Proyecto

X-Kating

Grupo

WASTED HORCHATA

Documento de diseño técnico del motor de red

Hito: 1
Fecha entrega: 22-12-2017
Versión: 2

Componentes:

- Luis González Aracil
- Laura Hernández Rielo
- Adrián Francés Lillo
- Pablo López Iborra
- Alexei Jilinskiy

1. Introducción

A continuación se detallan los detalles técnicos del motor de red que va a utilizar nuestro proyecto. Estamos trabajando sobre la tecnología RakNet, la cual proporciona las funcionalidades básicas para la implementación del motor.

2. Diseño técnico del servidor

El proceso de comunicación se realiza a través de la clase base de Raknet “RakPeerInterace”, que es la que inicializa la conexión y el socket con el puerto de escucha correspondiente, tal y como vemos a continuación.

```
peer=RakNet::RakPeerInterface::GetInstance();
//Then we initialize the socket and set his socketFamily (for default communication, use AF_INET)
RakNet::SocketDescriptor socket(PORT,   0);
socket.socketFamily = AF_INET;

peer->Startup(MAXCLIENTS, &socket, 1);
peer->SetMaximumIncomingConnections(MAXCLIENTS);
```

A continuación el juego gestiona las conexiones entrantes, aceptandolas y actualizando el número de usuarios activos. También se encarga de controlar que la conexión con los jugadores siga activa.

Toda la comunicación se realiza enviando y recibiendo paquetes de información. Para ello, en el servidor se definen dos funciones, update() y updateLobby(), donde recibimos los paquetes, los tratamos, y en función del evento ocurrido enviamos al cliente o clientes debidos. La lectura de los paquetes se realiza de la siguiente forma:

```
RakNet::Packet* packet;
packet = peer->Receive();
if(packet)
{
    switch(packet->data[0])
```

Donde data[0] es un enumerator de todos los posibles mensajes de la red, tanto los propios definidos por RakNet como los propios.

Una vez los jugadores en el lobby decidan iniciar la partida, se cierra el lobby y se inicia la partida. El servidor otorga una posición inicial al azar (de entre una serie de posiciones iniciales predefinida) a cada jugador y da por comenzada la partida.

```
stream.Write((unsigned char)ID_GAME_START);
peer->Send(&stream, HIGH_PRIORITY, RELIABLE, 0, RakNet::UNASSIGNED_RAKNET_GUID, true);
for(int i=0; i<nPlayers; i++)
{
    //This switch marks the predefined positions for the players. Later this will be on map's info
    switch(i)
    {
        case 0:
            x=-35.f;
            y=5.f;
            z=-20.f;
            break;
        case 1:
            z=-10.f;
            break;
```

Aquí se ve el proceso de inicio de la partida. En primer lugar, se transmite a cada jugador la señal de inicio y a continuación tanto la posición inicial como la id de red a cada uno de los jugadores.

Desde ese momento, el servidor se ocupa de la lectura de los distintos mensajes de red y la transmisión de su resultado a otros. Los mensajes, según el papel del servidor en ellos, se pueden dividir en dos categorías.

Por un lado, en ciertos mensajes el papel del servidor es únicamente el de transmitir la información recibida, como es el caso de las posiciones de los jugadores, mientras que en otros es el servidor el que realiza la mayor parte de la carga de trabajo, como es en la creación de los objetos. Cada uno de los mensajes está definido en el apartado “Llamadas entre Cliente y Servidor”.

3. Diseño técnico del cliente

Por el lado del cliente, la gestión de la conexión en red vendrá dada por el NetworkManager, el cual también leerá los datos necesarios

para la conexión de un XML. El juego de cada uno de los clientes contará con:

- Varios **NetworkComponents**, asociados a cada uno de los GameObjects del juego que necesiten recibir inputs de la red, los cuales se comunicarán directamente con el manager para leer los datos que reciban del servidor remoto. En ciertos casos, este componente sustituirá a otro u otros para ocupar su lugar. Por ejemplo, en el caso de los otros jugadores, sustituirá a su componentes de IA, ya que lógicamente se tratará de otro jugador en red.

- **El NetworkManager.** Aparte de la comunicación directa tanto con el servidor remoto como con sus componentes asociados, se encargará de obtener y controlar las estadísticas de la conexión, de modo que si ésta cayera se lo notificaría al jugador por pantalla. Además, el propio NetworkManager lanzará ciertos eventos, como por ejemplo las colisiones, y se encargará de aplicar las correcciones necesarias para que la versión del cliente sea igual a la del servidor.

