Informática Gráfica

Práctica 2: Generación e iluminación de objetos por revolución

UNIVERSIDAD DE GRANADA

Grado en Ingeniería Informática

Jesús Pereira Sánchez

jesuspereira@correo.ugr.es

Índice

1. Introducción

En esta práctica 2, se implementa un algoritmo de revolución para la generación de mallas a partir de un perfil 2D. Se presenta un algoritmo inicial, el cual tiene una iteración para la mejora de la iluminación, basado en conceptos de ejercicios anteriores de esta misma práctica.

2. Primera aproximación: LatheGeometrySmooth

Ésta primera aproximación del algoritmo de revolución genera una malla con los vértices compartidos. Esto quiere decir que a la hora de iluminar el objeto, se realizará de forma suave, y al final el objeto no tendrá los bordes marcados.

El algoritmo es el siguiente:

Listing 1: LatheGeometrySmooth

```
func LatheGeometrySmooth ( profile2D : PackedVector2Array,
                                                steps : int,
                                                vertices : PackedVector3Array,
                                                indexes : PackedInt32Array) :

        assert(steps > 0,  "Number of steps must be positive")
        assert(positiveX(profile2D),  "All x coordinates from the profile must be
            positive")
        assert(vertices.is_empty(), "The vertices vector must be empty")
        assert(indexes.is_empty(), "The indexes vector must be empty")

        for i in range(profile2D.size()) :
                var point = profile2D[i]
                for j in range(steps):
                        var angle = (j / float(steps)) * 2 * PI
                        var x = point[0] * cos(angle)
                        var y = point[1]
                        var z = point[0] * sin(angle)
                        vertices.append( Vector3(x, y, z) )

        for i in range(profile2D.size() - 1) :
                for j in range(steps) :
                        var next_j = (j + 1) % steps
                        var current = i * steps + j
                        var next = (i + 1) * steps + j
                        var current_next = i * steps + next_j
                        var next_next = (i + 1) * steps + next_j


                        indexes.append(next_next)
                        indexes.append(next)
                        indexes.append(current)

                        indexes.append(current_next)
                        indexes.append(next_next)
                        indexes.append(current)
```

Empezamos con la comprobación de que los parámetros estén bien. Para ello, el número de pasos (rotaciones) debe de ser positivo. Además, todos los puntos deben de tener coordenadas en x que sean positivas. Y por último,

que las listas de vértices y de índices estén vacías al llamar al algoritmo.

Comenzamos calculando los vértices, usando los puntos para rotarlos. Para ello, iteramos por todos los puntos del perfil 2D que es parámetro de entrada. Para cada punto:

1. Calculamos el ángulo de rotación, el cual dependerá del step en el que se esté.
2. Calculamos cada componente, x, z, como producto de la rotación. La y es siempre la misma, puesto que rotamos en ese eje.
3. Añadimos el punto a la lista de vértices.

Una vez tenemos todos los vértices, comenzamos con la generación de las caras. En esta aproximación lo haremos con vértices compartidos en las caras.

Iteramos por todos los puntos del perfil, menos el último, ya que en cada iteración del bucle vamos a hacer los triángulos del punto en el que estamos usando el siguiente. Dentro de cada punto del perfil, iteramos en cada uno de los steps.

Hay que recordar que en *vertices* tenemos una lista de vértices, la cual puede ser tratada como una matriz. Vamos a hacer la transformación de lista a matriz y viceversa para calcular los 4 puntos de un cuadrado, y definir los triángulos.

Este cuadrado está definido por las variables *current*, *next*, *current next* y *next next*. Es por ello que nuestro cuadrado es el siguiente:

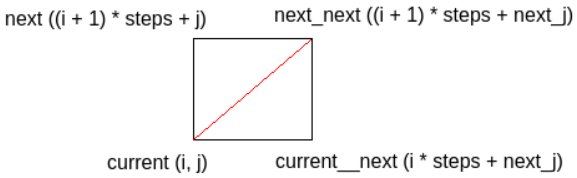

Figura 1: Cuadrado de ejemplo

El sentido en el que construimos los triángulos es contrario a las agujas del reloj. En el algoritmo los construimos del siguiente orden para cada triángulo:

1. *next next - next - current*
2. *current next - next next - current*

Se van añadiendo los puntos según vamos iterando por la matriz hecha en una lista, y obteniendo los puntos necesarios. Es importante remarcar la importancia del cálculo de módulo para poder envolver la figura totalmente a la hora de calcular el siguiente índice.

