IJHSR
ijhighschoolresearch.org
REVIEW ARTICLE

Simulating Physics in Single and Networked Multiplayer Games

Bhavya Babbellapati

Mission San Jose High School, 41717 Palm Ave, Fremont, CA, 94539 USA; bhavya.babb@gmail.com

ABSTRACT: Simulating realistic physics in video games often involves mathematical approximations to optimize performance. Limited computational power forces game developers to simplify physics simulations, as real-time updates require many calculations each frame. In the case of networked multiplayer games, the physical limitations of data transmission introduce additional performance-degrading factors like network lag. This paper analyzes common numerical methods for single-player game physics, including Euler's methods and Verlet integration, highlighted for their widespread use and illustrative trade-offs in accuracy and computational efficiency. A subsequent section discusses techniques employed in network-based multiplayer games and how game developers overcome data transmission limitations. These techniques are demonstrated through simulations to explain different lag compensation mechanisms. Finally, we discuss the results and the game contexts where these techniques are applicable.

KEYWORDS: Embedded Systems, Networking and Data Communications, Multiplayer, Numerical Game Simulation.

Introduction

Realistic physics in video games makes interactions feel natural and believable, reducing inconsistencies that could disrupt gameplay. The effectiveness of video game animation hinges on smooth renditions of visuals. This rendition rate for a human visual system is between 30 and 60 frames per second.[1] All game animations must be computed, composed, and rendered in a frame interval. Even with advances in display systems, game developers tend to target a wide range of computing platforms with different capabilities. This stringent time frame necessitates the utilization of computational optimizations and approximations in single-player games. To balance performance, game developers and physics engine developers often prioritize certain aspects, sometimes at the expense of realistic physics. Network-based multiplayer games create additional challenges because of the physical limitations of the data transmission and additional queueing delays imposed by the data networks.[2] Other sources of delay can arise from wireless connections and delay from peripheral interfaces like keyboards and mice. In addition to rendering challenges, the integrity and correctness of the game come into play. This paper surveys various techniques for resolving the identified issues and demonstrates different scenarios through simulations. It discusses where some of these techniques are employed and how game developers tend to work around the limitations of network physics.

Kinematics and Numerical Techniques

In physics, we encounter problems in kinematics that compute a final position at the end of an interval. In video games and simulations, game inputs are sampled periodically, and simulations run in repeated intervals, giving the impression of continuous motion updates. Closed-form/Analytical solutions exist for the most basic situations. Advanced physical phenomena need solutions to complex integrals for which it is extremely hard to arrive at a closed-form solution. **Numerical Integration** is a fundamental technique used in game physics to simulate the motion of objects over time.[3] It allows game developers to approximate solutions to differential equations that describe the physical laws governing the game objects. These techniques help us simulate the continuous behavior of the objects using discrete steps. Games utilize these techniques in small time steps to compute velocity, acceleration, and position. They effectively predict what happens at the end of every time step, generating an impression of continuous motion. A typical time step for 30 frames/second is 33ms (1/30th of a second). In practice, the time step used for computation aligns with the refresh interval of the game rendering.[3]

Given the use of time steps, consider the fundamental 1D kinematics equations:

$$v_{n+1} = v_n + a_n \Delta t$$
$$s_{n+1} = s_n + v_n \Delta t$$

Here, v_{n+1} is the velocity of the object in the $(n + 1)th$ frame, v_n is the velocity in the previous frame (nth frame), a_n is the acceleration of the object in the nth frame, and Δt is the time step between the frames. Similarly, s_{n+1} and s_n denote the displacements in the corresponding frames. We know from our first course in calculus that acceleration = dv/dt (rate of change of velocity in an interval) and velocity = ds/dt (rate of change of displacement in the interval). Analytically computing these would involve finding derivatives of these functions. Numerical integration takes an iterative approach by computing these variables repeatedly in very small intervals.

DOI: 10.36838/v8i1.56

Euler's Methods:

The above set of numerical calculations is called **Explicit Euler's integration**.[4] It gives a simple set of equations that allows us to compute velocity and position for each displayed frame. This method is computationally inexpensive, as output variables are calculated in a straightforward manner. It gives accurate results as long as there are no significant variations in the variables in a short span of time. A C programming code snippet for Explicit Euler's integration is shown below.

```
#include <stdio.h>
float t = 0.0;
float dt = 0.033; // timestep
float velocity = 0.0f; // initial velocity
float displacement = 0.0f; // initial displacement
float acceleration = 10.0f;
int main(int argc, char** argv)
{
   // compute for a journey time of 5 seconds
   while ( t <= 5.0 )
   {
      printf("t = %f, velocity = %f, displacement = %f\n", t, velocity, displacement);
      velocity = velocity + acceleration * dt;
      displacement = displacement + velocity * dt;
      t += dt;
   }
}
```

Figure 1: Euler's explicit method. Here is a simple snippet of code demonstrating how Euler's Explicit Integration works on computers. It uses a timestep of 5 seconds to compute velocity and displacement at each frame.