El proceso de comunicación del cliente con el servidor será el mismo que hemos definido en el apartado anterior, solo que a la inversa. Ya que la velocidad de comunicación será distinta para cada cliente dependiendo de la conexión, estos intentarán transmitir la información siempre lo más rápido que sea posible, con el objetivo de minimizar el ping de cada cliente.

4. Proceso de comunicación inicial

5. Llamadas entre cliente-servidor

Detallamos aquí los diferentes códigos que se utilizan en el proceso de comunicación entre cliente-servidor, los parámetros que se pasan en el paquete de datos y el resultado de los mismos

Mensajes base de Raknet:

- **ID_REMOTE_CONNECTION_LOST // ID_CONNECTION_LOST // ID_REMOTE_DISCONNECTION_NOTIFICATION // ID_DISCONNECTION_NOTIFICATION :** Notifican de la desconexión de un jugador del servidor, de diferentes maneras. Tanto desconexión propia, repentina o por parte del server.

```
case ID_REMOTE_CONNECTION_LOST:
    nPlayers--;
    std::cout << "Client disconnected from the server" << std::endl;
    std::cout << "Number of players: " << nPlayers << std::endl;
    break;
case ID_CONNECTION_LOST:
    nPlayers--;
    std::cout << "Client disconnected from the server" << std::endl;
    std::cout << "Number of players: " << nPlayers << std::endl;
    break;
case ID_REMOTE_DISCONNECTION_NOTIFICATION:
    nPlayers--;
    std::cout << "Client disconnected from the server" << std::endl;
    std::cout << "Number of players: " << nPlayers << std::endl;
    break;
case ID_DISCONNECTION_NOTIFICATION:
    nPlayers--;
    std::cout << "Client disconnected from the server" << std::endl;
    std::cout << "Number of players: " << nPlayers << std::endl;
    break;
```

Server side

- **ID_NEW_INCOMING_CONNECTION:** Un nuevo cliente se ha conectado al servidor. En este caso cogemos el SystemAdress del cliente y lo añadimos a la lista de jugadores.

```
case ID_NEW_INCOMING_CONNECTION:
    nPlayers++;
    std::cout << "New client in the server" << std::endl;
    std::cout << "Number of players: " << nPlayers << std::endl;
    players.push_back(packet->systemAddress);
    break;
```

Server side

Nuestros mensajes:

- **ID_GAME_START: El cliente envía al servidor que quiere empezar la partida.** El servidor ejecuta StartGame() en el que el juego empieza. En esta función verifica que se reciben las condiciones necesarias para empezar la partida (que no esté

empezada y que haya más de 1 jugador), y envía la señal a todos los jugadores de empezar la partida, y de crear el jugador principal y a los otros jugadores.

```
case ID_GAME_START:
    startGame();
    break;
```

Recepción del servidor de la orden

```
case ID_GAME_START:
    std::cout << "Game started " << std::endl;
    started = true;
    break;
```

Recepción del cliente de la orden de empezar

```
RakNet::BitStream stream;
started=true;
std::cout << "Starting game" << std::endl;
stream.Write((unsigned char)ID_GAME_START);
peer->Send(&stream, HIGH_PRIORITY, RELIABLE, 0, RakNet::UNASSIGNED_RAKNET_GUID, true);
for(int i=0; i<nPlayers; i++)
{
```

Función StartGame() invocada por el servidor. Mensaje a los demás jugadores para empezar.

```
stream.Write((unsigned char)ID_CREATE_PLAYER);
//Its Network ID;
stream.Write(i);
//Its position, later will be removed
stream.Write(x);
stream.Write(y);
stream.Write(z);
peer->Send(&stream, HIGH_PRIORITY, RELIABLE, 0, players[i], false);
//Repeat for broadcast
stream.Reset();
stream.Write((unsigned char)ID_CREATE_REMOTE_PLAYER);
stream.Write(i);
stream.Write(x);
stream.Write(y);
stream.Write(z);
peer->Send(&stream, HIGH_PRIORITY, RELIABLE, 0, players[i], true);
```

Función StartGame() mandando un mensaje a todos los players para crear el jugador principal y a los competidores. Cada uno con su id única del servidor.