Ahora, vamos a probar el algoritmo con un perfil. El que usaremos de ejemplo es el de un peón. Obtenemos lo siguiente:

Figura 2: Peón suave

Como vemos en la imagen, el algoritmo ha generado un peón con iluminación suave. Aquí ocurre lo mismo que en el ejercicio anterior con el cubo, estamos usando en un vértice la aproximación de sus colindantes. Es por ello que tenemos que aplicar la misma solución que anteriormente para poder tener los bordes marcados.

3. Aproximación final: LatheGeometryHard

Esta solución mejora lo comentado anteriormente, ahora cada triángulo tiene su normal, y los vértices no tienen la suma de sus colindantes. Este es el código, mucho más distinto a la solución anterior:

Listing 2: LatheGeometryHard

```
func LatheGeometryHard(profile2D: PackedVector2Array,
                                                steps: int,
                                                vertices: PackedVector3Array,
                                                indexes: PackedInt32Array):

        assert(steps > 0,  "Number of steps must be positive")
        assert(positiveX(profile2D),  "All x coordinates from the profile must be
            positive")
        assert(vertices.is_empty(), "The vertices vector must be empty")
        assert(indexes.is_empty(), "The indexes vector must be empty")

        for i in range(profile2D.size() - 1):
                for j in range(steps):
                        var next_j = (j + 1) % steps
                        var p0 = profile2D[i]
                        var p1 = profile2D[i + 1]

                        var angle0 = (j / float(steps)) * 2 * PI
                        var angle1 = (next_j / float(steps)) * 2 * PI

                        var v0 = Vector3(p0.x * cos(angle0), p0.y, p0.x * sin(angle0
                            ))
                        var v1 = Vector3(p1.x * cos(angle0), p1.y, p1.x * sin(angle0
                            ))
                        var v2 = Vector3(p1.x * cos(angle1), p1.y, p1.x * sin(angle1
                            ))
                        var v3 = Vector3(p0.x * cos(angle1), p0.y, p0.x * sin(angle1
                            ))

                        var base = vertices.size()

                        # First triangle
                        vertices.append(v0)
                        vertices.append(v1)
                        vertices.append(v2)
                        indexes.append(base + 2)
                        indexes.append(base + 1)
                        indexes.append(base)

                        # Second triangle
                        vertices.append(v0)
                        vertices.append(v2)
                        vertices.append(v3)
                        indexes.append(base + 5)
                        indexes.append(base + 4)
                        indexes.append(base + 3)
```

Ahora, solo tenemos un bucle anidado, mientras que antes teníamos dos (uno para la generación de vértices y otro para la generación de los índices ordenados).

Al igual que antes, comprobamos las condiciones de entrada, las mismas obviamente.

Ya en los bucles, iteramos por cada punto del perfil (menos la última, ya que vamos a calcular en cada iteración siempre la siguiente). Y para cada punto del perfil, hacemos su rotación. Es por ello que el índice ***i*** hace referencia a los puntos del perfil, y que el índice ***j*** hace referencia al *step*, o ángulo de rotación en el que estemos.

Para cada ángulo de rotación, calculamos cual es el siguiente índice de rotación (con ayuda del módulo para envolver la figura).

Después, para el ***p0*** y ***p1*** calculamos su angulo de rotación respectivamente, como podemos ver en ***angle0*** y ***angle1***.

