Proyecto
Lab21

Grupo
Dire Wolf Games

DISEÑO DE REQUERIMIENTO Y FUNCIONES DE RED

Hito:
Fecha entrega: 12-01-2017
Versión: 1.1

Componentes:
- Aarón Colston Avellà Hiles
- Sergio Huertas Ferrández
- Eduardo Ibáñez Frutos
- Marina López Menárguez
- Rubén Moreno Mora
- Rafael Soler Follana

Contenido

1. Introducción

1.1. Propósito

Desarrollar un sistema de control para multijugador en red para aplicarlo en el Videojuego. Hemos elegido el motor de red RakNet para implementar este sistema, el cual posee licencia de uso BSD que permite su uso, modificación y distribución libre manteniendo el copyright de la empresa Oculus VR.

En este documento se pretende detallar las acciones que lleva a cabo el motor, mensajes y la especificación de la API usada.

2. Especificación de la API

2.1. Clase NetGame.

*void open(Scene *scene, bool multiplayer);*

Inicializa los objetos de RakNet necesarios para gestionar las conexiones. Necesario llamarlo antes de utilizar el motor de Red.

- *scene: la escena donde se crearán los objetos de red.*
- *multiplayer: si es multijugador o no.*

void close();

Cierra todas las conexiones y las libera.

void update();

Se encarga de enviar y recibir los mensajes y procesarlos. Se debe llamar periódicamente durante el juego.

*void addNetObject(dwn::DrawableReplica *drawReplica);*

Añade un objeto de tipo DrawableReplica a la lista de objetos sincronizados. Son los Players que se replican en los demás hosts mediante el objeto Replica3 de RakNet.

void addNetEntity(Entity entity);*

Añade un objeto de tipo Entity a la lista de objetos sincronizados por red. Este tipo de objetos son por ejemplo puertas y generadores.

void addNetConsumable(Consumable consumable);*

Añade un objeto de tipo Consumable a la lista de objetos sincronizados por red. Estos objetos son los que pueden ser cogidos por los Players.

void addNetEnemy(Enemy enemy);*

Añade un enemigo a la lista de enemigos que son sincronizados por red.

bool isLocalObject(RakNet::RakNetGUID id);

Devuelve si el objeto indicado mediante el GUID de objeto, es local a la máquina.

bool isServer();

Devuelve si es el primer host conectado al servidor, que realiza las funciones de servidor de partida.

bool isMultiplayer();

Devuelve si es multijugador.

void setMultiplayer(bool m);

Modifica el valor de multijugador.

bool getConnected();

Devuelve si está conectado correctamente al servidor.

bool getConnectionFailed();

Devuelve si no se ha podido conectar al servidor al llamar a connectToServer()

bool getConnectionRejected();

Devuelve si la conexión ha sido rechaza por el servidor al llamar a connectToServer()

bool getGamesSearched();

Devuelve si han sido buscadas ya las partidas. Se buscan al conectar a un servidor con connectToServer().

unsigned short getParticipantOrder();

Devuelve el número de participante en la partida del personaje local.

PlayerMate getPlayerMate(unsigned int i);*

Devuelve el compañero con índice "i" de la partida.

int getNumPlayerMates();

Devuelve el número de compañeros en la partida.

void startGame();

Envía una señal a todas las máquinas de la partida para empezar.

bool getGameStarted();

Devuelve si la partida ha empezado.

std::vector<std::string> getServers();*

Devuelve la lista de los servidores encontrados con searchForServers()

std::vector<std::string> getGamesIP();*

Devuelve la lista de IPs de las partidas encontradas en el servidor.

bool searchForServers();

Busca en la red los servidores disponibles pudiéndose recuperar más tarde con getServers().

- *Devuelve true si ha encontrado servidores.*

bool connectToServer(unsigned int index);

Una vez buscados los servidores con searchForServers(), se llama a esta función para conectar al servidor específico. Después de conectar se pueden consultar getConnected(), getConnectionFailed() y getConnectionRejected().

