Animación por Computador y Videojuegos

Carga de Máquinas de estados finitas a través de un parser

Trabajo del Alumno: Jorge Izquierdo Ciges

Ficheros que lo componen:

FSMParser.h
FSMParser.cpp

Ficheros de las FSM:

FSM_game.html
FSM_ship.html
FSM_supplyship.html

Contenido de la modificación:

Se ha implementado una clase (FSMReader). Esta clase ofrece la funcionalidad de un parser html que es capaz de leer e interpretar una estructura DOM html que se ha definido para este trabajo. Para respetar el diseño anterior para generar un FSM, se han tenido que tomar una serie de decisiones de diseño. Se ha de tener en cuenta que las FSM requieren de la posibilidad de llamar a una serie de funciones. Dado que en este proyecto no se emplea ningún tipo de lenguaje de script que nos sirva para exponer públicamente las funciones para cada FSM, el programador debe de especificarle al parser las funciones que puede utilizar el FSM, para ello se le expone públicamente las funciones mediante un string (que deberá tener el mismo nombre de la función) y un puntero a función. Esto se realiza mediante la función “addPublicFunction”. Es importante que este paso se realice antes de la lectura del archivo, ya que si no conoce las funciones en tiempo de lectura del archivo, se generará un error de lectura. Una vez se ha realizado la lectura del archivo la clase ha generado, internamente, un vector de estados y un estado inicial. Estos parámetros se pueden recuperar mediante una serie de getters para pasárselos a la FSM.

El proceso para la carga de una FSM consiste en los siguientes pasos:

Inicialización de las variables

```
CFile               HTMLFile;       //Variable para el parser HTML
FSMReader           reader;         //Especialización FSMReader
CLiteHTMLReader     HTMLReader;     //El lector HTML

//Inicialización del parser
reader.Init(".\\FSM_supplyship_Log.txt");
```

Registrar las funciones en el lector de FSM

```
reader.addPublicFunction("NULL",NULL);
reader.addPublicFunction("CSS_FSM_Move",(mFuncPtr)CSS_FSM_Move);
reader.addPublicFunction("CSS_display",(mFuncPtr)CSS_display);
reader.addPublicFunction("CSS_FSM_Healthing",(mFuncPtr)CSS_FSM_Healthing);
reader.addPublicFunction("CSS_FSM_Dye",(mFuncPtr)CSS_FSM_Dye);
reader.addPublicFunction("CSS_FSM_Death",(mFuncPtr)CSS_FSM_Death);
```

Lectura del archivo FSM

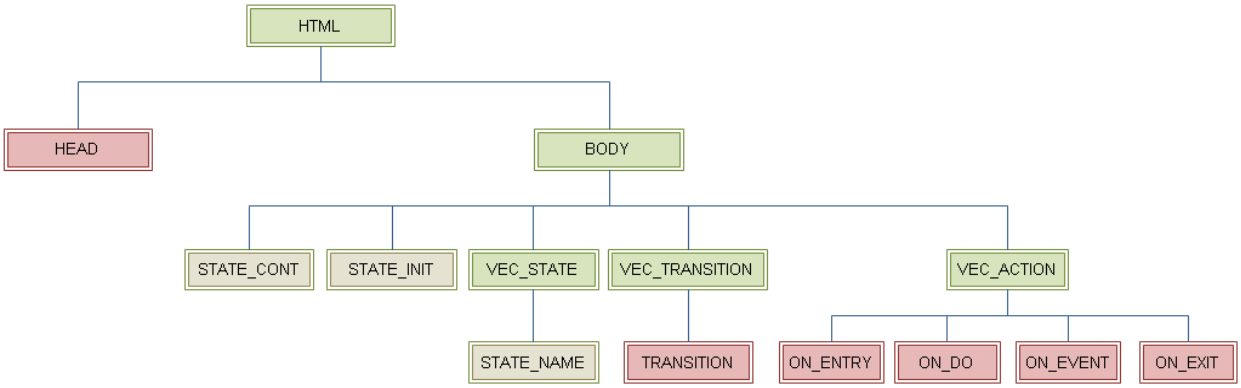
```
HTMLReader.setEventHandler(&reader);
if (HTMLFile.Open(_T(".\\FSM_supplyship.html"), CFile::modeRead))
{
        HTMLReader.ReadFile((HANDLE) HTMLFile.m_hFile);
        HTMLFile.Close();
}
```

Creación de la FSM

```
AI->addStates(reader.getFSMStatesVector(),reader.getFSMNumStates());
AI->SetState(reader.getFSMInitState());
```

DOM FSM html:

Esencialmente el DOM FSM tiene la siguiente estructura jerárquica



HEAD

El grupo HEAD tiene la misma estructura de otros ficheros Html del proyecto

```
<HEAD>
        <TITLE>Space Invaders</TITLE>
        <VERSION>1.0.0</VERSION>
        <TYPE>FSM</TYPE>
</HEAD>
```

Identifica inequivocamente el nombre, la versión del proyecto y el tipo de script, en este caso FSM.

