CMP303/501 Network (Systems for) Game Development

Lab 7: Prediction

(Gaz Robinson), Andrei Boiko and Laith Al-Jobouri
School of Design and Informatics

Introduction

In this lab, you will implement and test several algorithms for position prediction in a real-time network game. This isn't going to involve any sockets programming – we're focusing on prediction algorithms this week.

If you have any questions or problems with this week's practical, ask us on-campus or in the MS Teams channel for "Lab Help". If you have a general question about the module, please use the "General" channel or email us (a.boiko@abertay.ac.uk, l.al-jobouri@abertay.ac.uk).

The scenario

Get Lab_07_Prediction.zip from MLS and unpack it. It contains a Visual Studio project. Check that it compiles and runs.

This is a simulation of a two-player tank game, with a 2D game world where each player controls a tank. Each tank has an (x, y) position within the world, using floats for the coordinates. The example uses SFML (Ver. 2.4.2) for rendering the tanks, so this might be a good start for your coursework.

The game has established a shared idea of time between all the participants: the network simulator has a local timer that measures the elapsed real time since the start of the game in seconds, and the clocks have been synchronised between all the players.

Twice a second, each player communicates their position to the other player by sending them a network message, containing a position and timestamp. These messages arrive somewhat later at the other player's machine owing to network latency. The game program should use prediction to mitigate this.

The simulation starts with the game's clock at 1.5 seconds, and runs until 12s have passed. During this time, the opposing (blue) player is moving in a straight line.

You should look at the following files:

- `TankMessage.h` – this defines the `TankMessage` structure.
- `Tank.h`/`Tank.cpp` – this defines the Tank class, which contains the tank's predicted position, and a collection of `TankMessages` that have been received about it.
- `main.cpp` – this contains the game's main loop, which executes once for each display frame in the game. It processes incoming messages, asks the Tanks to predict their current position, and updates the display.

(You don't need to understand or modify the `NetworkSimulator` class – this just simulates the network communication and the behaviour of the other tank.)

Each tank also has a semi-transparent “ghost” that the main loop sets to the current predicted position. This helps illustrate the difference between the last “real” position from the network (solid colour) and the programs predicted position (transparent). You can see it stuck at (-1, -1) when you run the game.

If you’re tired of watching the simulation play out every time, you can modify the “`gameSpeed`” variable in `main.cpp` to increase the speed of the simulation.

Implementing prediction

If you run the program, it'll show the predicted position for the opposing tank at each timestep, along with the messages received from the other player. The predicted positions are wrong, because the `Tank::RunPrediction` method is currently incomplete.

No model

Implement `RunPrediction` so that it updates the tank's predicted position using no model (just the latest message received from the network).

When you run the program, you can compare the predicted positions from your code to the messages you receive about the real position – look at the message you receive later about the position at 10.5s, versus the prediction that your code gave when the local time was 10.5s.

Linear

Implement `RunPrediction` so that it uses a linear model (just velocity). This should result in accurate predictions, where the ghost tank moves slowly ahead of the last “real” position.

Quadratic

Comment the linear code out for now, or build a separate function, and implement `RunPrediction` using a quadratic model (velocity and acceleration – remember the equations of motion).

The predictions should be... remarkably similar to the linear model.

Discontinuities

Modify the test at the start of the main loop so that the game runs until 18s of game time have passed, rather than 12s. The blue tank turns a corner at 12.7s and starts heading in a different direction; if you run the program, you should be able to see this from the messages received.

Try both the linear and quadratic models above, and look at how bad is the discontinuity created when the opposing player's direction changes is. The linear model should settle down fairly quickly, whereas the quadratic model has a much bigger discontinuity. In a real game, we would probably use linear prediction, and smooth out the discontinuity using interpolation.

Try modifying the `sendRate` and `latency` variables in the `main.cpp` and see how this affects the methods of prediction.

If you’ve got time left over…

- It's not necessary to keep all the messages you've received, just the last N needed for prediction; make add_message throw away any messages that aren't needed any more. It'd also be nice if it kept them in sorted order, in case messages are received out of order.
- Implement simple interpolation. Add another vector to the Tank class that keeps a history of the predicted positions (you can reuse the `TankMessage` struct for this). Make a linear prediction forward from this history to the current time, then use a point halfway between that prediction and your prediction based on received messages as your current prediction. How effective is this?
- Network it! The current program uses a fake network. Using WinSock or SFML’s Networking API, why not look into making this example work across a network through Peer-to-peer or a dedicated server/client model. Start from here if you want to use SFML: https://www.sfml-dev.org/tutorials/2.5/network-socket.php