Ahora, calculamos los 4 vértices rotados. Los vértices ***v0*** y ***v3*** son al final el ***p0*** rotado, y los vértices ***v1*** y ***v2*** son el ***p1*** rotado. Además, los vértices ***v0*** y ***v1*** rotan con un ángulo distinto al de los vértices ***v2*** y ***v3***. Es por ello que nuestro cuadrado es el siguiente:

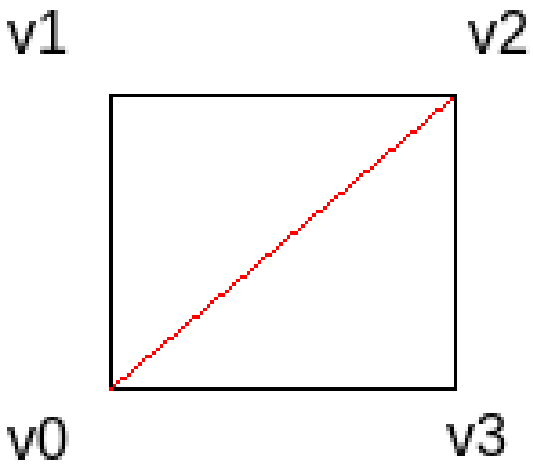

Figura 3: Cuadrado para bordes marcados

Estos 4 vértices forman un cuadrado el cuál tiene contenido los 2 triángulos (para el vector de índices).

Ahora, añadimos los vértices. Se ha separado en primer triángulo del segundo. Los añadimos en el orden que se puede ver en el código, en sentido horario. A la hora de añadirlos al vector de índices, los tenemos que añadir en sentido antihorario para que se iluminen como caras exteriores.

Para añadir los índices al vector de índices se hace de manera antihoraria como se ha comentado. Para hacerlo, y sin saber cuantos índices llevamos en cada iteración, sacamos cuantos elementos hay en el vector de vértices, antes de haber añadido índices en esta iteración. Esta será nuestra referencia (nuestro 0). Para ir metiendo los índices en sentido antihorario, hay que revertirlo en cuanto a la forma que fueron introducidos en el vector. Es por ello que en ***indexes*** vamos añadiendo primero el 3º añadido (referencia + 2), el segundo (referencia + 1) y el primero (referencia).

Hacemos lo mismo para cada triángulo.

Ahora, ejecutamos, y obtendremos nuestra figura bien iluminada de la siguiente forma:

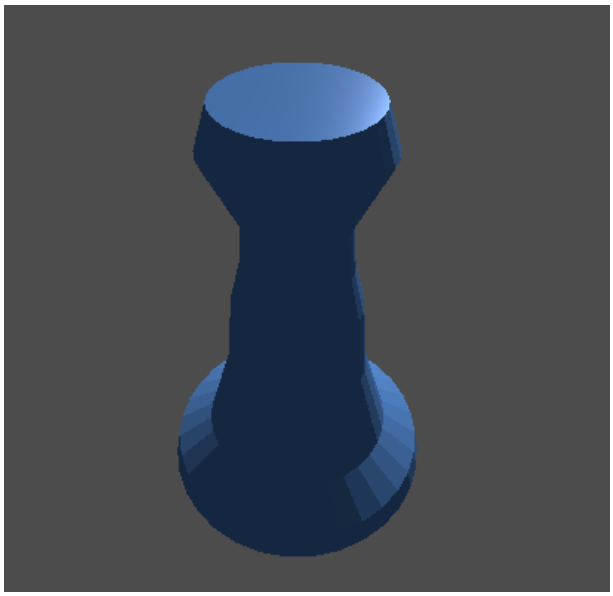

Figura 4: Peón con bordes marcados

4. Comparativa de sombreados en las figuras por revolución

4.1. Peón por revolución

Para el peón por revolución, podemos ver la clara diferencia entre los dos tipos de iluminación. En la iluminación por vértices vemos como las caras iluminadas son completamente coloreadas y difuminadas, mientras que en la iluminación por pixel es más realista, pero gasta más recursos:

