ITP 484 Lab 3
TrojanBlast Networking!

Purpose

The purpose of this lab is to take the single-player, top-down, coin collection game "TrojanBlast" and add support for multiple players on different hosts. This will be done by breaking apart the single player code into a dedicated server executable and a client executable. For this lab, all logic will run on the server, The client will be responsible for gathering user input, sending it to the server, and receiving relevant state updates from the server. This is similar to the networking model of the original Quake. Transport should be handled by UDP.

Specifications

The deliverable for this lab is the code for two separate executables: a dedicated server, and a client. The dedicated server should accept one command line parameter, interpreted as the port number on which to listen. For instance if the command arguments field in the Debug property sheet for the server is

45000

then the server should start up a socket listening on port 45000.

The client should accept two command line parameters, the first of which is the ip and port of the server to which the client should connect, and the second is the name of the client. For instance, if the command arguments field in the Debug property sheet for the client is

192.168.2.103:45000 Joshua

Then the client should attempt to connect to port 45000 at ip 192.168.2.103 and identify itself to the server as Joshua.

To turn in your work, just commit it to svn before the due date- I will grade the last commits you make that are before 11:59 pm on Sunday 3/2 and 11:59 pm on Sunday 3/16.

.

Procedure:

Observe!

Get familiar with the game. Open your solution and set TrojanBlastClient as the startup project. Set its single command line param to be your first name. Run the project. Use wasd to move the ship and collect the Tommy coins. Examine the code to see how things work. Note the following things

You are currently running the TrojanBlastClient, which currently runs all the logic as well as processing the input.

Most of the code is in the TrojanBlast library, which is linked into the TrojanBlastClient and TrojanBlastServer. You will be keeping any shared code in this project and moving client specific code to the client and server specific code to the server.

Examine the GameObject class- it's the base class of all moving objects in game.

Examine the SpriteComponent and the RenderManager. Examine how GameObject's create their SpriteComponents and register them with the RenderManager for rendering.

Examine the InputManager and the Ship classes to see how you control the ships

Examine the ScoreBoardManager class to see how scoring works

Examine the GameObjectRegistry- it's how you can create an object by an identifier, similar to what we discussed in class.

Examine the StringUtils class- it has a nice function for reading a command line argument as a wstring.

Examine the Timing class- that will allow you to get the time in seconds so that you can accurately time when you send packets.

Examine the Engine class and its methods, it ties everything together.

Examine the ClientMain.cpp file. See how it creates an instance of the Client, which is a subclass of the Engine.

PART 1: Splitting into multiple projects. Due Sunday 3/2 at 11:59 pm

In your solution you'll notice there is also a TrojanBlastServer project. Currently, that project doesn't do much- try setting it as your startup project and running it. Nothing happens! If you look at ServerMain.cpp you'll notice the code that creates the server is commented out, as is the StaticInit function in the Server.cpp file. Try commenting these back in and running the server. You'll see that you have a link error and cannot run.

This link error occurs because the d3d11.lib is not linked into the Server. This is proper behavior- it is often useful to be able to run a dedicated server "headless" (with no graphics component) so that you can run multiple server processes at the same time on a single host. Our dedicated server will be no different, so your first task is to get the server to compile and link without linking in the d3d11.lib.
To do this, you'll need to find all code in the shared TrojanBlast library that should only run on the client, and move it into the client. This means:

Any code that references rendering or drawing should only be in the client (e.g. the RenderManager)
Any render resources, like textures, should only be in the client.
And code the reads the user's input should only be in the client.

You'll want to move the relevant files into the client directories, removing them from the shared project and moving them into the client project.

You'll also need to take certain functions from the Engine class and move them into the Client class- for instance, Engine::DoFrame currently calls RenderManager::sInstance->Render(). Since there should be no RenderManager at all on the server, this function will have to be made virtual and implemented differently on the client and server.

Finally, you'll want to make sure that the SpriteComponents used by the RenderManager aren't created on the server. To do this, create 2 new subclasses of Ship- ShipClient and ShipServer. Make sure that only ShipClient creates (and has a member variable for) the SpriteComponent. Then, find where the Engine registers the Ship class to have the id 'SHIP'. Change the code so that on the server it registers the ShipServer class and on the client it registers the ShipClient class. You've now done something really cool- made it so that you can have a persistent object replicated from server to client, but in a way that the client object can have client specific code and the server can have server specific code- all without using messy defines or other hacks. Now do the same for the TommyCoin, as it also has a SpriteComponent.

