Jeremy Hauschildt
CSU Chico
Department of Computer Science

MULTIPLAYER DANNYKART RACING
A networked multiplayer 3D racing game.
# Table of Contents

Introduction to Project ........................................................................................................ 2  
Introduction to Problem ....................................................................................................... 2  
Description of Solution ........................................................................................................ 3  
What I Learned .................................................................................................................... 7  
Appendix: Links to installers, code repository and instructions .......................................... 8  
Bibliography ......................................................................................................................... 9
Introduction to Project

Multiplayer DannyKart Racing is a multiplayer 3D racing game. It allows multiple people to connect, over a network, and race each other in real time. It consists of a server that can allow multiple races at one time and a client that connects to the server in order to race. When a user starts up the game they can, after selecting their name, select a server to connect to from a list with brief descriptions of each. Once connected to a server they can see a list of races and how many people are in them and join one of these.

DannyKart Racing [1] was originally created last semester by Daniel Phelps [2] as his senior project with the intent that future students could extend his project. It was a single player, race against time, 3D racing game written in C#© [3] using XNA Game Studio© and the JigLibX© physics engine. The goal of my project was to add multiplayer features to this existing game.

Introduction to Problem

At the beginning of this semester when I saw DannyKart Racing and the opportunity to extend it the first thing that came into mind was to add multiplayer features. Single player games without a multiplayer component in this current generation are few and far between and rarely feature much success. I felt that racing would lend itself nicely to these improvements and provide some intractability that most consumers seek in a video game experience. I also, having no experience in networking or game development, thought that this would be a good opportunity for me to learn about these things as they have interested me for a while.

Multiplayer games are nothing new and a few different architectures exist for providing the network play depending on the type of game being created. For a game with real-time interaction, such as a racing game, there were basically two widely used architectures that I could find. One is to have clients send requests to the server for particular actions (accelerate, turn, etc.), have the servers physics engine calculate positions of objects and then broadcast them to all the clients. In this architecture the server is in charge of physics calculations and synchronizes its physics engine with every client who does its own calculations in between updates from the server [4]. The second option is to have the clients keep their own physics engine and broadcast their positions to the server who then updates its game state and updates clients. Both options would need some kind of prediction and lag compensation [5].
Description of Solution

In my midterm documents and design I chose the first architecture (server has physics state) in the beginning. After getting into the implementation stage I started to realize that it might get too complicated with the current design of the game so I ended up switching to the second architecture (clients keep physics state) which fit much better. The first architecture required that you design the game with this pattern in mind. Since DannyKart Racing was designed originally as a single player game far too many changes to its structure would have to be made. I was trying to retain as much of the original structure as possible so the second architecture made much more sense.

I set out to add a new gameplay mode in the client that would support multiplayer without changing the design pattern which was very well laid out. The basic goal was to have each client send their current position, orientations in the 3D space, and some information to predict where the car would go. This would be sent out at a regular interval to the server, who would update its game state with this information and broadcast to all clients in the current game. Each client then draws outside vehicles and continues drawing them in between updates from the server.

The Gameplay class drove the original game and logic. In order to retain this structure I refactored some of its functionality to a base class. I could then add new Gameplay modes (like multiplayer) by just inheriting this class. This can be seen in the class diagram for the client below. The job of my new NetGameplay class is to basically draw the GameState (contains all information from each player and information about the game) every frame and predict and interpolate positions.
You can see on the left that some new menus needed to be added as well as some classes to deal with the networking logic. Networking is done with UDP and a networking library called Lidgren. The Lidgren library allows you to bring some of the reliability of TCP for certain messages. This allowed me to keep track of important messages like server and game information for which speed was not necessary. Things such as position information that are sent rapidly do not need the overhead of TCP as I only want the newest message. It allows the order of unreliable messages to be tracked so that old information can be discarded without the overhead of guaranteeing delivery. It also helps facilitate latency and lag simulation that I used to help ensure my solution worked under both packet loss and (relatively) high latency.

The server was not part of the previous project. As you can see in the object diagram below the server can have multiple game instances at a time. Each game instance is stored in a list to keep track of them. They are also associated with the MpPlayer class in a dictionary (Player is the unique key, ServerGameInst is the value). This is done because of the constant (and slightly faster) lookup time of a hash table. This collection is used to connect a player to a specific game (when they ask to join a game) and for routing messages to a game instance. Each message as it comes into the server is checked to see if it belongs to a game instance or is unconnected. Unconnected messages would be something like a request for a game list or a

---

**Figure 1 - Game Client Class Diagram**
request to join a game. If it is not unconnected it is passed to the specific game instance that the player belongs to. From here it is processed and the game instance can take action and broadcast to its clients. The game instance is basically receiving position messages from clients, updating the GameState accordingly and broadcasting messages at a regular interval as small as possible.

Figure 2 - Game Server Class Diagram
As a quick overview I will show you what happens when a client connects to a server. This can be seen with some detail in the diagram below.

