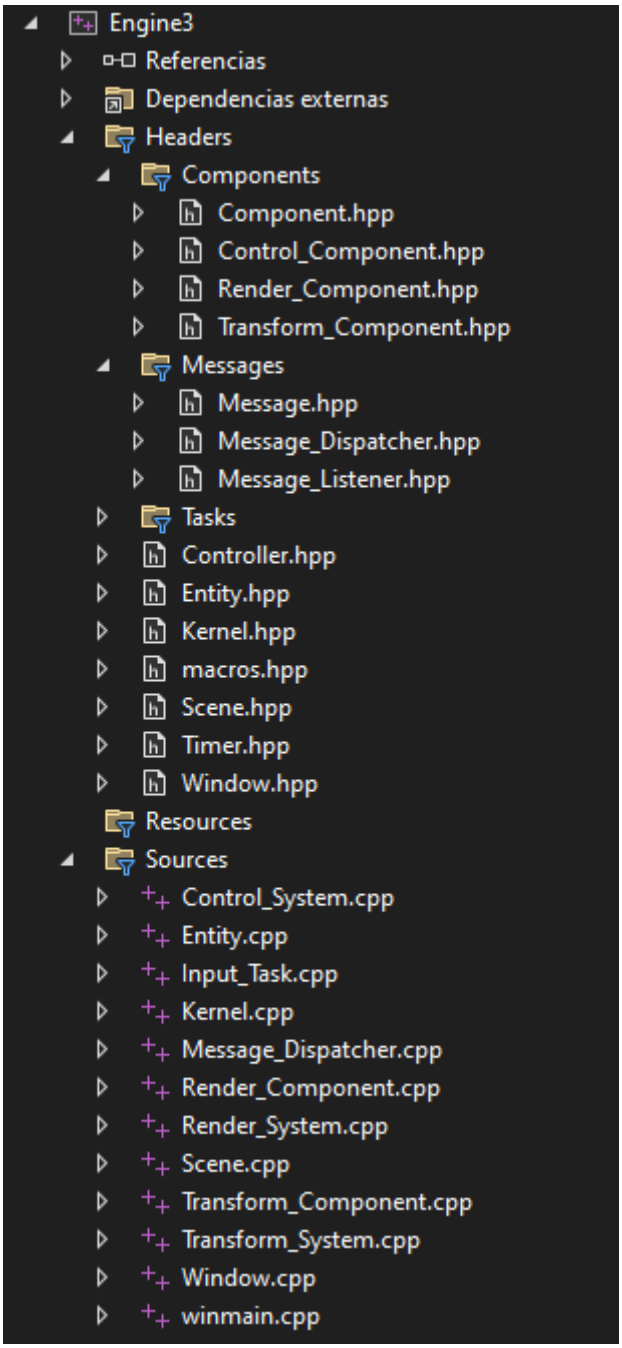
Documentación Práctica 1 Motores



Para esta práctica se ha optado por una composición de entidades, componentes y sistemas (ECS). El sistema ECS contiene una serie de sistemas que registran componentes de su mismo tipo. Todo ello luego se maneja desde un control de escenas con un kernel para controlar el bucle principal.

Las entidades contienen un mapa de componentes asociados con un identificador propio para cada componente, el cual se define en el archivo xml de carga. Contiene una

referencia a un componente de Transform ya que se sobreentiende que todas las entidades tendrán este de base como hacen otros motores como Unity o Unreal Engine. Las entidades se guardan en la escena con un identificador único en un mapa. Permiten añadir, quitar y recibir componentes. El resto de los comportamientos se deja a los sistemas y componentes que los expandan ellos.

Los componentes son lo que dan a las entidades ciertos atributos para poder ser renderizados o movidos en la escena, guardan la información necesaria para que los sistemas apliquen comportamientos. Existe una clase padre de componente la cual solo contiene un método que es el de parsear, lo cual permite recibir un nodo de xml usando la librería rapidXML y recoger toda su información, el parseo de información al ser único por nodo se deja que se hereden a los hijos. Existen los siguientes componentes:

- **Control_Component**: Permite controlar entidades desde fuera del motor.
- **Render_Component**: Guarda información sobre el tipo de objeto que se tiene que renderizar. Ya sea una cámara, un objeto o una luz.
- **Transform_Component**: Guarda información sobre la posición de la entidad en la escena.