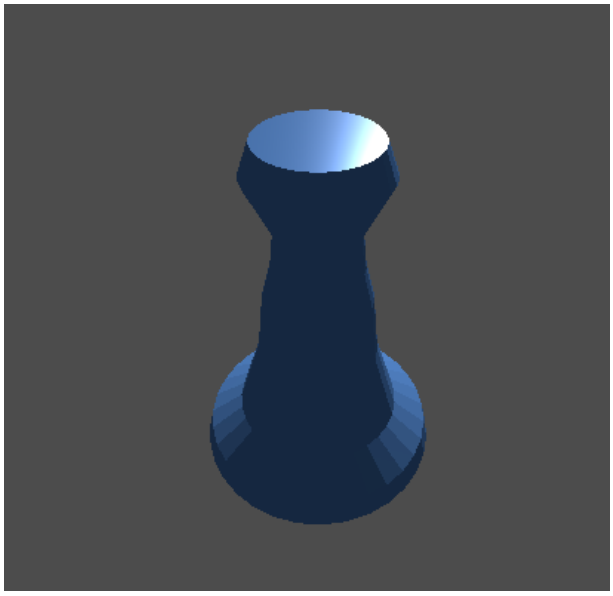

Figura 5: Peón iluminado por píxel

Figura 6: Peón iluminado por vértice

4.2. Esfera por revolución a partir de semicírculo

A continuación, a partir de un perfil 2D de un semicírculo, construimos una esfera usando revolución. La diferencia es un poco más sutil que con el peón. Aún así, podemos ver como el número de superficies iluminadas es mayor en el caso de la esfera iluminada por pixel que en la esfera iluminada por vértice.

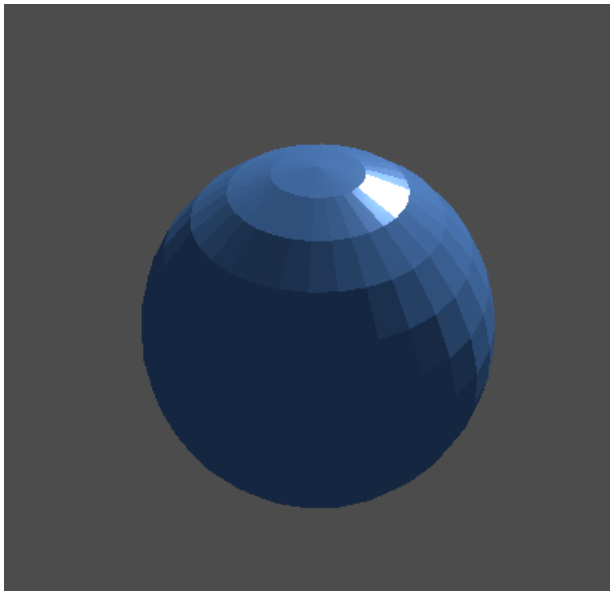

Figura 7: Esfera iluminada por píxel

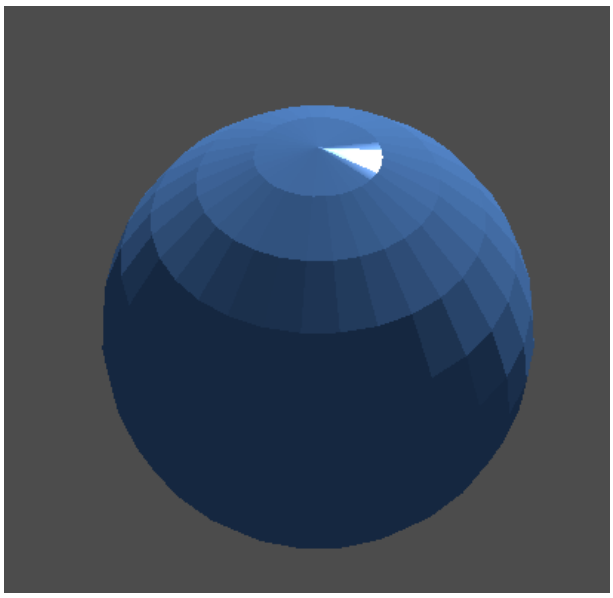

Figura 8: Esfera iluminada por vértice