The computation in Figure 1 is discrete in nature, and the rendered movements may look jerky depending on how frequently the velocity and displacement variables are updated and the visuals are rendered. Below is a sample output for two different values of timestep **(dt)**.

```
t = 0.000000, velocity = 0.000000, displacement = 0.000000
t = 0.500000, velocity = 5.000000, displacement = 2.500000
t = 1.000000, velocity = 10.000000, displacement = 7.500000
t = 1.500000, velocity = 15.000000, displacement = 15.000000
t = 2.000000, velocity = 20.000000, displacement = 25.000000
t = 2.500000, velocity = 25.000000, displacement = 37.500000
t = 3.000000, velocity = 30.000000, displacement = 52.500000
t = 3.500000, velocity = 35.000000, displacement = 70.000000
t = 4.000000, velocity = 40.000000, displacement = 90.000000
t = 4.500000, velocity = 45.000000, displacement = 112.500000
t = 5.000000, velocity = 50.000000, displacement = 137.500000
```

Figure 2: Sample output for timestep (dt)= 0.5 seconds. These are the results when running the loop in Figure 1 for a timestep of 0.5 seconds. At 3 seconds, the velocity is 30 m/s and the displacement is 52.5 m.

```
t = 0.000000, velocity = 0.000000, displacement = 0.000000
t = 0.300000, velocity = 3.000000, displacement = 0.900000
t = 0.600000, velocity = 6.000000, displacement = 2.700000
t = 0.900000, velocity = 9.000000, displacement = 5.400000
t = 1.200000, velocity = 12.000000, displacement = 9.000000
t = 1.500000, velocity = 15.000000, displacement = 13.500000
t = 1.800000, velocity = 18.000000, displacement = 18.900000
t = 2.100000, velocity = 21.000000, displacement = 25.200001
t = 2.400000, velocity = 24.000000, displacement = 32.400002
t = 2.700000, velocity = 27.000000, displacement = 40.500000
t = 3.000000, velocity = 30.000000, displacement = 49.500000
t = 3.300000, velocity = 33.000000, displacement = 59.400002
t = 3.600000, velocity = 36.000000, displacement = 70.200005
t = 3.900000, velocity = 39.000000, displacement = 81.900009
t = 4.200000, velocity = 42.000000, displacement = 94.500008
t = 4.500000, velocity = 45.000000, displacement = 108.000008
t = 4.800000, velocity = 48.000000, displacement = 122.400009
...
```

Figure 3: Sample output for timestep (dt)= 0.3 seconds. These are the results when running the loop in Figure 1 for a timestep of 0.3 seconds. At 3 seconds, the velocity is 30 m/s and the displacement is 49.5 m.

We have more intermediate velocity and displacement values if we decrease our integration interval (**dt**) for a given journey. We can observe (from the figures above) that as **dt** decreases (from 0.5s to 0.3s) for a given timestamp, the computed velocity remains the same while the displacement drifts. Our computation for displacement is an approximation that assumes velocity is constant over **dt**. In reality, velocity changes over the interval **dt** as acceleration is not zero.

Timestep (dt) (s)	Timestamp (s)	Velocity (m/s)	Displacement (m)
0.500	3.00	3.00	52.00
0.300	3.00	3.00	49.00
0.100	3.00	3.00	46.00
0.033	3.00	3.00	45.58

Figure 4: Euler's Explicit Displacements (for t = 3s). The summary of the highlighted data in Figures 2 and 3 are presented in this table. It displays velocity and displacement at t=3s for 4 different timesteps.

The closed-form value from kinematics is:

$$S = v_0 t + 1/2at^2 = 0.5 * 10 * 3 * 3 = 45m \text{ for } (v_0 = 0, \Delta t = 3s, a = 10m/s^2)$$

As seen in Figure 4, if we decrease **dt** to a much smaller value, our computation approaches the expected value (45m). Running the computations for a very long time accumulates significant errors, especially at higher values of **dt**. A very low value of **dt** is desirable, but it makes the computations prohibitively expensive and is rarely used in current physics engines. When acceleration is no longer a constant, Euler's Explicit method fails again, as it does not account for another varying value over time.

