

Programación Optimizada para Videojuegos

Actividad Patrones de diseño

Juan Siverio Rojas

La Laguna, 6 de julio de 2023

Master Universitario en Desarrollo de Videojuegos

Contenido

[1.1 Objetivo 2](#_Toc139566743)

[*1.2* Proceso 2](#_Toc139566744)

[*1.3* *Método* getAttributeLocations(WebGLprogram p) 4](#_Toc139566745)

[1.4 Modificaciones en clase Game.cs 5](#_Toc139566746)

[1.5 Conclusión 6](#_Toc139566747)

[1.6 Process 6](#_Toc139566748)

[1.7 Conclusion 6](#_Toc139566749)

[Ilustración 1. Clase base del patrón Command. 3](#_Toc139566737)

[Ilustración 2.Clases definidas para Command 3](#_Toc139566738)

[Ilustración 3.Parte de la declaración de clase hija CommandDrawShadow.cs 4](#_Toc139566739)

[Ilustración 4. Variables de clase con los diferentes shaders. 5](#_Toc139566740)

[Ilustración 5. Instanciación e inicialización de los shaders personalizados. 5](#_Toc139566741)

[Ilustración 6. Aspecto métodos GameLoop() y Draw() 6](#_Toc139566742)

## Objetivo

Analizar el código desarrollado en las actividades WebGl y buscar posibles patrones de diseño de videojuegos que puedan aplicarse para mejorar su eficiencia o la capacidad de expansión.

## Proceso

Tras analizar el código de la práctica tercera de WebGl, en la que se crean varios shaders de fragmentos, uno para los colores, otro para las sombras personalizado por cada objeto, me parece interesante la posibilidad de poder aislar ese proceso de manera independiente, pudiendo agregar los shaders necesarios en cualquier momento o modificando su funcionamiento sin necesidad de retocar el código.

El patrón de diseño elegido, que parece que cubre esta situación bastante bien, será el patrón de comportamiento Command.

Con este patrón puedo crear una clase que implemente todo lo referente a un nuevo shaders, y su funcionamiento sería llamando a la clase base, al método Exec() en este caso. Gracias al polimorfismo, se ejecutará el método concreto y definido en cada clase. Además, una ventaja también de este patrón, es el poder hacer uso de toda esta implementación en cualquier momento, ya que tenemos la posibilidad de simplemente pasar la clase como parámetro.

En la siguiente imagen se muestra el código de la clase base utilizada, a la que he llamado en este caso, igual que el patrón de diseño, Command. Se ha declarado como abstracta para obligar siempre a definir una clase hija que implemente los métodos **Exec()** y opcionalmente , el método **Initialize(),** el cual es el encargado de preparar los WebGLShader de vértices y de fragmento y el WebGLProgram.

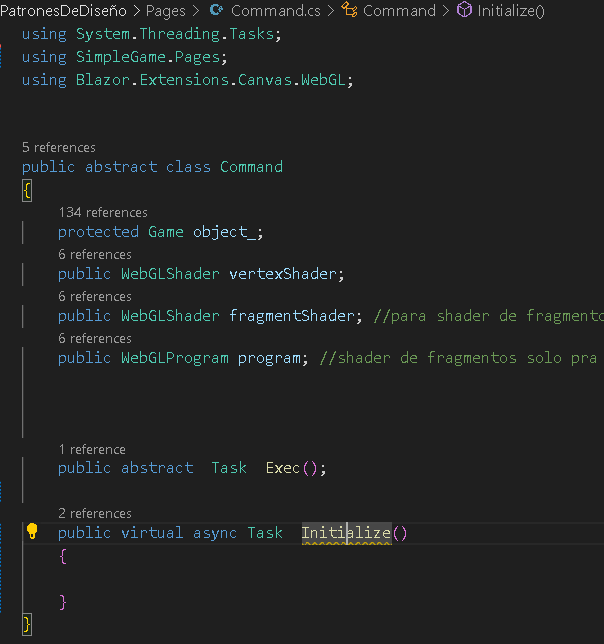


Ilustración . Clase base del patrón Command.

En este caso, he creado dos shaders distintos, mas bien he adaptado los ya existentes, el de malla o principal y el de las sombras.