BODY

En el body se encuentran todos los campos indispensables para realizar una FSM. Body debe tener la siguiente estructura:

```
<BODY>
        <STATE_CONT></STATE_CONT>
        <STATE_INIT></STATE_INIT>
        <VEC_STATE>
                <---------------->
        </VEC_STATE>
        <VEC_TRANSITION>
                <---------------->
        </VEC_TRANSITION>
        <VEC_ACTION>
                <---------------->
        </VEC_ACTION>
</BODY>
```

El campo STATE_CONT sirve para indicar el número de estados presentes en el FSM, a su vez, STATE_INIT indica cual de todos los estados es considerado como el estado inicial de la máquina. Los campos VEC_STATE, VEC_TRANSITION y VEC_ACTION sirven para proporcionar la información necesaria sobre los estados, las transiciones y las acciones.

VEC_STATE

VEC_STATE sirve a modo de vector donde se enumeran los N estados mediante sus nombres con la etiqueta STATE_NAME.

```
<VEC_STATE>
        <STATE_NAME></STATE_NAME>
</VEC_STATE>
```

VEC_TRANSITION

En este caso, la estructura almacena las diferentes transiciones. Una transición tiene que estar formada por un nombre, un estado de origen y un estado de destino.

```
<VEC_TRANSITION>
        <TRANSITION>
                <T_NAME></T_NAME>
                <T_ORIGIN></T_ORIGIN>
                <T_DESTINATION></T_DESTINATION>
        </TRANSITION>
</VEC_TRANSITION>
```

VEC_ACTION

Por otra parte, VEC_ACTION almacenará las diversas acciones que puedan existir en una FSM sin discriminar el estado origen. Aunque un estado pueda tener un número indeterminado de eventos (ON_EVENT) únicamente puede tener una acción de entrada (ON_ENTRY), de realización (ON_DO) o de salida (ON_EXIT). Si bien esto es así a nivel de FSM, el lector no realizará ninguna comprobación al respecto. Esto significa que si un estado incumple estra restricción, el comportamiento no está definido y tenderá a asumir como única acción de dicho tipo la última que haya leido.

```
<VEC_ACTION>
        <ON_ENTRY>
                <---------------->
        </ON_ENTRY>
        <ON_DO>
                <---------------->
        </ON_DO>
        <ON_EVENT>
                <---------------->
        </ON_EVENT>
        <ON_EXIT>
                <---------------->
        </ON_EXIT>
</VEC_ACTION>
```

EVENTOS

Fundamentalmente, todos los eventos tiene una idea similar. Estan formados por un tipo (0 acción, 1 evento) un nombre, una función a ejecutarse y un número de estado origen. En el caso de los eventos ejecutados por una transición además se necesitará incluir el nombre que identifica a la transición.

```
<ON_ENTRY>
        <ENTRY_TYPE></ENTRY_TYPE>
        <ENTRY_NAME></ENTRY_NAME>
        <ENTRY_FUNC></ENTRY_FUNC>
        <ENTRY_ORIGIN></ENTRY_ORIGIN>
</ON_ENTRY>
<ON_DO>
        <DO_TYPE></DO_TYPE>
        <DO_NAME></DO_NAME>
        <DO_FUNC></DO_FUNC>
        <DO_ORIGIN></DO_ORIGIN>
</ON_DO>
<ON_EVENT>
        <EVENT_TYPE></EVENT_TYPE>
        <EVENT_NAME></EVENT_NAME>
        <EVENT_FUNC></EVENT_FUNC>
        <EVENT_ORIGIN></EVENT_ORIGIN>
        <EVENT_TRANSITION></EVENT_TRANSITION>
</ON_EVENT>
<ON_EXIT>
        <EXIT_TYPE></EXIT_TYPE>
        <EXIT_NAME></EXIT_NAME>
        <EXIT_FUNC></EXIT_FUNC>
        <EXIT_ORIGIN></EXIT_ORIGIN>
</ON_EXIT>
```

MODELO

El archivo: FSM_sample_struct.txt tiene una estructura de ejemplo para emplear en la generación de nuevas FSM.

Otras modificaciones

Las clases FSM y State han tenido que ser modificados para permitir el uso de un número de caracteres ilimitado en los diversos nombres e identificaciones. Para ello se han adaptado al uso de las clases STL de C++.