Euler's implicit integration method takes a different approach to dealing with this issue. It uses the first derivative and evaluates it at the next time step. The following equations include the necessary changes.

$$v_{n+1} = v_n + a_{n+1}\,\Delta t$$
$$s_{n+1} = s_n + v_{n+1}\,\Delta t$$

These equations rely on knowing the future value of the acceleration (i.e., a_{n+1}). Approximating a future value could be done with mathematical techniques. However, these equations are costly and prohibitive for a game engine that is responsible for many updates over the frame interval.[4] As a result, even though the Implicit Euler Integration method gives more accurate results, it is not widely used in game simulation. A hybrid and practical approach to the problem comes from **Euler's semi-implicit integration**.

$$v_{n+1} = v_n + a_n\,\Delta t$$
$$s_{n+1} = s_n + v_{n+1}\,\Delta t$$

It computes the acceleration at the current *timestep* and velocity in the subsequent time step. This method provides a computationally easy integration with fewer errors than the explicit method. Euler's semi-implicit method that uses the nth frame's acceleration to calculate the $(n + 1)th$ frame's velocity, which is then used to compute the object's new position. This eliminates the computationally expensive part (calculating a_{n+1}) of Euler's implicit. It also minimizes Euler's explicit integration inaccuracies by using v_{n+1} instead of v_n to compute position.

Even Euler's semi-implicit method can lead to error because we use a rounded value in each timestep.

More specifically, we use our value of a_n to calculate v_{n+1} and use this value once again to calculate s_{n+1}. Therefore, after many iterations, the error produced could still deviate from the true value, posing a problem for those who want to code extremely accurate simulations.[4] One second-order method that

computes velocity differently is called **Verlet Integration**. Instead of Euler's integration techniques that find velocity and position, Verlet's method finds position straight from the acceleration.[5] Computing acceleration needs a second derivative. Starting from Euler's equations, we can arrive at

$$s_{n+1} = s_n + v_n \Delta t + 1/2 a_n t^2$$

The velocity could be computed from:

$$v_n = (s_n - s_{n-1}) / \Delta t$$

The equations could be easily combined to obtain:

$$s_{n+1} = 2s_n - s_{n-1} + 1/2 a_n t^2$$

Although this algorithm is straightforward and has low error, we must use further approximations to find velocity. One advantage of Verlet integration is reversibility.[5] We could compute positions and velocities in reverse, which could be useful for game replays.

Euler's semi-implicit method performs fine if acceleration is constant in a given timestep. However, higher-order methods may yield even more accurate calculations that are rarely necessary in video game simulations. Games like Grand Theft Auto, Red Dead Redemption, and The Witcher series use physics engines that often employ Euler's method for simulating various physical phenomena.

Collisions and Approximations

Collisions are key elements in many video games. Some examples of collisions are when a game object or character collides with a surface or terrain, a bullet collides with an object, or a character bumps into a wall. A collision is declared when two bodies intersect or the distance between them falls below a certain threshold.

Game characters and objects are modeled using simple geometric shapes called **bounding volumes**. These shapes approximate the object's actual geometry, making collision detection more efficient. Some common models are:

1. Axis Aligned Bounding Box (AABB) - rectangular box aligned with the world axis.[6]

2. Sphere - sphere in which the object is assumed to be enclosed. Efficient for collision checks, but overestimates the shape of the object.

3. Oriented Bounding Box (OBB) - rectangular box aligned with the object's orientation. It is a more complex, but also accurate method.[6]

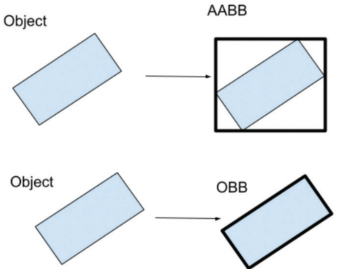

Figure 5: Bounding boxes. This diagram simplifies how AABB and OBB bounding boxes work, using a rectangle as the object. With AABB, the bounding box is aligned with the x and y axes, overestimating the area bounded. The OBB aligns the box to the rectangle's tilted orientation and therefore perfectly matches its shape.

Simple geometries and numerical methods introduce errors in detecting collisions. It is common to experience incorrect collisions in games. Sometimes, collisions (hits) are registered when we feel there is no actual contact; other times, collisions are registered at a slightly off location. Dealing with a complex geometric shape is a challenging task as well. For example, when a player encounters a rugged wall, it is costly and often unnecessary to create a geometrically complex boundary suited for it. This is one instance where a game developer might use rectangles to approximate this boundary. In a video game, one might see this as "glitching" or being able to walk through a wall in certain areas. Floating-point approximations could also cause this.

Collisions in games are dealt with in two phases: **collision detection** and **collision resolution**. Collision detection involves algorithms to check whether any two objects in a frame have collided. When the number of objects increases, it becomes computationally intense $O(n^2)$. For this reason, continuous collision detection is usually reserved for simulations that require highly accurate physics.[7,8] Most game engines perform a two-phase detection, with the broad phase shortlisting the potentially colliding bodies and the *narrow* phase computing the points of collisions of the bodies in question.[7]

Collision resolution in video game physics determines how objects in a virtual world react to a collision. It could involve repositioning objects and changing their velocities. When two objects collide, the system must apply constraint-based methods and rebound forces on them. Sometimes, this introduces the problem of adding energy to the system, causing many physical inaccuracies.[8] One example is when a stationary stack of blocks collapses on itself because of the continuous rebound effects applied to these objects.[8] It is much harder to apply dynamic equations and render a visually pleasing game with limited computing resources. Much like the integration techniques explained previously, collision detection techniques get complex quickly as we approach a realistic outcome.