At the end of this step, you should still have a fully runnable client, but you should also have a server that you can build and run. It doesn't necessarily do anything yet, but it has all the logic necessary without any of the drawing code.

You can test running your client and server at the same time by right clicking on your solution at the top of the solution explorer and setting multiple startup projects.

Make sure you commit this by the due date.

PART 2: Networking. Due 3/16 at 11:59 pm

1) UDPSocket and UDPSocketUtil
Similar to lab 1, create a UDPSocket and UDPSocketUtil to wrap the UDP socket functionality. Specifically create the following functions:

shared_ptr< UDPSocket > UDPSocketUtil::Create(uint16_t inPortNum)

Creates a udp socket and binds it to the given port(pass port 0 on the client to pick a free port)

int UDPSocket::SendTo(const void* inData, uint32_t inLength, const sockaddr& inToAddress)

Sends data to the address- returns how many bytes were sent, or an error code

int UDPSocket::ReceiveFrom(void* outData, uint32_t inLength, sockaddr& outFromAddress)

Receives data up to inLength in size. Returns the number of bytes received, or an error code. Returns the sender's address in outFromAddress.

Make sure all sockets are set to non blocking mode on both the client and server. Remember that UDP is based on a datagram, not streams, so you will only need one socket per host, no matter how many clients there are.

2) create a PacketBuffer class similar to the one discussed in lecture that lets you read from and write to packets as a stream. Add a function to allocate memory for the mBuffer. When creating a packet pick a nice large size but one that's small enough it won't be fragmented on Ethernet. Add the following functions to UDPSocket

int UDPSocket::SendTo(const PacketBuffer* inPacketBuffer, const sockaddr& inToAddress)

This sends the entire packet buffer, up until the mBufferHead. Returns the number of bytes sent, or an error code.

int UDPSocket::ReceiveFrom(PacketBuffer* outPacketBuffer, sockaddr& outFromAddress)

This should receive into the packet, setting the mBufferCapacity to the amount of data received, and the mBufferHead to 0 so it's ready to read. Return any error code.

3) Create a basic protocol for connecting from a client to a server.
Add a shared_ptr< UDPSocket > mSocket member variable to the Client class. When the client starts up, attempt to connect to the server by creating a packet buffer and writing 'HELO' into it, followed by the null terminated client name (read from the command line). Send this packet EVERY SECOND INSIDE YOUR RUN LOOP ON THE CLIENT. UDP is unreliable, so you can't be sure your first packet will get through.

On the server, similarly add an mSocket to the Server class. In the run loop, add a function that checks for a packet on the socket. If there is a packet, see if it begins with HELO. If so, call OutputDebugStringW with the name of the client in the packet (this will print out the name from the client, so you can test to make sure your communication is working)

4) We need to track our clients on the server.
Create a class called ClientProxy. It should store a client's name, their player ID (assigned from an every increasing counter whenever someone connects) and their sockaddr so we can send messages back to them.

Add

vector< ClientProxy > mClientProxies

to the Server class.

Add code to the server so that when it receives a HELO packet with a client name, it creates a ClientProxy to store the name, assigns a Player ID and stores the sockaddr from which the connection came. It should then insert the ClientProxy in the mClientProxies vector. You should also check to make sure that you don't already have a ClientProxy with a matching sockaddr- if so, this is a duplicate packet and you should *not* recreate the client proxy. It will be helpful to make a method in the server that finds a ClientProxy by sockaddr. To check if two sockaddrs are the same, just cast to a sockaddr_in and check if the port and ip address are equal.

Once the proxy is created, send a message back to the client that contains 'WLCM' and then the player ID. On the Client, store the PlayerID in a member variable named mPlayerID on the Client class. Once you've received this welcome packet with a player ID, stop sending your hello packet- you've been acknowledged! Note that on the server if you receive a duplicate 'HELO' packet you don't want to recreate the ClientProxy, but you *do* want to resend the 'WLCM' packet, because the client might not have received it. On the client, if you receive a second 'WLCM' packet, just ignore it- it's the server responding to duplicate packets.

You'd better be done with this by 10/9. If not, you're falling behind!

5) Now we're going to start sending our Input State over the network from client to server. Let's arbitrarily decide that we'll send 30 packets per second to the server- this seems pretty reasonable for our LAN. So, in your Client's run loop, if you've been welcomed already, and if it's been 1/30 of a second since the last time you sent out a packet, create a new packet buffer. We'll give all our different managers a chance to fill the packet with data, and then send it out to the server. Right now we'll start with just the InputManager.

