Informe sobre la Implementación del Triángulo de Sierpinski

Alonso Valdivia Quispe
E.P. Ciencia de la Computación
Univesidad Católica San Pablo
Arequipa, Perú
Email: **alonso.valdivia.quispe@ucsp.edu.pe**

***Resumen*—En esta oportunidad conoceremos los detalles de sobre la implementación del Triángulo de Sierpinski en OpenGL 3.3. Cubriremos todos los jugosos detalles sobre como ser realizo su construcción y cual es el elemento personal que posee la implementación. La estructura del documento es el siguiente; primero daremos una introducción, segundo repasaremos las clases que se creo para la implementación, tercero analizaremos los experimentos fallidos, cuarto detallaremos la técnica usada para renderizado(dibujo y coloreo) y finalmente veremos cual es el elemento personal que se implemento.**

I. Introducción y Motivación

Permitanme hablar de en primera persona, porque siento que conecto con el lector de esta forma. Inicié este trabajo con la idea de hacer interesante, y creo que en gran medida lo logré. Veremos mi recorrido por todo OpenGL y sus estresantes, curiosas, graciosas, toscas y divertidas funciones que usé para completar este trabajo. Mi idea central fue poder acceder a cada triángulo de forma óptima y elegante, de tal forma que mi imagen no sea solo una figura estática que se ve bien pero nada más, yo quería que fuese útil y que tenga potencia de evolucionar a algo más complejo.

II. Clases: ¿El inicio del fin?

II-A. Shader Class

Mi fiel compañera fue la Shader Class que encapsula todas las instrucciones para generar un ShaderProgram, sin embargo, es la misma que trabajos anteriores. Una gran duda que aún tengo es el hecho de no poder modificar el Alpha, lo intenté pero no resultó. Averiguando empecé a ver temas como matrices y programación un poco más avanzada en GLSL, todo esto forma parte de un tópico conocido como **Blending** que decidí no profundizar más debido a su complejidad. No logré implementar la transparencia.

II-B. Color Class y Vao Class

Mi idea inicial era que cada triángulo tuviera su propio VAO y su propio ShaderProgram, con ese fin las Clases Color y Vao fueron implmentadas. Sin embargo, tuve serios problemas con el ámbito de OpenGL y adquirí nuevos conocimientos sobre GLSL que me hicieron cambiar de pensamiento. Estas clases quedan como un recuerdo de lo que intenté hacer. La Clase Color guarda un índice a un supuesto ShaderProgram y la Calse Vao contiene todas las intrucciones para crear un VAO.

II-C. Triangle Class

La Triangle Class es la que más trabaja, pues es el corazón de todo el código. La clase tiene los siguientes parámetros:

- coor : Es un vector de puntos que contiene los vértices que forman dicho triángulo.
- childrens : Es un vector de punteros a Triangle Class, este vector guardará los hijos de este Triángulo.

Algo interesante es la función **getVao()** que retorna un vector de **float** listo para ser usado.

II-D. Sierpinski Class

Inspirado en la conocida estructura RTree, realicé algo muy similiar con esta clase. La clase tiene los siguientes campos:

- I : Puntero a una Triangle Class que será el inicio de toda nuestra estructura.
- VAO : Vector que contiene todos los vértices que deben ser renderizados en un orden específico.

Sobre las funciones hay dos que son super importantes:

- AutoDiv(Triangle*,int) : Esta función da inicio a todas las subdivisiones. Reclama un punto de partida y un nivel requerido al cual llegar. Cálcula cuantos triángulos debe construir. Todo este recorrido se realizará con un BFS, lo cual permitirá recorrer y guardar el árbol por niveles. Al final de todo el recorrido la variable **VAO** contendrá todos los vértices para el renderizado.
- Search(Triangle*,pto,vector¡float¿) : Esta función será la que busque un punto en toda la estructura de forma óptima. Lo clásico sería realizar una búsqueda linear sobre todos los triángulos posibles. Sin embargo gracias a mi estructura, el número de triángulos a verificar se reduce a la tercera parte a cada paso. Va buscando a través de todos los hijos del actual Triángulo y evalúa si puede irse hacia algún hijo, si puede irse entonces lo hace sino verifica si pertenece al área del Triángulo y de ser así entonces devuelve ese Triángulo. Finalmente si no está ni en sus hijos ni en si mismo, entonces devuelve **False**. El último parámetro es un vector de float que guardará los vértices del triángulo que contiene el punto dado.

