Assignment #5: Client-Side Prediction

Goal

In this assignment, you will implement a simple client-move refinement for optimistic replication, measure queue utilization in several scenarios, and implement one of several options for improvements.

You will begin with a provided project, rather than your Assignment 4 submission. This project does not include a Node/WSL component, and the C++ REST SDK components of the last assignment are not present. We will return to that implementation in Assignment 6.

Tasks

You will begin with the Initial project provided with this assignment. You will submit the same project, with your code improvements/additions. The project is essentially the same as the project used in the replication labs, so the skills and familiarity you developed there will help you here. For example, you should continue to use Visual Studio to configure the solution for "multi-project startup", as that will be very convenient when developing with this code.

This initial code has several improvements over past versions of the testbed:

- dt (time since last update) is calculated accurately for each scenario using QueryPerformanceCounters.
- dt is passed between host and client to aid in prediction/rewind.
- The snapshot control is limited to the [0,1] range.
- In the optimistic scenario, the player is now controllable via WASD.
- In the optimistic scenario, the client attacks by clicking the mouse. The full "confirmation" implementation from the Server-Side Rewind lab is present, with minor fixes.

There are three major tasks in this assignment, which all focus on the Optimistic scenario.

1) (65 points) Implement the TODOs in the OptimisticHost/ClientScenarioState files to improve the quality of the WASD movement, even on low latency connections.
    a. Currently, the client is sending control data to the host, which has authority over movement. However, to keep the movement smooth and responsive on high-latency connections, the client also applies the control data locally. It respects the host authority by resetting the position to the authoritiative position whenever one arrives.
    b. However, under high-latency conditions, there are numerous prediction errors, leading to "jerky" movement, especially as the player changes direction frequently.
    c. In order to resolve this issue, you will implement a client-side movement history record, and then apply the movement history since the last confirmed local-frame in the latest update from the host. This will minimize the feeling/appearance of the "jerky" movement.
    d. Examine the TODOs in the Optimistic files. They are nearly 1:1 with the code changes you will need to make; if you're making more extensive changes, you may be off-track.

   e. Note the testing scenario below, under Testing, carefully! This process will also be demonstrated in class, and it incorporates both Clumsy and the host-side send interval.
   f. Note: you can read more about this concept in our textbook, Multiplayer Game Programming, in chapter 8 "Client Move Prediction and Replay" (pp 243 - 247 – though our code is implemented quite a bit differently. The TODOs are the key!

2) (15 points – three 5-point questions) Once you have completed task 1, fill out the first three questions of CS261_Assignment5.txt, which is included in the Initial project and will be part of your submission.
   a. You will need to add a little code to measure the number of history records actually used, and track the maximum used. You should cout the maximum value when it changes.
   b. You will evaluate the maximum number of history records utilized under several different scenarios.
   c. You should not modify the actual constant in code – all you need to do is measure and include the data in your .txt file.
   d. You should remove the cout output (as well as any other debugging output you added) before you submit!
3) (20 points) You have the option to complete **one of two** possible improvements to the replication testbed:
   a. Option A: Implement a smoother version of the Snapshot Control, so the host will interpolate the snapshots in a more natural way.
      i. You may interpolate over more snapshots, or you may send additional/different data, such as angular velocity.
      ii. You may not dead-reckon off of velocities (the DeadReckoningControl is present in code but not used in this assignment). You must focus on interpolation between snapshots.
      iii. You cannot break the server-side rewind implementation! You may need to change how the sync ratio is calculated and shared for that purpose.
      iv. You may not simply send more updates! You ***cannot*** change the code that implements the send interval in the host.
   b. Option B: Implement robust network buffering/confirmation/resending for dropped packets from the client to the host.
      i. If you use Clumsy to simulate packets dropped from the client to the server, then you will see major prediction errors in movement, depending on the prevalence of the drop. It's possible to miss attacks, but that's harder to reproduce since it occurs in a single packet.
      ii. You would need to implement a buffer for past packets, and a system for the host to confirm the client's previous frames, and resend the missing data.
      iii. The goal is to achieve similar results under 10%, or even 30%, packet drop.
      iv. This scenario will be tested in low-latency conditions – 0ms send interval, no Clumsy latency.
      v. Once you are resending the data, you will need to handle the fact that the data is arriving out of order on the host. You could have the client resend all

subsequent packets, but you will get better performance out of buffering the "newer" updates.

vi. You can read more about this subject in the textbook, Multiplayer Game Programming, in chapter 7 – particularly "Packet Delivery Notification" (pp 209 - 221). The code samples are for their particular demo project, which is quite different from ours. You can't use their code directly, but the concepts should be similar.

c. Option A is more mathematically complex, while option B is more straightforward but likely involves a bit more code.