- **ID_GAME_ENDED:** El servidor recibe la información de uno de los jugadores de que la partida ha terminado y hace un broadcast a los demás jugadores de que la partida ha terminado.

```
case ID_GAME_ENDED:
    endGame(packet);
    break;
```

Recepción de la condición en el servidor

```
void ServerManager::endGame(RakNet::Packet* packet)
{
    broadcastData(packet);

    started=false;
}
```

Función en el servidor que realiza el broadcast a todos los jugadores

```
void NetworkManager::remoteEndGame(RakNet::Packet* packet)
{
    setStarted(false);
    Game::getInstance().setState(IGameState::stateType::CLIENTLOBBY);
}
```

Función en el cliente que cambia el estado de la partida y la finaliza

- **ID_CREATE_PLAYER // ID_CREATE_REMOTE_PLAYER:** El primer método posiciona el player principal en la posición que toca, y el segundo crea el player secundario que mimicará el comportamiento de un jugador conectado, posicionándolo en el mapa inicialmente.

```
case ID_CREATE_PLAYER:
    createPlayer(packet);
    break;
case ID_CREATE_REMOTE_PLAYER:
    createRemotePlayer(packet);
    break;
```

Recepción desde el cliente del paquete

- **ID_BOX_COLLISION:** Se ejecuta cuando un jugador ha colisionado con una de las cajas. Transmite a los demás jugadores con un broadcast que la caja no se puede usar.

```
case ID_BOX_COLLISION:
    remoteItemBoxCollision(packet);
    break;
```

Recepción del mensaje desde el lado del cliente

```
case ID_BOX_COLLISION:
    broadcastData(packet);
    break;
```

Paso de la orden a todos los jugadores

```
void NetworkManager::remoteItemBoxCollision(RakNet::Packet* packet)
{
    RakNet::BitStream parser(packet->data, packet->length, false);
    uint16_t id;

    parser.IgnoreBytes(1);
    parser.Read(id);
    ObjectManager::getInstance().getObject(id).get()->getComponent<ItemBoxComponent>().get()->deactivateBox();

}
```

Función que desactiva la caja para los jugadores. La *id* de las cajas se comparte dado que se genera al crear el mapa, por tanto no necesitan de un identificador por parte del servidor.

- **ID_CREATE_BANANA / ID_DELETE_BANANA / ID_CREATE_RED_SHELL / ID_DELETE_RED_SHELL / ID_CREATE_BLUE_SHELL / ID_DELETE_BLUE_SHELL :**

Funciones de creado y destrucción de objetos. Las peticiones de creado o destrucción llegan a través de un evento por parte del event manager. Estas envían la petición con el identificador del servidor del jugador para saber quién ha creado el objeto.

```
void createBananaEvent(EventData eData)
{
    NetworkManager::getInstance().createBanana(eData);
}

void destroyBananaEvent(EventData eData){
    NetworkManager::getInstance().destroyBanana(eData);
}
```

Ejemplo de par creación/destrucción de funciones delegadas. Se llaman al crearse el evento que las invoca.

El Cliente a través de su *NetworkManager* envía la petición al Server de crear los objetos o borrarlos.

1. Cuando deben de crearse, el servidor manda una petición a todos los jugadores, incluso al creador, para que lo creen, pasando en el paquete de datos el identificador del servidor que tendrá el objeto, y el identificador del servidor del jugador que la creó.
2. Cuando deben de destruirse el cliente ejecuta la destrucción del objeto, y pasa su identificador al servidor, que los pasa al resto de jugadores para que lo destruyan.

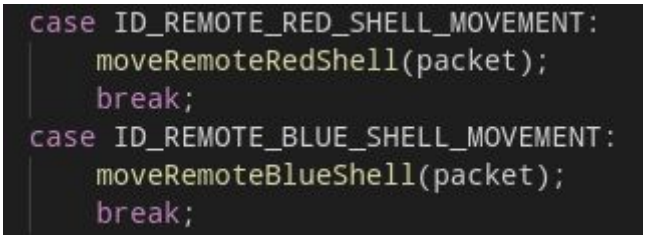
```
case ID_CREATE_BANANA:
    broadcastObject(packet);
    nObjects++;
    std::cout << "Objeto numero "<<nObjects<< " creado." << std::endl;
    break;
case ID_DESTROY_BANANA:
    broadcastData(packet);
    break;
```

Lado del servidor. La función broadcastObject manda la información a TODOS los jugador con el identificador del servidor que tendrá el objeto que se cree.

Los clientes reciben las peticiones del servidor, y generan o destruyen los objetos en sus correspondientes funciones.

- **ID_REMOTE_PLAYER_MOVEMENT // ID_REMOTE_RED_SHELL_MOVEMENT // ID_REMOTE_BLUE_SHELL_MOVEMENT :**

Son funciones encargadas de transmitir la información de movimiento (posición y rotación) de un objeto a todos los demás jugadores. El Cliente tiene unas listas internas de cada tipo de objeto creado por el servidor. Recorre las listas buscando aquellos que ha creado el propio cliente y realiza un broadcast de sus posiciones en el update del NetworkManager.



Ejemplo de llamada desde el lado del Cliente