III. Experimentos Fallidos: Es el fin

Cuando empecé a diseñar las clases e intentar compilarlas, tuve serios problemas al llegar a trabajar con el VAO y

los Shader, aún hoy me pregunto la razón de que aveces simplemente cierro mi computador o borro y vuelvo a escribir algo hace que compile de una u otra manera. Tengo serios presentimientos que tiene que ver con la memoria que se usa, pues leyendo un poco encontré que todo en OpenGL tiene un límite dictaminado por tu máquina.

III-A. VAO Class

La idea era que cada triángulo tenga su propio VAO y VBO para el renderizado, sin embargo, no podía lograr la compilación por algún problema con el ámbito de la memoria. Pero me di cuenta que hacerlo de esa forma era muy ineficiente pues solo necesitaba un VAO y VBO que guarde todo, el resto lo podía hacer con índices, cuando empecé a pensar así entendí que el VAO debería renderizarse solo una vez de ser posible. Sin embargo encontré que si se puede renderizar el VAO en una clase o una función si se hace forma correcta, algo que hago en mi proyecto para renderizar el Triángulo que cubre el área donde se hace click.

III-B. Tiempo, mi gran enemigo

En experimentos fallidos y averiguación quemé dos días de trabajo, hasta que finalmente logré cambiar mi pensamiento y mudarme de un paradigma enteramente dinámico a un paradigma que tiene dinámico solo lo necesario y el resto es estático. Todo esto me consumió mucho tiempo pero logré entender que estaba mal en mi programa y siento que más adelante ya no tendré estos problemas, así que en teoría todo este tiempo realmente lo invertí pues más adelante ya no tendré problemas, incluso aprendí como interactuar con las posiciones del mouse.

IV. Renderizado: Una luz de esperanza

Como lo dije, para renderizar la figura lo primero que hago es inicializar la estructura del Triángulo e invocar a la función **AutoDiv()** para que toda la estructura se construya. Usamos esto para construir el VAO que renderizará todo el triángulo. Para colorear la estructura voy a usar un vector que guardará un shaderProgram para cada nivel de la estructura.
Finalmente en la parte de renderización, ya que todo esta ordenado por niveles puedo renderizar bajo ese mismo principio. La fórmula usada es sencilla, el número de triángulos a rendericar es 3^{level} y ya que tenemos 3 vértices por triángulo entonces la fórmula final es $3^{level} * 3$. Finalmente a cada paso avanzamos el índice j hasta donde inicia el siguiente nivel. Esto lo vemos en la figura 1. Como vemos en la renderización

```
for (int i = 0; i <= level; ++i) {
    glUseProgram(IDSH_por_level[i]);
    glDrawArrays(GL_TRIANGLES, j, pow(3,i)*3);

    j = j+(pow(3, i) * 3);
}
```

Figura 1. Forma de renderizado por niveles

de cada nivel usamos un distinto ShaderProgram.
Para que el mouse pueda rellenar un triángulo seleccionado, usaremos un shader propio para este propósito. Además quiero que pueda iluminarse solo el marco del triángulo seleccionado o todo el triángulo. Para dicho propósito usamos un booleano que cambie con la tecla S. Esto lo podemos ver en la figura 2. El renderizado del triángulo apuntado por el mouse

```
if(wire_Pointer)glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
else glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glBindVertexArray(Pointer);
glUseProgram(IDSH_Pointer);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Figura 2. Renderizado de triángulo apuntado

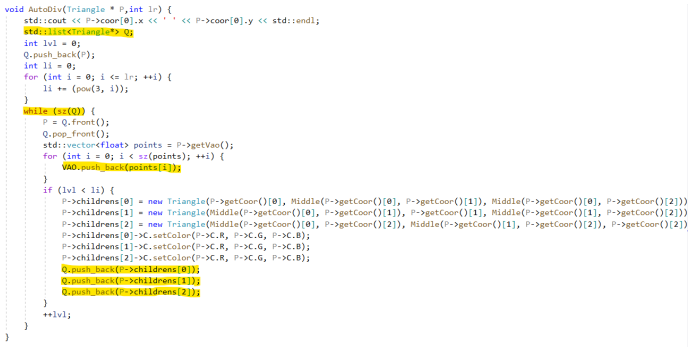
se dará funciones parecidas al **glfwSetKeyCallback(window, processKey)** usados en trabajos previos. Las funciones que use son:

- processKey(window,key,scancode,action,mods) : La función más allá de lo ya conocido, uso esta función para poder cambiar tres variables clave:
  - level : Cambiará con las teclas UP y DOWN para controlar hasta que nivel se renderizará.
  - wire : Controlará si la renderización de los triángulos se realizará solamente los marcos o el triángulo relleno.
  - wire_Pointer : Controlará si la renderización del Triángulo seleccionado con el mouse se realizará solamente los marcos o el triángulo relleno.
- mouse_button_callback(window,button,action,mods) : Es muy parecida a la función anterior solo que esta vez captamos que botón del mouse es presionado. Con esto controlamos la variable **init** que indica si queremos resaltar el triángulo seleccionado o no. Además de que capturamos las posiciones del mouse con la función **glfwGetCursorPos(window,xpos,ypos)** para finalmente convertirlas a las coordenas de nuestro sistema y darle el punto convertido a nuestra estructura.

En este parte todo pareció andar sobre ruedas, y avancé a pasos agigantados pues en tan solo 10 minutos logré hacer todo lo que mis demás compañeros hicieron, excepto algunos cuya mecánica se basaba en el hecho de que el orden de los triángulos estaba dado por un DFS, ese detalle te complica mucho trabajar tu triángulo por niveles, es por eso que use un enfoque basado en BFS para renderizar mi triángulo. Este enfoque basado en BFS puede verse en la figura 3. Este avance rápido me permitió terminar a tiempo con un relativamente buen trabajo.

V. Aporte personal

La última parte que veremos es el aporte final que tuve. Mientras algunos hicieron cosas super interesantes como controlar el renderizado del triángulo por niveles, otros hicieron variar los colores con los que se pintaba el triángulo. Finalmente vi varios que renderizaban todo solo con marcos. Mi triángulo puede:



