Documentación de la Clase Game

Gerstep

August 28, 2024

1 Introducción

La clase `Game` es una subclase de `GameWindow`, que forma parte de OpenTK, y representa la ventana principal de un juego o aplicación gráfica. Esta clase es responsable de inicializar y gestionar todos los componentes necesarios para renderizar gráficos 3D, incluyendo la cámara, la entrada del usuario, los shaders, y el proceso de renderización en sí. La clase también gestiona el ciclo de vida del juego, desde la carga de recursos hasta la limpieza final.

Esta clase es parte del espacio de nombres `OpenTK_Tarea_3` y es la estructura central que mantiene unidos todos los componentes del juego.

2 Atributos Privados de la Clase

2.1 _camera

- **Tipo de Dato:** `camera`
- **Descripción:** Instancia de la clase `camera` que controla la posición y orientación de la cámara en la escena 3D. Es fundamental para determinar qué parte de la escena se ve y cómo se ve.

2.2 _input

- **Tipo de Dato:** `Inputs`
- **Descripción:** Instancia de la clase `Inputs` que gestiona las entradas del usuario, incluyendo el teclado y el ratón, y traduce esas entradas en acciones que afectan la cámara.

2.3 _escenario

- **Tipo de Dato:** `Escenario`
- **Descripción:** Instancia de la clase `Escenario` que contiene todos los objetos 3D que se renderizarán en la escena. Cada `Objeto` en el `Escenario` representa un modelo 3D en el mundo del juego.

2.4 _renderizacion

- **Tipo de Dato:** `Renderización`
- **Descripción:** Instancia de la clase `Renderización` que gestiona la configuración de buffers de OpenGL y la renderización de los objetos en pantalla.

2.5 _shader

- **Tipo de Dato:** `Shader`
- **Descripción:** Instancia de la clase `Shader` que maneja el programa de shaders utilizado para aplicar efectos de sombreado y transformar los vértices en la escena.

3 Constructor de la Clase

```
public Game(int width, int height, string title)
```

3.1 Descripción

El constructor de la clase `Game` inicializa la ventana del juego con las dimensiones y el título especificados. También inicializa las instancias de las clases `camera`, `Inputs`, `Escenario`, `Renderización`, y `Shader`, preparando todo lo necesario para el proceso de renderización.

3.2 Parámetros

- `width`: Un entero (`int`) que especifica el ancho de la ventana del juego.
- `height`: Un entero (`int`) que especifica la altura de la ventana del juego.
- `title`: Una cadena de texto (`string`) que especifica el título de la ventana del juego.

3.3 Funcionamiento

El constructor realiza las siguientes acciones:

1. Configura la ventana del juego con las dimensiones y el título proporcionados.
2. Habilita la sincronización vertical (`VSync`) para evitar el tearing en la pantalla.
3. Inicializa las instancias de `_renderizacion`, `_camera`, `_input`, `_escenario`, y `_shader`.
4. Llama al método `CrearObjetosT` para agregar un conjunto de objetos en forma de "T" al escenario.

3.4 Ejemplo de Uso

```
Game game = new Game(800, 600, "Mi Juego 3D");
game.Run(); // Inicia el ciclo de vida del juego
```

4 Métodos de la Clase

4.1 CrearObjetosT (Crear Objetos en Forma de T)

```
private void CrearObjetosT(int cantidad)
```

4.1.1 Descripción

Este método crea un conjunto de objetos 3D en forma de "T" y los agrega al `Escenario`. Los objetos se posicionan en una disposición regular en el espacio 3D.

4.1.2 Parámetro

- `cantidad`: Un entero (`int`) que especifica el número de objetos en forma de "T" que se crearán.

4.1.3 Funcionamiento

El método `CrearObjetosT` sigue los siguientes pasos:

1. Itera desde 0 hasta `cantidad` para crear los objetos.
2. En cada iteración, calcula una posición basada en el índice i, espaciando los objetos uniformemente en las direcciones X, Y y Z.
3. Crea un nuevo `Objeto` en la posición calculada y lo agrega al `Escenario` mediante el método `AgregarObjeto`.

4.1.4 Ejemplo de Uso

```
// Crear 10 objetos en forma de "T" y agregarlos al escenario
CrearObjetosT(10);
```

4.2 OnLoad (Carga de Recursos)

```
protected override void OnLoad()
```

4.2.1 Descripción

Este método se llama cuando se carga la ventana del juego por primera vez. Es responsable de configurar el estado de OpenGL, cargar los shaders, y configurar los buffers para todos los objetos en el escenario.

4.2.2 Funcionamiento

El método `OnLoad` realiza las siguientes acciones:

1. Establece el color de fondo de la ventana utilizando `GL.ClearColor`, configurándolo en un tono verde.
2. Habilita la prueba de profundidad (`GL.Enable(EnableCap.DepthTest)`) para asegurarse de que los objetos se dibujen correctamente en función de su distancia desde la cámara.
3. Activa el programa de shaders utilizando `_renderizacion.UsarProgramaShader`.
4. Itera sobre todos los objetos en el `Escenario`, configurando los buffers para cada uno de sus polígonos mediante el método `ConfigurarBuffers`.

4.2.3 Ejemplo de Uso

```
// Este método se ejecuta automáticamente cuando se carga la ventana del juego
OnLoad();
```

4.3 OnRenderFrame (Renderizar un Frame)

```
protected override void OnRenderFrame(FrameEventArgs e)
```

4.3.1 Descripción

Este método se llama en cada frame para renderizar la escena. Gestiona la entrada del usuario, actualiza las matrices de transformación, y dibuja todos los objetos en el escenario.