Fluid Motion

Fluids are often difficult to simulate in video games because they constantly change shape and flow, unlike rigid objects. One way to approach simulating realistic fluids in physics is by treating them as a system of particles. Each particle is controlled by an algorithm that calculates its velocity, position, and its interactions with other particles.[9] However, a high computational ability is required to maintain the physical accuracy of these methods. Another approach to this problem is to treat the fluid as a grid of cells and use each cell to store the properties previously calculated by the particle algorithm. By applying fundamental equations (like the Navier-Stokes equation) to the cells, the system can handle interactions and behaviors of fluids.[9] Sprites (2D animations) are commonly used for large-scale simulations like oceans/water surfaces. Getting a realistic effect is a challenge when dealing with the rendering of fluids.

Network Physics and Multiplayer Games:

On modern-day networks with fiber optic cables, data transmission happens incredibly fast, almost at the speed of

DOI: 10.36838/v8i1.56

light. However, the electrical signals that carry data undergo attenuation, experience propagation delay, and may experience interference when traveling over long distances. These factors constrain how fast data can be reliably transmitted over a network. **Network latency (lag)** in games is the time to send a user's input to a remote server and receive a response.[10] In a multiplayer game over a network, latency poses a considerable challenge for conducting smooth gameplay. If a player on the West Coast of the United States interacts with a server on the East Coast, there is a theoretical minimum latency of 25-30ms, but more like 40ms in the best case. It is also important to note that additional latencies arise from device performance, network congestion, wireless networks, security protocols, and network protocols such as routing.[1]

- ***Multiplayer Games and Authoritative Server:***

Authoritative servers arbitrate gameplay among multiple players. They are essential for maintaining fairness, consistency, and security in multiplayer games. They serve as the single source of truth, ensuring all players experience an identical game world. By validating player inputs and enforcing game rules, servers prevent cheating and provide a level playing field. For instance, a server can prevent a player with a game mod that could set a car's speed to an unrealistic level, maintaining the integrity of the game.

- ***Simulating Client, Server, and Network Lag:***

A simulation is developed in JavaScript and HTML to demonstrate the effects of network lag between game clients and the server. The core simulation consists of one or more client instances (running in their own threads). For the sake of simplicity, the round-trip network latency is configured as a property on the client (client_network_lag). The server component runs in its own thread. A shared buffer is used to communicate the input state from the client to the server. Client enqueues inputs to the shared buffer with a *msg_process_time* equal to the *current time* + *client_network_lag*. The server dequeues messages from the shared buffer when the current time is greater than or equal to the *msg_process_time*. The server is designed to process inputs and send updates periodically at a configurable refresh rate. The pseudo-code in Figures 6 and 7 summarizes the client and server loops for a game that involves firing a cannon in the air.

```
Client Loop
{
        Initialize velocity and firing angle of the cannon.
        do {
                Compute (x,y) position of the cannon
                msg_process_time = cur_timestamp+client_network_lag
                Enqueue  {msg_process_time,  position(x,y)} to the shared buffer
                Check for server world updates
                Render world
                sleep(x)
        } until end
}
```

Figure 6: Simulating the client loop. This figure highlights the process that the client executes before rendering the frame. The client loop computes position and a process time (for simulated network connection). It enqueues these two inputs to the shared buffer with the server. If the server sends out a word state, it will render it.

```
Server Loop
{
        do {
                For all clients {
                        Check for new inputs in the shared buffer
                        Process inputs if it is time ( if cur_time >= process_time)
                        Validate inputs
                        Enqueue world state to all clients
                        Sleep until next refresh time
                }
        }until end
}
```

Figure 7: Simulating the server loop. This figure demonstrates the process the authoritative server executes. The server loop checks the shared buffer for new inputs and processes them if the process time has passed. It validates the inputs and sends the world state to all clients.

The simulated client and server acting in lockstep is a very naive implementation and is rarely used by game developers. Here, the client sends inputs to the server and waits for it to update its state. This involves a round-trip delay to the server before the client renders its new state. We will analyze the rendering from both the client's and the server's point of view. In reality, the server is not in the business of rendering. We catch a glimpse of the server state through hypothetical server screenshots. These simulations are repeated for both turn-based and multiplayer racing games.

- ***Multiplayer Turn - based Game:***

Consider the case of a two-player turn-based game like Scrabble or Darts. An authoritative server maintains the game state. In this scenario, each player interacts with a remote server in the following manner:

- The players send inputs to the server
- The server receives inputs, validates them, and sends them back to all the players
- The players render the game world after receiving the updates from the server.

