



**Escuela Superior
de Ingeniería y Tecnología**
Universidad de La Laguna

Programación Optimizada para Videojuegos

Master Universitario en Desarrollo de Videojuegos

Actividad Patrones de diseño

Juan Siverio Rojas

La Laguna, 6 de julio de 2023

Contenido

1.1	Objetivo	2
1.2	Proceso	2
1.3	Método <code>getAttributeLocations(WebGLprogram p)</code>	4
1.4	Modificaciones en clase <code>Game.cs</code>	5
1.5	Conclusión	6
1.6	Process.....	6
1.7	Conclusion	6
Ilustración 1. Clase base del patrón <code>Command</code>		3
Ilustración 2. Clases definidas para <code>Command</code>		3
Ilustración 3. Parte de la declaración de clase hija <code>CommandDrawShadow.cs</code>		4
Ilustración 4. Variables de clase con los diferentes shaders.		5
Ilustración 5. Instanciación e inicialización de los shaders personalizados.		5
Ilustración 6. Aspecto métodos <code>GameLoop()</code> y <code>Draw()</code>		6

1.1 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.

1.2 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`.

```

PatronesDeDiseño > Pages > Command.cs > Command > Initialize()
using System.Threading.Tasks;
using SimpleGame.Pages;
using Blazor.Extensions.Canvas.WebGL;

5 references
public abstract class Command
{
    134 references
    | protected Game object_;
    | 6 references
    | public WebGLShader vertexShader;
    | 6 references
    | public WebGLShader fragmentShader; //para shader de fragmento
    | 6 references
    | public WebGLProgram program; //shader de fragmentos solo pra

    1 reference
    | public abstract Task Exec();

    2 references
    | public virtual async Task Initialize()
    | {
    | }
}

```

Ilustración 1. 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.

Pages	
Command.cs	1, M
CommandDrawProgram.cs	M
CommandDrawShadow.cs	M

Ilustración 2. 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.

```

PatronesDeDiseño > Pages > CommandDrawShadow.cs > CommandDrawShadow > getAttributeLocations(WebGLProgram p)

private const string vsSource=@"
uniform mat4 uModelViewMatrix;
uniform mat4 uProjectionMatrix;
uniform mat4 uNormalTransformMatrix;
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
attribute vec4 aVertexColor;
varying vec4 vVertexPosition;
varying vec4 vVertexNormal;
varying vec4 vVertexColor;
void main(void){
    vVertexPosition = uProjectionMatrix*uModelViewMatrix*vec4(0.5*aVertexPosition,1.0);
    vVertexNormal = uNormalTransformMatrix * vec4(aVertexNormal,0.0);
    vVertexColor=aVertexColor;
    gl_Position = vVertexPosition;
}";

1 reference
private const string fsSource=@"
precision mediump float;
uniform vec4 uShadowColor;
void main(){

    gl_FragColor=uShadowColor;

}";

2 references
public WebGLUniformLocation shadowColor; //para el atributo de sombra.
1 reference
public CommandDrawShadow(Game value)
{
    object_ = value;
}

2 references
public override async Task Initialize()
{
    vertexShader=await object_.GetShader(vsSource,ShaderType.VERTEX_SHADER);
    fragmentShader= await object_.GetShader(fsSource,ShaderType.FRAGMENT_SHADER);
    program= await object_.BuildProgram(vertexShader,this.fragmentShader);
    await object_.context.DeleteShaderAsync(vertexShader);
    await object_.context.DeleteShaderAsync(this.fragmentShader);
}

1 reference
public override async Task Exec()
{
    await object_.context.ClearColorAsync(0, 0, 1, 1);
    await object_.context.ClearDepthAsync(1.0f);
}

