barycentric tendría más posibilidad de implementarse con threads; la operación más tardada es calcular las baricéntricas para cada pixel, luego si hay una componente negativa no se dibuja, es todo. Podría crearse un thread por batch de pixeles y ejecutar los batches al mismo tiempo,

entonces cada batch se ejecuta por separado y se estarían pintando simultáneamente varios pedazos del render.

8. En la clase vimos cómo interpolar los valores de z utilizando las coordenadas baricéntricas. ¿Cómo interpolarían valores en el algoritmo de line sweeping?

Tal vez podría buscarse el punto medio de cada triángulo y luego calcular la distancia euclideana que existe entre cada pixel del borde del triángulo con respecto a punto medio y luego para cada valor de z comparar si ese valor es mucho más grande y si sí, se haría un swap.