Let us simulate this by considering the case of players firing a dart (cannon) in projectile motion, attempting to hit a target one after another. The figure below is a simulation of the client and the server receiving state packets over the network.

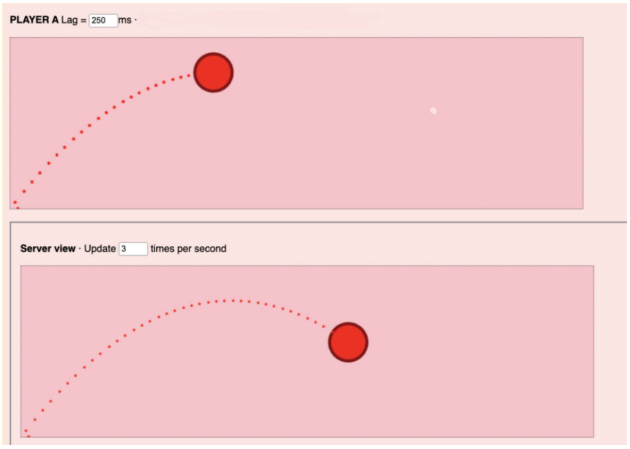

Figure 8: Visualizing state update arrivals on client and server. Here, both Client and Server are in lockstep synchronization, characterized by the server rendering faster than the client. Lag is set at 250 ms for Player A, and the server refresh rate is set to 3 updates per second.

The client and the server trace the trajectories of the cannonball. Each dot in the figure represents the arrival of a state update packet containing the (x,y) position of the object. The

gaps (spacing) between the dots correspond to the latency as the receiving entity processes the data. The figure shows that the spacing between the client's dots is almost identical to the server's. However, the client's state is updated upon receiving the world state from the authoritative server after some network delay.

The simulation depicts a couple of important points:

• A constant lag value simulates the reception and processing in perfect periodic intervals.

• The client is behind the server in terms of updating the state. In other words, the client follows the server, and the network latency between the client and the server governs the rendering experience.

Figure 9: Simulating random latency (10 -500ms). This is the server view for a client with a random lag value set. Each dot represents a packet arrival. The nonnon-uniform spacing of the dots is characteristic of a non -uniform network latency.

In reality, network lag is never constant in magnitude. Let's simulate applying a random latency between 10 and 500 ms. As shown in Figure 9, the time interval between the network packet arrivals is no longer uniform. If the client renders the state immediately upon the arrival of state update packets, the non-uniform spacing of arrivals could result in infrequent and jittery rendering, leading to a poor player experience.

- ***Dejittering Buffer:***

A common solution to this problem is a de-jittering buffer on the receiving end. A de-jittering (delay) buffer absorbs variation in network packet arrivals. This means that if the packet arrives at the client at time t0 and the duration of the de-jittering buffer is, say, d ms (de-jittering delay), the packet will be processed after $t_0 + d$ ms. This smooths the game experience as updates could be delayed and rendered constantly, even if their arrival has variable delays. The side effect of this technique is that the de-jittering of the delay offsets the client's rendering of the game. A decent de-jittering buffer would be useful in scenarios where smooth playback is essential. For example, while watching a TV show on an on-demand streaming service, the playability of the video depends on how frequently the frames are processed and rendered. If the media packets are rendered aggressively as they come in, there is a possibility of running out of them at times (underflows) when there are network hiccups. However, a very minimal de-jittering buffer is advised in a scenario where getting real-time updates is crucial, like a first-person shooter game. Such a game is not playable with significant network delays. A delay buffer of 500 ms is reasonable for turn-based games.

Multiplayer Racing Game

Let us now consider a simple case of a multiplayer racing game in which two players are connected to the authoritative server and synchronized periodically on a server refresh interval (in a lockstep manner). The simulation is now adjusted to demonstrate the client and server views as the game progresses.

Setup:

Player A (green car) and Player B (red car) are racing to the right from the start position. Each player's network lag (round-trip time) to the server is slightly different. Let's assume Player B has a larger lag of 250ms, and Player A has a lag of 150ms one way to the server.

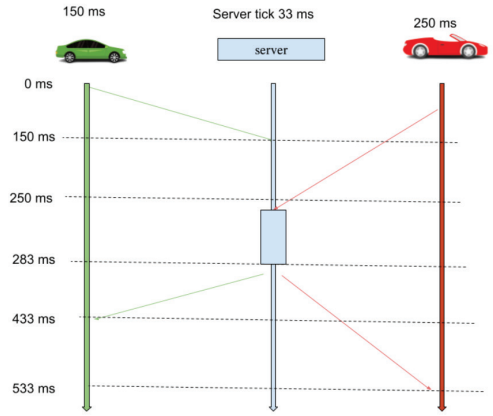

Figure 10: Network events in a racing game. The players report their position to the server, and their inputs are validated after input from both players is received. The server sends the world state to both clients, who then apply and render it.