4.3.2 Parámetro

- `e`: Un objeto de tipo `FrameEventArgs` que contiene información sobre el tiempo transcurrido desde el último frame, lo que permite que el renderizado sea independiente de la velocidad de fotogramas.

4.3.3 Funcionamiento

El método `OnRenderFrame` realiza las siguientes acciones:

1. Llama al método `_input.HandleInput` para gestionar la entrada del usuario.
2. Calcula la matriz de vista llamando a `_camera.GetViewMatrix`.
3. Calcula la matriz de proyección utilizando `Matrix4.CreatePerspectiveFieldOfView`.
4. Limpia la pantalla y el buffer de profundidad utilizando `GL.Clear`.
5. Activa el shader llamando a `_shader.Use`.

6. Envía las matrices de vista y proyección al shader mediante los métodos `SetMatrix4`.

7. Dibuja todos los objetos en el escenario llamando a `_escenario.DibujarEscenario`.

8. Cambia el buffer de pantalla con `SwapBuffers` para mostrar el frame renderizado.

4.3.4 Ejemplo de Uso

```
// Este método se ejecuta automáticamente en cada frame para renderizar la escena
OnRenderFrame(e);
```

4.4 OnUnload (Liberar Recursos)

```
protected override void OnUnload()
```

4.4.1 Descripción

Este método se llama cuando la ventana del juego se está cerrando. Es responsable de liberar los recursos gráficos utilizados durante el juego, como buffers y shaders, para evitar pérdidas de memoria.

4.4.2 Funcionamiento

El método `OnUnload` realiza las siguientes acciones:

1. Llama a `_renderizacion.LimpiarBuffers` para eliminar los buffers de OpenGL.

2. Llama a `_shader.Dispose` para liberar los recursos asociados con los shaders.

4.4.3 Ejemplo de Uso

```
// Este método se ejecuta automáticamente cuando se cierra la ventana del juego
OnUnload();
```

4.5 OnResize (Redimensionar la Ventana)

```
protected override void OnResize(ResizeEventArgs e)
```

4.5.1 Descripción

Este método se llama cuando la ventana del juego es redimensionada. Ajusta el viewport de OpenGL para asegurarse de que el contenido se renderice correctamente según las nuevas dimensiones de la ventana.

4.5.2 Funcionamiento

El método `OnResize` realiza las siguientes acciones:

1. Llama al método base `base.OnResize(e)` para manejar cualquier comportamiento predeterminado.
2. Ajusta el viewport de OpenGL utilizando `GL.Viewport` para que coincida con el nuevo tamaño de la ventana.

4.5.3 Ejemplo de Uso

```
// Este método se ejecuta automáticamente cuando la ventana se redimensiona
OnResize(e);
```

4.6 VertexShaderSource (Código Fuente del Shader de Vértices)

```
private const string VertexShaderSource
```

4.6.1 Descripción

Este es el código fuente del shader de vértices, escrito en GLSL. El shader de vértices es responsable de transformar las posiciones de los vértices de los modelos 3D de espacio local a espacio de clip.

4.6.2 Funcionamiento

El shader de vértices realiza las siguientes acciones:

1. Recibe las posiciones de los vértices (`aPosition`) y los colores (`aColor`).
2. Calcula la posición final de cada vértice en el espacio de clip multiplicando las matrices de modelo, vista y proyección con la posición del vértice.
3. Pasa el color del vértice al shader de fragmentos.

4.6.3 Código Ejemplo

```
private const string VertexShaderSource = @"
    #version 330 core
    layout(location = 0) in vec3 aPosition;
    layout(location = 1) in vec3 aColor;

    uniform mat4 model;
    uniform mat4 view;
    uniform mat4 projection;

    out vec3 ourColor;
```

```
    void main()
    {
        gl_Position = projection * view * model * vec4(aPosition, 1.0);
        ourColor = aColor;
    }";
```

4.7 FragmentShaderSource (Código Fuente del Shader de Fragmentos)

```
private const string FragmentShaderSource
```

4.7.1 Descripción

Este es el código fuente del shader de fragmentos, escrito en GLSL. El shader de fragmentos es responsable de calcular el color final de los píxeles que se muestran en la pantalla.

4.7.2 Funcionamiento

El shader de fragmentos realiza las siguientes acciones:

1. Recibe el color de cada fragmento (`ourColor`) desde el shader de vértices.
2. Establece el color final del fragmento como `vec4(ourColor, 1.0)`, lo que significa que el color del fragmento será el mismo que el color del vértice.

4.7.3 Código Ejemplo

```
private const string FragmentShaderSource = @"
    #version 330 core
    precision highp float;
    in vec3 ourColor;
    out vec4 FragColor;

    void main()
    {
        FragColor = vec4(ourColor, 1.0);
    }";
```

5 Conclusión

La clase `Game` es la estructura central de una aplicación gráfica 3D utilizando OpenTK. Al gestionar la ventana del juego, la cámara, la entrada del usuario, los shaders y la renderización, esta clase facilita la creación y gestión de entornos 3D complejos. Con su enfoque modular y el manejo eficiente de recursos, la clase `Game` proporciona una base sólida para el desarrollo de aplicaciones gráficas interactivas y dinámicas.