<table>
<thead>
<tr>
<th>[GameProtocol] &lt;data&gt;</th>
<th>Sent By &lt;DataType&gt;</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Connect] Client Name</td>
<td>Client &lt;String&gt;</td>
<td>Client connects to server</td>
</tr>
<tr>
<td>[Connect] 15ms</td>
<td>Server &lt;int&gt;</td>
<td>Server tells client update rate</td>
</tr>
<tr>
<td>[GameList]</td>
<td>Client &lt;&gt;</td>
<td>Client requests a list of games</td>
</tr>
<tr>
<td>[GameList] &lt;GameListObject&gt;</td>
<td>Server &lt;Object&gt;</td>
<td>Server compiles and sends list of games</td>
</tr>
<tr>
<td>[JoinGame]</td>
<td>Game # 12 Client &lt;int&gt;</td>
<td>Server requests to join a game</td>
</tr>
<tr>
<td>[GameState]</td>
<td>&lt;GameStateObject&gt;</td>
<td>Server &lt;Object&gt; Client gets state of current game</td>
</tr>
<tr>
<td>[Start]</td>
<td>Client &lt;&gt;</td>
<td>When client is ready, let server know</td>
</tr>
<tr>
<td>[Start]</td>
<td>Server &lt;&gt;</td>
<td>When all clients are ready server says go</td>
</tr>
<tr>
<td>[Position]</td>
<td>&lt;PositionObject&gt;</td>
<td>Client &lt;Object&gt; Game started, client sends its position</td>
</tr>
<tr>
<td>[PositionState]</td>
<td>&lt;PositionState&gt;</td>
<td>Server &lt;ushort&gt;&lt;Objects&gt; Server sends out positions of all players</td>
</tr>
<tr>
<td>[Position]</td>
<td>&lt;PositionObject&gt;</td>
<td>Client &lt;Object&gt; Client updates its position</td>
</tr>
<tr>
<td>[FinishedRace]</td>
<td>&lt;TimeObject&gt;</td>
<td>Client &lt;Object&gt; Client finished so let server know time</td>
</tr>
<tr>
<td>[FinishedRace]</td>
<td>&lt;2 , TimeObjects&gt;()</td>
<td>Server &lt;int&gt;&lt;Objects&gt; Server sends compiled list of times</td>
</tr>
<tr>
<td>[LeaveGame]</td>
<td>Client &lt;&gt;</td>
<td>Client leaves the game</td>
</tr>
<tr>
<td>[Disconnect]</td>
<td>Client &lt;&gt;</td>
<td>Client disconnects</td>
</tr>
</tbody>
</table>

Figure 3 - Sample client-server interaction

First a client will connect, giving its name. It can then request a game list which is a list of all the games a server has currently available. The client then displays this list and lets the user select one of them, sending a join game command to the server. As soon as the server receives this command the client connection is associated with a game instance. When clients have loaded the level they send a start message to let the server know they are ready. Once the server has start messages from enough clients (currently this is once the game is full) the server sends a start message to all the clients to tell them to begin. All clients count down from 15 seconds and then accept user input. Clients now send position messages that have much less information that the game state. These just contain a position vector, orientation matrix and a velocity vector. The server updates its game state with these packets and then broadcasts the positions to every client. Each client then interpolates vehicles with its last position and current position every time the screen is drawn in order to look smooth. When a client has finished the race their screen displays a waiting message while it waits for everyone to finish the race. Once everyone has finished the race the server sends out a list of race times so that the clients can display the winners. More detailed class diagrams can be found in the rest of my portfolio including some process diagrams and sequence diagrams.
What I Learned

I think I learned a great deal by doing this project. I got to experience several technologies I had never used before and I learned how to extend a rather large code base and design. Before starting this project I had never done any large project with network connectivity (just simple stuff with TCP, nothing using UDP). I quickly learned a lot about how things are sent over a network and about latency and the unreliability of UDP. I had also never done anything with 3D or game development before. This required a lot of understanding of trig and vectors so that I could understand what was going on. Less of a hurdle was C#© and Visual Studio© which I had also never used before. Since its syntax was similar to other languages this was not too much of a problem and I quickly learned some of its awesome features such as function delegates (which kind of replace function pointers) and the debugging powers of Visual Studio©. I also learned how important revision control is even when working by alone. Using Mercurial© (functionally the same as Git) I was able to branch every time I wanted to add another feature or experiment with a network architecture and then merge back in without effecting anything else or completely discard it if it wasn’t sufficient.

Probably one of the most important things I gained experience with through this project was working with an existing code base. Understanding others thoughts, design decisions and deciphering their code is never as easy as your own. Thankfully Daniel Phelps did a good job of organizing his code and documenting so it was easy to understand. Despite how clear it was it was still necessary to figure out what everything was actually doing before I could even begin to start my own design. I had to learn how he was using the physics engine, how the XNA Game Studio© framework worked and how the 3D was implemented. I think this skill set is often overlooked in some of the courses taught here and could be useful out in the industry. It seems very rare that you will start off engineering your own software from scratch as a lot of development is working with past projects and others code.
Appendix: Links to installers, code repository and instructions

As part of a backup of my project I kept a Mercurial© repository on BitBucket© with my code. This repository can be checked out or the source downloaded as a zip. This can be found here:

http://code.jhauschildt.com/dannycart-multiplayer/src

There you can also find a link to an installer for the game and all the needed redistributables:

http://code.jhauschildt.com/dannycart-multiplayer/wiki/Home

Please note that all of these have also been included in my portfolio, this is just an alternate location.

To install Multiplayer DannyKart Racing simply ensure that the XNA 3.1 Redistributable is installed (included in this portfolio and on the site). Next install “DannyKart Installer.msi”. This will install all the files Multiplayer DannyKart Racing needs and put a shortcut on your desktop to both the client and the server. To configure the client (add more servers) edit “C:\Program Files (x86)\DannyKart Racing\DannyKart Racing Multiplayer\DannyKart Client\Release\GameConfig.xml”. To configure the server (add more games, server settings) edit “C:\Program Files (x86)\DannyKart Racing\DannyKart Racing Multiplayer\DannyKart Server\Release\ServerConfig.xml”
Bibliography
   http://unreal.epicgames.com/Network.htm