Even though both clients receive the data at different times, the cars trace the same path, and there will be no discrepancies in determining the winner. The server view (state) is slightly ahead of the clients' views. The clients are not in perfect sync because of differences in network characteristics. However, the clients render the game state and progress without losing information.

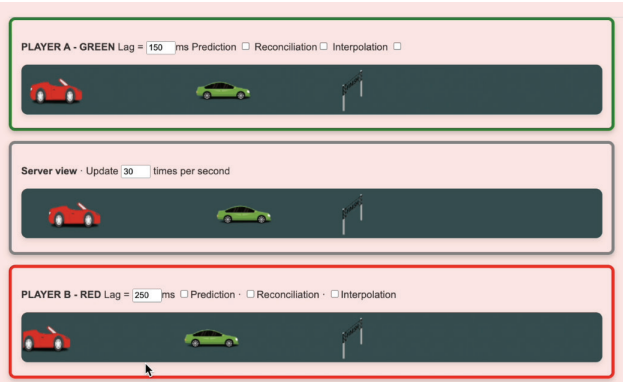

Figure 11: Racing simulation with lockstep. In the visualization of the diagram before, Player A (green car) has a lag of 150 ms, and Player B (red car) has a lag of 250 ms. All clients are in lockstep, so both Player A and Player B are perfectly in sync.

If we pay close attention to Figure 11, we see that the updates happened in the following order.

• Server

• Player A is a client with a shorter (150ms) lag.

• Player B is a client with a larger (250ms) lag.

DOI: 10.36838/v8i1.56

Network latency is still a challenge in this scenario, as it takes significant time to synchronize the server and client. A large network latency for one client ruins the game for all the clients, once again worsening the game's playability.

Working Around Lockstep Delay

The following techniques are commonly employed to improve the game experience and avoid the latency issues introduced by a player with a bad network.

Client SideSide-Prediction:

Client-side prediction is a technique that aims to reduce the perceived latency and improve the game experience. With client-side prediction on, the client will use the inputs (velocity, old position, time step) to locally compute their new position and immediately apply it to themselves. They still send their inputs to the server, which computes and sends out the world state. Once the world state is sent out, the client has to accept the server's state. This could mean returning to a position the client has already traversed.

An abrupt correction is sometimes necessary if the client-predicted state and the server state deviate significantly. This is noticeable in gameplay, where an object moves forward and suddenly returns to a prior position. Visually, this causes a jarring effect as it could confuse the player regarding their position.

It is especially troublesome in shooting games where the client's location is very important. If the perceived location is not equivalent to the true location processed by the server, the game experience could suffer. To escape this jarring effect, another technique called reconciliation is employed.

Client Side Prediction (with reconciliation):

Reconciliation attempts to deal with the abrupt adjustments to the client's state. This is accomplished by tracking the predicted states in a buffer and reconciling them with server-reported states. If a server-reported state is already close to what the client saw, the server update does not affect the client (no rendering changes). With this technique, the client accepts the state changes from the server and re-applies its stored inputs to maintain the continuity of the predicted state, thus allowing it to render faster when there is no deviation.

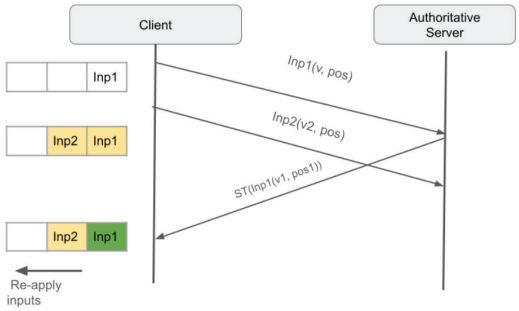

Figure 12: Tracking inputs with reconciliation buffer. This figure depicts the timing of input events to the server. The client stores the predicted states(Inp1 and Inp2) in the buffer shown on the left. The ST (state update) message from the server arrives later. The client reconciles the ST message, recognizes that Inp1 has already been locally rendered, and proceeds to reapply other predicted inputs.

In our simulation, if reconciliation is turned on for the client, the client predicts and renders its state without delay and updates the velocities and positions. It then sends this data to the server, validating the inputs and sending the world state to the clients. However, if the state (position, velocity, etc.) sent out by the server is one that the client already has close to the client's predicted state, the update is applied seamlessly.

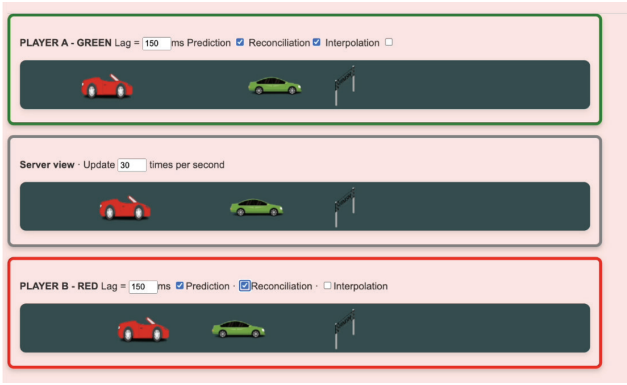

Figure 13: Prediction and reconciliation for Player A/B. Both cars have a lag of 150ms, and client-side prediction and reconciliation are enabled for both Player A and Player B. Each player renders their own car at a faster rate than the server.

