TP Videojuegos
Ejercicio SDL_Net

El objetivo de este ejercicio es practicar lo que hemos visto en clase sobre SDL_Net.

Parte 0: Descargar el código

Descarga el proyecto correspondiente desde el campus y ejecútalo. El juego actual consiste en un caza que puede mover y disparar.

Parte 1: Modificar main.cpp

Añade los siguientes métodos en el archivo `main.cpp`

```
#include "AsteroidsGame.h"
#include "ClientInfo.h"
#include "Server.h"

void clientMode(char* host, int port) {
    ClientInfo::initInstance(host, port);
    AsteroidsGame g;
    g.start();
}

void severMode(int port) {
    Server s;
    s.start(port);
}
```

El método `severMode` empieza un servidor en el puerto "`port`". El método `clientMode` conecta al servidor (en la dirección indicada por "`host`" y "`port`") y empieza el juego.

De momento no hay que saber mucho sobre la clase `Server`, sólo que asigna identificadores únicos a los clientes y que cada mensaje que recibe lo reenvía a todos los clientes. La clase `ClientInfo` es un Singleton que usamos para establecer la conexión con el servidor y nos permite:

1. Consultar el identificador del cliente usando `ClientInfo::instance()->getClientId()`
2. Enviar y recibir mensajes usando `ClientInfo::instance()->getConnection()`

TP Videojuegos, Práctica 5, Fecha Limite: 24/04/2019 a las 09:00.

En este ejercicio no vamos a enviar/recibir mensajes usando `getConnection()` directamente, sino a través de la clase `NetworkMessenger`.

Cambia el método `main` al siguiente para poder ejecutar el programa en modo `server` o `client` depende de los parámetros de la línea de comandos:

```
  int main(int ac, char** av) {
     if (SDLNet_Init() < 0) {
        cout << "Error: " << SDLNet_GetError() << endl;
        cout << "Connected: " << ClientInfo::instance()->getClientId() << endl;
        exit(1);
     }
     if ( ac == 3 && strcmp(av[1],"server") == 0) {
        severMode(atoi(av[2]));
     } else if (ac == 4 && strcmp(av[1],"client") == 0 ) {
        clientMode(av[2],atoi(av[3]));
     } else {
        cout << "Usage: " << endl;
        cout << "  " << av[0] << " client host port " << endl;
        cout << "  " << av[0] << " server port " << endl;
     }
}
```

Para probar los cambios, después de haber compilado el programa, abre una consola y ejecuta el servidor:

```
cd C:\hlocal\SDLProject\SDLProject
..\bin\SDLProjectDebug.exe server 2000
```

y abre otras dos consolas y en cada ejecuta un cliente:

```
cd C:\hlocal\SDLProject\SDLProject
..\bin\SDLProjectDebug.exe client localhost 2000
```

Observa que estamos ejecutando el programa desde `C:\hlocal\SDLProject\SDLProject` para que encuentrae el directorio `resources`. En principio no se ve ninguna diferencia con respecto a la versión original, sólo observa los mensajes en las consolas, etc.

En lugar de ejecutar desde la consola, se puede guardar los comandos en 2 archivos de texto `server.bat` y `client.bat` y ejecutarlos desde el "*File Explorer*" directamente con doble clic como cualquier otro programa. Además, se puede ejecutar desde máquinas distintas cambiando `localhost` por la dirección IP de la máquina del servidor -- ten cuidado porque hay que usar el mismo compilador en las dos máquinas.

En lugar de ejecutar el servidor por separado, se puede hacer que el modo `server` empiece el servidor en un hilo y ejecute un cliente. Para conseguir esto cambia el método `serverMode` al siguiente:

```
#include <thread>

void severMode(int port) {
   Server s;
   thread serverThread( [port,&s]() {
         s.start( port );
   });
    char* host = "localhost";
    clientMode(host,port);

    s.stop();
    serverThread.join();
}
```

En este caso hay que ejecutar el programa sólo dos veces: una vez en modo `server` (el primer jugador) y otra en modo `client` (el segundo jugador).