```

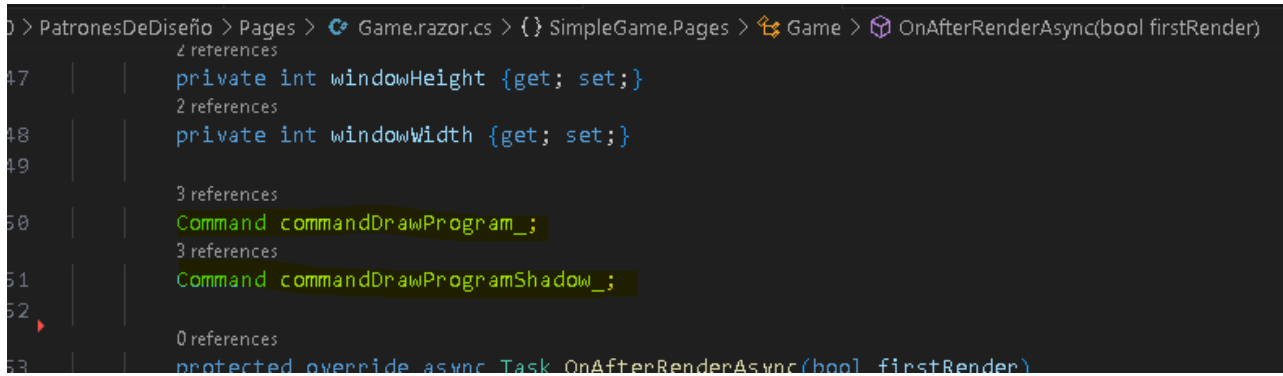
Ilustración 3. Parte de la declaración de clase hija *CommandDrawShadow.cs*

1.3 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.

1.4 Modificaciones en clase Game.cs

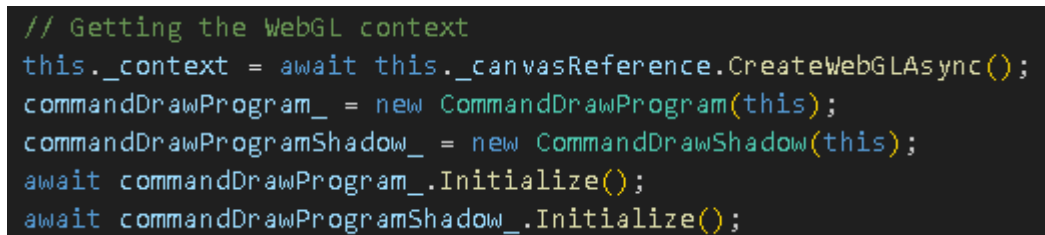
Se crean las variables del tipo de la clase base del patrón de diseño, en este caso, *Command*.



```
0 > PatronesDeDiseño > Pages > Game.razor.cs > {} SimpleGame.Pages > Game > OnAfterRenderAsync(bool firstRender)
2 references
47 |     private int windowHeight {get; set;}
2 references
48 |     private int windowWidth {get; set;}
49 |
3 references
50 |     Command commandDrawProgram_;
3 references
51 |     Command commandDrawProgramShadow_;
52 |
0 references
53 |     protected override async Task OnAfterRenderAsync(bool firstRender)
```

Ilustración 4. 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()*.



```
// Getting the WebGL context
this._context = await this._canvasReference.CreateWebGLAsync();
commandDrawProgram_ = new CommandDrawProgram(this);
commandDrawProgramShadow_ = new CommandDrawShadow(this);
await commandDrawProgram_.Initialize();
await commandDrawProgramShadow_.Initialize();
```

Ilustración 5. 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.

```

////////////////////////////////////
////////////////////////////////////      RENDERING METHODS      //////////////////////////////////////
////////////////////////////////////

2 references
public async Task Draw(Command command){

    await command.Exec();
}

////////////////////////////////////
////////////////////////////////////      GAME LOOP      //////////////////////////////////////
////////////////////////////////////

[JSInvokable]
0 references
public async void GameLoop(float timeStamp ){

    this.Update(timeStamp);

    await this.Draw(commandDrawProgram_);
    await this.Draw(commandDrawProgramShadow_);
}

```

Ilustración 6. 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()*.

1.5 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()*.

1.6 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.

1.7 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.