Notice in Figure 13 that Player A (top section) renders his object faster than the server (middle section). There is no jarring effect since the inputs the server has validated are already traversed by the client and don't cause any disruptive impact on rendering.

However, we see that reconciliation only fixes the problem for oneself. It does not render another client's entity at a faster rate. This could quickly result in problems if there is enough lag to cause a discrepancy in who the winner is. To mitigate the effects of this problem, game developers may use the reconciliation buffer as a sort of de-jittering(delay) buffer to minimize the drift between oneself and the other players. A client may use different techniques, like *interpolation* (discussed below), to smoothly transition from the locally predicted state to the server-computed state.

Reconciliation is a great technique in first-person shooter games like Counter-Strike, Call of Duty, etc. Fast-paced games like these need frequent computations that could overwhelm the network. Using client-side prediction with reconciliation improves the responsiveness of these games in the face of network latency. These games also use reconciliation to ensure fair gameplay and prevent exploits. In games that require complex and accurate physics simulations, reconciling states becomes extremely cumbersome. Developers may avoid local client computations and lean towards waiting for server updates.

Interpolation:

Interpolation is a technique for predicting a user's intermediate path of motion using given inputs. Let us consider the first case where interpolation might be used. To smoothly render another client's motion, we must effectively "guess" the path taken. This relates to the different integration techniques in kinematics previously discussed for single-player games.

DOI: 10.36838/v8i1.56

When player A receives discrete packets of information about player B, interpolation predicts what values could have come in between to create the illusion of a continuous and periodic arrival of packets.

Moreover, interpolation could be helpful in scenarios where the server is temporarily down or the client does not have a good connection with it. By guessing the player's path based on inputs, the game will not shut down with a temporary loss of connection. Essentially, interpolation allows the game to simulate a live connection for small periods of time, even when an actual connection is not present.

A straightforward case to interpolate in one dimension is shown below.

$$\textbf{interpolated displacement} = x_0 + (x_1 - x_0)(\textbf{rendering timestamp} - t_0)/(t_1 - t_0)$$

The interpolation code fragment takes two positions (x_0, x_1) and their corresponding timestamps (t_0, t_1) and interpolates an intermediate position at the *rendering timestamp* likely to be on the path between x_0 and x_1. As discussed in the kinematic integration techniques, this simple equation grows far more complex when we have to deal with more variables, such as two-dimensional motion, air resistance, variable forces, and collisions, that can cause the entity to deviate from the predicted path.

Now that we have established the basic methods game developers use to render multiplayer objects and their complexities, let us discuss two techniques often used to deal with these common problems.

Deterministic Lockstep:

(Note here that Deterministic Lockstep is slightly different from Lockstep Synchronization mentioned previously). Players calculate and send out all their 'inputs' in deterministic lockstep.[11] Once every player has these inputs, each player will apply them to their game, leaving all players with the same frame at the same time. This method is bandwidth efficient as only game inputs are passed around the network. The state computation is performed on every player's device. However, this deterministic process has certain drawbacks. If the various clients' hardware differs, their floating-point state computations could drift over time.[11] The inputs are expected to be received in the same order over the network as they are sent. If packets are sent on a lossy network, adaptations must be made to the game to ignore lost packets and move on to the next available state.

Snapshot Interpolation:

Snapshot interpolation is a method of taking a 'snapshot' of a user's state, containing all relevant information like orientation, position, etc., and sending it to other players. The snapshots are queued in an interpolation buffer and are used for rendering. Snapshots don't need to arrive in lockstep. For example, information from two snapshots could be used to interpolate the player's path in between those snapshots.[12] While this technique is inaccurate, it fits well for large player counts. To accomplish smooth rendering, the snapshot interpolation buffers up snapshots rather than instantly rendering them.

More Challenges

Through simulations and scenarios, we have analyzed how networked multiplayer games are affected by network lag. However, this problem grows more complicated as we introduce a different type of game, including the concept of a target, a common theme in first-person shooter games. When we take our most basic case, a person is shooting at a moving target, and we don't have to account for the time of travel of the bullet. For the server to deal with this type of lag, it must essentially 'traceback' the position of the target to when it was first shot.