IMPORTANTE: antes de compilar de nuevo, hay que cerrar el servidor y los clientes, si no `Visual Studio` queja de que `SDLProjectDebug.exe` está ocupado, etc.

Parte 2: Transmitir mensajes entre los clientes

Vamos a modificar el programa para que todos los mensajes que se envían entre los containters/componentes de en un cliente llegan también a los containters/componentes del otro cliente. Para eso vamos a usar la clase `NetworkMessenger`. Es una clase que se puede registrar como observador y a partir de ese momento cualquier mensaje que recibe lo envía automáticamente

al servidor usando

```
ClientInfo::instance()->getConnection()->sendMessage(&msg)
```

Además, `NetworkMessenger` tiene un método `update()` para recibir los mensajes del servidor y enviarlos a los componentes/containers locales.

En la clase `AsteroidsGame`

1. Añade un nuevo atributo `networkMessenger_` de tipo `NetworkMessenger` y inicializarlo en el constructor usando `networkMessenger_(this)`
2. En el método `iniGame()`, registra `networkMessenger_` como observador al juego para que envíe todos los mensajes al servidor
3. El el método `start()`, antes de la llamada a `handleInput` añade una llamada al `networkMessenger_.update()` para sacar los mensaje pendientes y enviarlos a los componentes/containers locales

Para probar los cambios, ejecuta los dos cliente de nuevo, ahora cuando el caza dispara las balas tienen que aparecer en los dos clientes porque el mensaje `Shoot` llega a los dos.

Parte 3: Sincronizar el Fighter

Vamos a cambiar el código para que sólo el cliente 0 controle el caza y para que el cliente 1 tenga una réplica.

Modifica la clase `Fighter`

1. Añade un nuevo atributo `broadcastInfoPC_` de tipo `BroadCastObjectInfoPC`. Es un componente de física que envía continuamente mensajes con el estado del caza.
2. Modifica el constructor para que añade los componentes (actuales y el nuevo `broadcastInfoPC_`) sólo si cumple la siguiente condición

   ```
   (ClientInfo::instance()->getClientId() == 0 && getId() == msg::Fighter_0)
   ```

   En el caso que no cumpla añade sólo el componente de gráfica `fighterImage_`.

3. Modifica el método `receive` del caza para que cuando reciba un mensaje de tipo `RemoteObjectInfo` modifica el estado del caza con la información que lleva el mensaje, pero

sólo si cumple la siguiente condición:

```
(ClientInfo::instance()->getClientId() == 1 && getId() == msg::Fighter_0)
```

Para probar las modificaciones ejecuta los dos clientes de nuevo, ahora sólo el cliente 0 controla el caza y el cliente 1 tiene una réplica.

Parte 4: Añadir otro caza

Vamos a modificar el juego para que el cliente 1 tenga su propio caza con una réplica en el cliente 0.

En la clase `AsteroidsGame`:

1. Añade un nuevo atributo `fighter1_` de tipo `Fighter` y inicializarlo en la constructora usando `fighter1_(msg::Fighter_1,this)`
2. Modifica el método `iniGame()` para incorporar el nuevo caza al juego

En `Fighter.cpp`

1. Modifica la primera condición del apartado anterior (en el constructor) a

```
(ClientInfo::instance->getClientId() == 0  && getId() == msg::Fighter_0) ||
    (ClientInfo::instance->getClientId() == 1 && getId() == msg::Fighter_1)
```

2. Modifica la segunda condición del apartado anterior (en el método `receive`) a

```
(ClientInfo::instance()->getClientId() != msg.clientId_ &&
     getId() == msg.sender_)
```

3. Modifica el método `initFighter()` para que coloque el caza con el identificador msg::`Figher_0` a la izquierda y el caza con el identificador `msg::Fighter_1` a la derecha.

Para probar las modificaciones ejecuta los dos clientes de nuevo, ahora cada cliente controla su propio caza y los dos clientes están sincronizados.