If you look at the InputManager (which hopefully is now only in the Client project, because the server has no need for an input manager) you'll notice that it updates an InputState object whenever the user's input changes. For now, let's just send that object in every packet that goes from the client to the server. Add the following functions to InputState

```
bool Write( PacketBuffer* inPacketBuffer ) const;
bool Read( PacketBuffer* inPacketBuffer );
```

Write should write the InputState's state to the packet buffer and Read should read it out of the buffer. On your client, whenever a packet is going to go out first write the 4 bytes 'TJBC' identifying us as a Trojan blast client packet that's not a HELO packet and then write the current input state into the packet.

On the server, whenever you receive a packet that doesn't start with 'HELO', make sure it starts with 'TJBC' and store the InputState in the appropriate ClientProxy.

6) Now we need something to do with the input state
See how the client creates Ships in the Engine::SetupWorld function? Remove that code so that no ships are created on the client or server by default. While you're at it, make sure the client creates NOTHING in the SetupWorld function- everything should be sent by the server eventually.

Now change it so that a ship IS created on the server right when the Client Proxy is created. Make sure the ship get's the playerID that is assigned to the Client Proxy. Then change the ShipClient's Update method so that it doesn't call Upate or ProcessInput on the client. Change the ShipServer's ProcessInput method to use the InputState from the appropriate ClientProxy. Now the ship will be moving on the server, but you just can't see it- it's not spawned on the client and nothing replicates it over so it just doesn't exist.

7) Now we need to start replicating the ships over to the client. For now, our game state is small enough that we're going to use the most naïve system possible- just

like Quake did, we're going to replicate the entire state of the world in every packet. Let's similarly decide that we're going to send a packet to each client every 1/30 of a second on the server. We'll have to loop through the client proxies to make sure we hit each one. The good news, though, is that since we're sending the whole state, we only have to build up the packet buffer once and can then send it to each of the clients.

When you make a packet on the server, make sure it starts with 'TJBS' to identify it to the client, similarly to how the client identified itself to the server.

Now we have to figure out how to replicate the world state. Add some virtual functions to GameObject for Reading and Writing to a packet buffer, and override them for Ship and TommyCoin, similar to how we added them to InputState. (TommyCoin doesn't have much state to read or write) Also, add a uint32_t mNetworkID to GameObject. Every time the server creates a GameObject, make sure it is assigned an ever increasing mNetworkID.

Now, when you have a packet ready to go, loop through all the GameObjects in the world, and write into the Packet Buffer their network ID, then their four character type (used in the object registery), and then call their Write method. To get their type just call the virtual function GetFourCCName.

When you receive a server packet on the client, loop through, reading network id and type. Look in the world for an object with the given network id- if there isn't one, use the four cc code to create one from the object registry and assign the given network id. Then, whether the object existed or not, read in the latest state for the object. As a final step, since we're sending the state of the whole world every frame, if there's any object that exists on the client that didn't have state sent from the server, call SetDoesWantToDie on it so it will go away. This is inefficient, but thorough and we'll be improving on it in the next lab.

8) Now that ships are moving, make sure that ALL logic is running only on the server. That is, no movement should ever be done by the client, nor should any collisions be processed, etc. Anything that seems like game logic should be run only on the server. Test by connecting three different clients to your server. If possible, try one or more on a different computer.

9) The next task is to replicate over the scoreboard so you can see the names and scores of all the other players. Implement your own protocol for this. You're going to have to prefix the world state with an integer identifying how many GameObjects are included in the packet, so the client know when it's done reading world state and on to reading scoreboard state.

10) Next, we want to handle clients dropping out. If the server doesn't get an input state from a client for over three seconds, it means they've disconnected. Remove

their client proxy, take them off the scoreboard and clean up anything else the server keeps around for them.

11) If you've implemented everything correctly, you should be able to join and drop multiple clients repeatedly and have the correct state present at all times. Test this out and fix anything broken.

That's it! You've now implemented a networking system similar to the original Quake. It's worth noting that although we didn't add sequence numbers, we've implemented a sort of reliability layer on top of UDP by sending all the data in the world in every packet, guaranteeing that unless somebody quits or disconnects, the latest version of the data will eventually arrive for them.

In the next lab we'll be making various improvements and optimizations and adding a more efficient reliability layer similar to the one Tribes implements.