- *index: El índice del servidor a conectar.*
- *return: true si ha conectado (getConnected() es true), y no ha habido Failed (getConnectionFailed() es false) o Rejected (getConnectionRejected() es false).*

bool connectToGame(unsigned int index);

Conecta a una de las partidas de gamesIP que deben haber sido previamente buscadas (getGamesSearched() es true).

- *index: el índice de la partida en getGameIP() donde conectar.*
- *return: devuelve true si no ha habido ningún error.*

void sendBroadcast(unsigned int messageID, unsigned int value);
void sendBroadcast(unsigned int messageID, RakNet::RakString value);
void sendBroadcast(unsigned int messageID, dwe::vec3f position, float angle);
void sendBroadcast(unsigned int messageID, unsigned int objectID, dwe::vec3f position, dwe::vec3f rotation);.

Envían mensajes a todas las máquinas conectadas en la misma partida y se reciben en update() en los demás ordenadores.

- *messageID: El id de mensaje que se envía.*
- *El resto de parámetros son los datos que se envían junto con el mensaje.*

2.2. Clase DrawableReplica.

virtual const char getNetObjectID() const;*

Esta función hay que sobrescribirla en los objetos heredados para que devuelva el nombre del objeto que estamos creando.

virtual void update(RakNet::TimeMS curTime);

Actualiza las propiedades del objeto con los recibidos remotamente.

3. Mensajes.

- *ID_CONSUMABLE_TAKEN: Indica que un objeto de tipo Consumable ha sido cogido y debe de borrarse en el resto de equipos. Es enviado por la clase Consumable.*
- *ID_DOOR_OPEN, ID_DOOR_CLOSE: Mensaje enviado cuando un objeto de clase Door abre o cierra la puerta.*
- *ID_GENERATOR_ACTIVE: Mensaje enviado por la clase Generator cuando se activa.*
- *ID_SEND_AMMO: Mensaje enviado por la clase Player que indica que un player le ha mandado munición a otro. Se envía como parámetro el GUID del player al que se le pasa la munición. La clase NetGame procesa el mensaje para añadir la munición al player destino.*
- *ID_SEND_MEDKIT: Mensaje enviado por la clase Player que indica que un player le ha dado un botiquín a otro jugador. Funciona de igual manera que ID_SEND_AMMO.*
- *ID_PROJECTILE_CREATE: Mensaje que se envía al disparar para replicar el disparo en los demás equipos. Se envía con los parámetros Posición y la Rotación del eje Y (el ángulo con el que se dispara).*
- *ID_ENEMY_UPDATE: Mensaje enviado por la máquina que sirve la partida para sincronizar la posición de los enemigos en el resto de equipos. Se envían los datos de posición y rotación del enemigo. No se envían en cada actualización de red, sino que en cada actualización se actualiza un enemigo.*
- *ID_GAME_STARTED: Mensaje enviado por la máquina que sirve la partida al resto de máquinas para indicar que el juego ha empezado.*

4. Sincronización.

4.1. Medición del lag.

(pendiente)

4.2. Sincronización de entidades simples.

La clase NetGame mantiene listas de punteros con los objetos que son susceptibles de ser replicados en red (su comportamiento). Estos objetos en sus acciones envían la actualización de la misma mediante un sendBroadcast() al resto de equipos de la partida. Este mensaje es procesado por el método NetGame::update() realizando la acción pertinente en la máquina local.

Clases soportadas en esta versión: Entity y Consumable.

4.3. Sincronización de la IA.

En esta primera versión, con una IA simple, se realiza una sincronización de la posición de los enemigos recorriendo el vector de Enemigos y actualizando posición y rotación, realizando esta actualización en el método NetGame::update() actualizando un enemigo en cada llamada.

4.4. Predicción de movimiento de los jugadores.

(pendiente)