Para los sistemas se optó por un modelo de Tasks, que permiten crear y actualizar comportamientos que ejecutar dentro del kernel en tiempo de ejecución. La clase base de Task expone tres métodos principales de **initialize, finalize y execute**, para que las tareas puedan añadir aquí sus comportamientos de comienzo, final y actualización en tiempo de ejecución. Hay tareas hijas que no son sistemas ya que no contienen componentes y pueden ejecutarse independientemente, luego existen otras que son sistemas. Existen las siguientes tareas:

- **Control_System**: Este sistema se encarga de crear componentes en entidades, pudiendo crearlos sobre la marcha mientras se van leyendo del archivo xml de carga.
- **Input_Task** (No es un sistema): Se encarga de controlar el input del jugador en el juego. Lo recibe y manda los pertinentes mensajes a los correspondientes sistemas.
- **Render_System**: Es el sistema encargado de actualizar los visuales de los componentes de renderizado.
- **Transform_System**: Se encarga de actualizar la posición de las entidades en pantalla. Guarda una matriz de posición y la actualiza. Permite herencia y jerarquía.

La escena contiene como se comentó anteriormente un mapa de entidades y un mapa de sistemas a la vez que un kernel y un controlador de mensajería. El mapa de entidades y sistemas se va poblando según se van cargando los contenidos desde el archivo xml.

Para cargar el archivo de la escena se le pasa una dirección de memoria, luego lo primero que hace la escena es ir recorriendo ese archivo XML buscando entidades y mirando que componentes tienen esas entidades dentro. En base a que componentes tengan se van

creando sistemas para manejarlos dentro de la propia escena en tiempo de ejecución usando una fábrica de sistemas.

La información XML se muestra de la siguiente manera en base al tipo de objeto que estemos cargando.

- Ventana

```
<window name="Game" width="800" height ="700" fullscreen="false"> </window>
```

- Cámara

```
<entity name ="camera">
    <component type="transform" pos_x = "0" pos_y = "3" pos_z = "50" rot_x = "0" rot_y = "0" rot_z = "0"  scl_x = "1" scl_y = "1
    " scl_z = "1"></component>
    <component type="render" node="camera" fov="20" near="1" far="5000" ratio="1"></component>
</entity>
```

- Luz

```
<entity name ="light">
    <component type="transform" pos_x = "10" pos_y = "10" pos_z = "10" rot_x = "0" rot_y = "0" rot_z = "0"  scl_x = "1" scl_y =
    "1" scl_z = "1"></component>
    <component type="render" node="light"></component>
</entity>
```

- Objetos Físicos

```
<entity name ="sphere">
    <component type="transform" pos_x = "0" pos_y = "0" pos_z = "0" rot_x = "0" rot_y = "0" rot_z = "0"  scl_x = "5" scl_y = "5"
     scl_z = "5"></component>
    <component type="render" node="modelObj" mesh_path="../../assets/objects/sphere.obj"></component>
</entity>
```

Cada componente tiene su propia nomenclatura personalizada en XML:

- **Transform**: Tipo (transform) + Posición (x,y,z) + Rotación (x,y,z) + Escala (x,y,z)
- **Render**:
    - **Cámara**: Tipo (render) + Nodo (Cámara, Luz, ModelObj) + Fov + Near + Far + Ration
    - **Luz**: Tipo (render) + Nodo (Cámara, Luz, ModelObj)
    - **Objeto**: Tipo (render) + Nodo (Cámara, Luz, ModelObj) + Ruta del archivo

El kernel se encarga de actualizar las tareas que tiene la escena y los sistemas para su correcta ejecución en tiempo de juego. Guarda las tareas en una lista y contiene métodos internos de **initialize, finalize y run**. En donde, consecuentemente, se ejecutan los métodos de **initialize, finalize y excute** de las tareas. Tiene una condición de salida que puede ser llamada desde la escena al cerrar la ventana o si se fuera a cambiar de escena por algún motivo.

Para el sistema de mensajería se ha usado el clásico de receptores y emisores, cuando ocurre algo que permite recibir el trigger de que se ha de mandar un mensaje se crea uno y se lanza desde la escena.