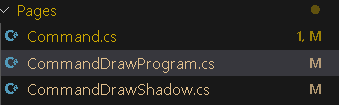


Ilustración .Clases definidas para Command

En la ilustración número 3, se muestra parte del código de una clase hija, en este caso la encargada de las sombras. Se puede observar como cada clase puede definir su propio código para vértices y fragmentos. El constructor nos dará acceso al objeto de tipo Game desde el que se creó. Se puede ver también el funcionamiento del método **Initialize()** encargado de preparar los shaders, y **Exec(),** el encargado de renderizar, sustituye al método **Draw() original.**



Ilustración .Parte de la declaración de clase hija CommandDrawShadow.cs

## *Método* getAttributeLocations(WebGLprogram p)

El método getAttributeLocations(WebGLprogram p), se ha movido a cada clase command, puesto que es un método que dependerá en gran medida de las variables definidas en los shaders, por lo que debe de estar independiente por cada clase. Por ejemplo, en la clase CommandDrawShadow(), se utiliza la variable shadowColor, mientras que en shader de malla, no se utiliza.

## Modificaciones en clase Game.cs

Se crean las variables del tipo de la clase base del patrón de diseño, en este caso, *Command.*

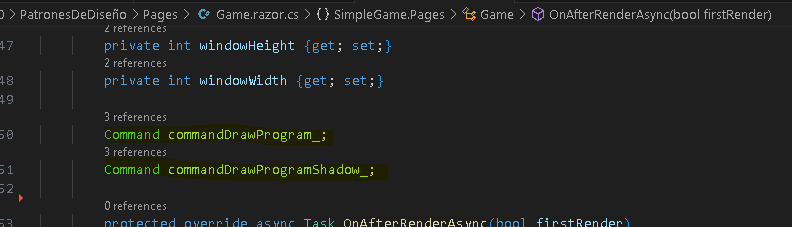


Ilustración . Variables de clase con los diferentes shaders.

En el método OnAfterRenderAsync(), en el código que se ejecutará solo la primera vez, se agregará lo que aparece en la imagen. Tras definir el contexto, creamos los dos shaders pasando como argumento el objeto actual y los inicializamos con su método **Initialize()**.

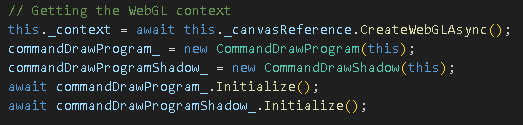


Ilustración . Instanciación e inicialización de los shaders personalizados.

En la ilustración número 6 se puede como han quedado el método **GameLoop()** que es el encargado de llamar al nuevo método **Draw(),** pero esta vez, pasando como parámetro el comando a ejecutar. El método **Draw()** simplemente ejecutará el método **Exec(),** del comando pasado como parámetro, primero se pasa el shader de la malla y después el de las sombras.

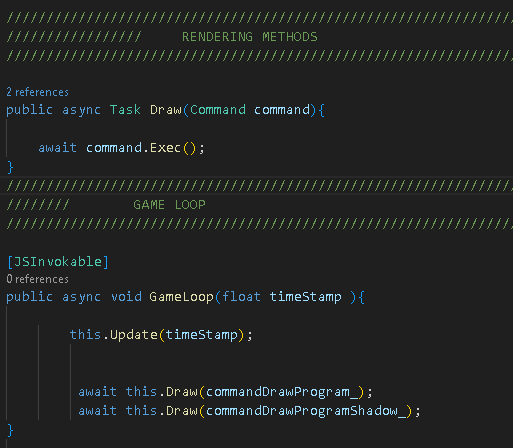


Ilustración . Aspecto métodos GameLoop() y Draw()

Como último paso, se ha simplificado la clase Game.cs, ya que no necesita la declaración de los distintos shaders, las variables que las referencian, el método getAttributeLocations(), ni el método Draw().

## Conclusión

Creo que, gracias a este patrón, ahora el proyecto es mucho más sencillo de administrar y más fácilmente escalable. Ahora no hace falta modificar el código principal para modificar los shaders, y se pueden crear tantos como se quiera, para después simplemente utilizarlos instanciándolos y agregándolos al método GameLoop().

## Process

The behavioral design pattern called Command() has been implemented. Now, each new derived class will implement a new shader with minimal modifications to the original code. This allows for greater scalability, flexibility, independence, and code cleanliness through the separation of concerns. Furthermore, the ability to pass the entire class as parameters provides a lot of flexibility.

## Conclusion

I believe that thanks to this pattern, the project is now much easier to manage and more easily scalable. There is no longer a need to modify the main code to modify the shaders, and you can create as many as you want, and then simply use them by instantiating and adding them to the GameLoop() method.