```
void AutoDiv(Triangle * P,int lr) {
    std::cout << P->coor[0].x << ' ' << P->coor[0].y << std::endl;
    std::list<Triangle*> Q;
    int lvl = 0;
    Q.push_back(P);
    int li = 0;
    for (int i = 0; i <= lr; ++i) {
        li += (pow(3, i));
    }
    while (sz(Q)) {
        P = Q.front();
        Q.pop_front();
        std::vector<float> points = P->getVao();
        for (int i = 0; i < sz(points); ++i) {
            VAO.push_back(points[i]);
        }
        if (lvl < li) {
            P->childrens[0] = new Triangle(P->getCoor()[0], Middle(P->getCoor()[0], P->getCoor()[1]), Middle(P->getCoor()[0], P->getCoor()[2]))
            P->childrens[1] = new Triangle(Middle(P->getCoor()[0], P->getCoor()[1]), P->getCoor()[1], Middle(P->getCoor()[1], P->getCoor()[2]))
            P->childrens[2] = new Triangle(Middle(P->getCoor()[0], P->getCoor()[2]), Middle(P->getCoor()[1], P->getCoor()[2]), P->getCoor()[2])
            P->childrens[0]->C.setColor(P->C.R, P->C.G, P->C.B);
            P->childrens[1]->C.setColor(P->C.R, P->C.G, P->C.B);
            P->childrens[2]->C.setColor(P->C.R, P->C.G, P->C.B);
            Q.push_back(P->childrens[0]);
            Q.push_back(P->childrens[1]);
            Q.push_back(P->childrens[2]);
        }
        ++lvl;
    }
}
```

Figura 3. Construcción del triángulo usando BFS

- Controlar el renderizado por niveles, haciendo uso de las teclas UP y DOWN.
- Controlar el modo de renderizado, puede ser solo el marco o el triángulo entero. Esto se controla con la tecla W(presiona para cambiar entre los modos de renderizado),
- Seleccionar un triángulo y pintarlo de otro color, esto se realiza haciendo click derecho sobre un triángulo que deseemos renderizar. Si queremos desactivar esta función solo debemos presionar el click izquierdo del mosuse.
- Controlar el modo con que se pinta el triángulo seleccionado, esto se logra con la tecla S.

Y eso es todo, no vi que ningún otro trabajo pueda seleccionar y pintar un triángulo de la misma forma que yo la hice. Es verdad que se puede hacer esto realizando una búsqueda lineal pero yo lo hago en logaritmo base 3 del número de triángulos. Ese fue mi aporte personal, se que no es tan vistoso pero me parece que cumple todos los requisitos bastante bien.

VI. Conclusión

Apesar de que alcancé todos mis objetivos con este trabajo, no pude aplicar todo el conocimiento que adquirí debido a que en sí el ejercicio no es muy complejo, basta con crear todos los triángulos y listo, en base a eso podemos jugar con los colores que me parece que fue lo que más se hizo. Aposté por algo más extremo que pueda ayudar a alguien que quiera hacer uso del Triángulo en sí. Para concluir pueden ver el **repositorio** donde esta todo el proyecto listo para construirse(con video incluído), y verán cosas mucho más interesantes de parte mía en proyectos futuros pues tengo ahora una gran capacidad de renderizar cosas relativamente complejas con mucho más orden, Gracias.