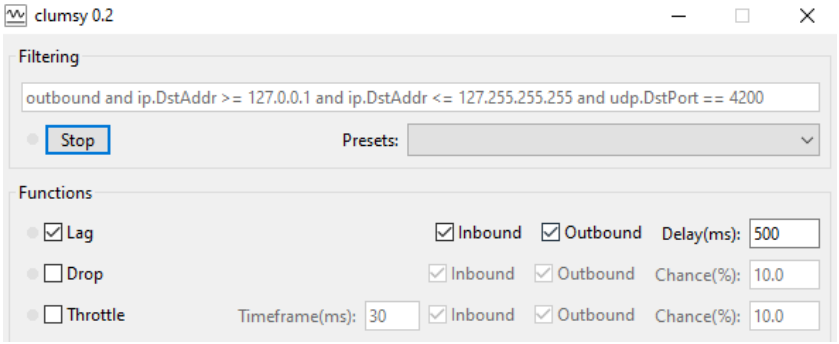
d. If you complete option A, you **must** fill out question 4 in CS261_Assignment5.txt, or question 5 for option B. If you don't briefly describe your implementation in the .txt file, you will not receive any credit, even if you did the work!

You may be tempted to make other changes to the C++ codebase – improvements or other refactoring. **You must not do so – you may only implement/modify code relevant to the changes described above.**

Testing

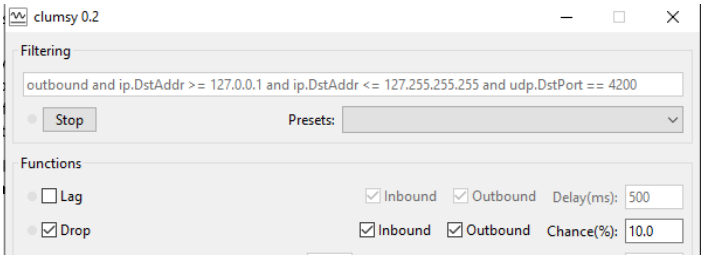
For the first task, your code will be tested under a **combination** of **both** latency conditions:

1) 500ms send intervals from the host, configured by pressing W several times in the host (per our labs).
2) Enabling 500ms latency with a UDP destination-port of 4200 in Clumsy:



For the third task:

- Option A: Your smoothing will be tested at 200ms and 400ms send intervals on the host, qualitatively judging a "reasonable" smoothness by judging if it's similar to a 0ms send interval.
- Option B: your code will also be tested by enabling 10%, and then 30%, packet drop in Clumsy:



Submission

Your submission should be named **yourAlias-CS261-5.zip**, where yourAlias is the username you use to log into your email and lab PC.

The submission should be exactly like the Initial project, without any build output, etc.

- Assets *(exactly as provided in Initial)*
- CProcessing *(exactly as provided in Initial)*
- CS261_Assignment5
  - *No Debug or Release directories in here!*
- CS261_Assignment5_Client
  - *No Debug or Release directories in here!*
- CS261_Assignment5_Server
  - *No Debug or Release directories in here!*
- CS261_Assignment5.sln
- CS261_Assignment5.txt **(with your answers!)**
- *No Debug or Release directories out here, either!*

Note that you should not include any built binaries (Debug or Release directories, at any level), nor any .git or .vs directories. **The submission filesize should be around 1.9 MB, just like the initial zip.**

Rubric

NOTE: If your code does not build under Debug and/or Release configurations, crashes, etc., you will likely receive a 0 for the assignment.

65 Points: Smooth movement under high latency conditions

Per aboveThe TODOs are implemented appropriately to implement smooth movement under high latency, per testing above. Failure to achieve these results, as demonstrated in class, will result in partial credit.

15 points: Three Measurement Questions (CS261_Assignment.txt)

Per above, you should correctly answer the first three questions in CS261_Assignment.txt, replacing "<ANSWER N>" with a number. You will receive 5 points for each correct answer.

20 points: Smooth Snapshot OR Resend on Drop

Per above, you may choose to either implement a smoother snapshot implementation, or to handle packet-drop with equivalent experiences.

- Option A: Snapshot Smoothing
  - If your snapshot looks reasonably smooth at a send interval of 200ms, you will receive 10 points.
  - If your snapshot looks reasonably smooth at a send interval of 400ms, you will receive the full 20 points.
- Option B: Packet Drop
  - If your WASD movement feels reasonably smooth at 10% packet drop, you will receive 10 points.

    - If your WASD movement feels reasonably smooth at 30% packet drop, you will receive the full 20 points.

Partial credit may be awarded depending on the situation.

Important notes:

- **You must briefly describe your implementation in CS261_Assignment5.txt, or you will receive no credit, even if you did the work.** This should include a description of the level you believe you have achieved.
- **You will not get credit for both (you cannot get 120% on the assignment).** If you do both, the grader will pick which one to grade… there are better ways to spend your time!

Submission Penalties

In addition to the grading rubric above, you can receive these penalties if your submission is not correct:

- -5 if your submission files are not named as above.
- Up to -10 if your submission includes additional files, depending on the severity
- Up to -30 if other changes are made to the C++ codebase and projects beyond the specified TODOs, depending on the severity.

Other penalties may apply if needed.

Technical Notes

- This code is not implemented using engine and engineering patterns that you would expect to find in a real game, down to including too much code in the Update functions. The code has been intentionally condensed and simplified for clarity, and to focus our efforts on the relevant work.
- You are welcome to use code (like the high-performance clock) in other classes and projects, with the permission of your instructor. Please do not remove my copyright.