When we move from a rifle to a cannon, the trajectory now has a significant time and shape.[5] Let us take the case where the client is firing a cannon to hit a target. Once the client fires the projectile, this data must be sent to the server. However, the trajectory has partially been completed by the time it reaches the server. Therefore, many servers use two techniques - looking into the past and staying synchronized in the present.[13] By keeping track of the time it takes for the client to send data to the server, they can trace back the path it has already traveled, checking to see if the projectile has collided with the target. If this check fails, the server can synchronize with the client and complete the rest of the trajectory.[13] These calculations include mathematical approximations and often do not account for factors such as air resistance and air density. They often use straight trace lines to approximate the path of the trajectory to see if the collision occurred, while in reality, the path traversed could be a parabola.

Conclusion

This paper has explored how approximations are used to simulate the physics of animated objects in video games effectively and realistically. Through this analysis of simulations, we can see that the physics of our current games is far from perfect. Features like Virtual and Augmented reality require more computations, necessitating better approximations and efficient calculations. While we might have the tools to make a very computationally costly, highly accurate physics simulation, this is often unnecessary for many video games. Instead, the challenge comes in determining which aspects of physics are vital and what compromises and tradeoffs must be made to best suit the type of game. Numerical methods like the ones discussed above could introduce errors that get magnified over time. Game developers use various simplified physics models and constraint-based dynamics to restrict the domain of the object's positions and ensure certain stability. Developers take creative liberties, often sacrificing physical laws in the process. Within multiplayer games, developers may use different synchronization techniques in the face of network lag. As discussed, there are benefits and drawbacks of different techniques like client-side prediction, reconciliation, and interpolation. With each technique comes a different tradeoff that one must strategically optimize so that the game runs as smoothly and efficiently as possible. Game development that involves physics is about balancing realism with reality.

DOI: 10.36838/v8i1.56

Acknowledgments

Thank you to Ms. Hannah Bollar for guiding me through this research process.

Thank you to Gabriel Gambetta for the inspiration for a simulated network setup.

References

1. Larson, Jennifer. "How Many Frames Per Second Can the Human Eye See?" Healthline, 30 Oct. 2024, www.healthline.com/health/human-eye-fps. Accessed 15 Mar. 2025.
2. ”What Is Network Latency?" IR.com, www.ir.com/guides/what-is-network-latency. Accessed 15 Mar. 2025.
3. ”Physics Tutorial 2: Numerical Integration Methods Summary." Physics Classroom, www.physicsclassroom.com/class/kinematics/Lesson-2/Physics-Tutorial-2-Numerical-Integration-Methods-Summary. Accessed 15 Mar. 2025.
4. Fiedler, Glen. "Integration Basics." Gaffer on Games, 1 June 2004, gafferongames.com/post/ integration_basics/. Accessed 15 Mar. 2025.
5. ”Numerical Integration in Games Development." Understanding Games Development, 22 Jan. 2015, jdickinsongames.wordpress.com/2015/01/22/numerical-integration-in-games-development-2/. Accessed 15 Mar. 2025.
6. Kokhan, Olga. "Guide to Bounding Boxes in Computer Vision." TinkGroup, 16 July 2024, tinkogroup.com/what-is-a-bounding-box/. Accessed 15 Mar. 2025.
7. Souto, Nilson. "Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects." Toptal Engineering Blog, www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects. Accessed 15 Mar. 2025.
8. Peitso, Loren, and Don Brutzman. "Defeating Lag in Network-Distributed Physics Simulations." ACM Transactions on Modeling and Computer Simulation, vol. 29, no. 3, 2019, pp. 1–22, doi.org/10.1145/3208806.3208826. Accessed 15 Mar. 2025.
9. Stam, Jos. "Stable Fluids." Computer Graphics 30, no. 2, 1996, pp. 121–128, graphics.cs.cmu.edu/nsp/course/15-464/Spring11/papers/StamFluidforGames.pdf. Accessed 15 Mar. 2025.
10. Gambetta, Gabriel. "Client-Server Game Architecture." Gabriel Gambetta, gabrielgambetta.com/client- server-game-architecture.html. Accessed 15 Mar. 2025.
11. Fiedler, Glenn. "Deterministic Lockstep." Gaffer on Games, 2014, gafferongames.com/post/ deterministic_lockstep/. Accessed 15 Mar. 2025.
12. Fiedler, Glenn. "Snapshot Interpolation." Gaffer on Games, 30 Nov. 2014, gafferongames.com/post/ snapshot_interpolation/. Accessed 15 Mar. 2025.
13. Teymory, Neema. "Why Making Multiplayer Games Is Hard: Lag Compensating Weapons in Mech." Game Developer, 6 Sept. 2016, www.gamedeveloper.com/programming/why-making-multiplayer-games-is-hard-lag-compensating-weapons-in-mechwarrior-online/. Accessed 15 Mar. 2025.

Author

Bhavya Babbellapati is a junior at Mission San Jose High School in Fremont, California. She enjoys playing the cello and animating in her free time. She hopes to pursue further studies in physics, engineering, and coding, and is looking forward to working on more projects and simulations in these fields.