18

Believable Dead Reckoning for Networked Games

Curtiss Murphy
Alion Science and Technology

18.1 Introduction

Your team's producer decides that it's time to release a networked game, saying "We can publish across a network, right?" Bob kicks off a few internet searches and replies, "Doesn't look that hard." He dives into the code, and before long, Bob is ready to begin testing. Then, he stares in bewilderment as the characters jerk and warp across the screen and the vehicles hop, bounce, and sink into the ground. Thus begins the nightmare that will be the next few months of Bob's life, as he attempts to implement dead reckoning "just one more tweak" at a time.

This gem describes everything needed to add believable, stable, and efficient dead reckoning to a networked game. It covers the fundamental theory, compares algorithms, and makes a case for a new technique. It explains what's tricky about dead reckoning, addresses common myths, and provides a clear implementation path. The topics are demonstrated with a working networked game that includes source code. This gem will help you avoid countless struggles and dead ends so that you don't end up like Bob.

18.2 Fundamentals

Bob isn't a bad developer; he just made some reasonable, but misguided, assumptions. After all, the basic concept is pretty straight forward. *Dead reckoning* is the process of predicting where an actor is right now by using its last known position, velocity, and acceleration. It applies to almost any type of moving actor, including cars, missiles, monsters, helicopters, and characters on foot. For each

remote actor being controlled somewhere else on the network, we receive updates about its kinematic state that include its position, velocity, acceleration, orientation, and angular velocity. In the simplest implementation, we take the last position we received on the network and project it forward in time. Then, on the next update, we do some sort of blending and start the process all over again. Bob is right that the fundamentals aren't that complex, but making it believable is a different story.

Myth Busting—Ground Truth

Let's start with the following fact: there is no such thing as ground truth in a networked environment. "Ground truth" implies that you have perfect knowledge of the state of all actors at all times. Surely, you can't know the exact state of all remote actors without sending updates every frame in a zero packet loss, zero latency environment. What you have instead is your own perceived truth. Thus, the goal becomes believable estimation, as opposed to perfect re-creation.

Basic Math

To derive the math, we start with the simplest case: a new actor comes across the network. In this case, one of our opponents is driving a tank, and we received our first kinematic state update as it came into view. From here, dead reckoning is a straightforward linear physics problem, as described by Aronson [1997]. Using the values from the message, we put the vehicle at position $\mathbf{P}'_0$ and begin moving it at velocity $\mathbf{V}'_0$ with acceleration $\mathbf{A}'_0$, as shown in Figure 18.1. The dead-reckoned position $\mathbf{Q}_t$ at a specific time T is calculated with the equation

$$\mathbf{Q}_t = \mathbf{P}'_0 + \mathbf{V}'_0 T + \frac{1}{2}\mathbf{A}'_0 T^2.$$

Figure 18.1. The first update is simple.

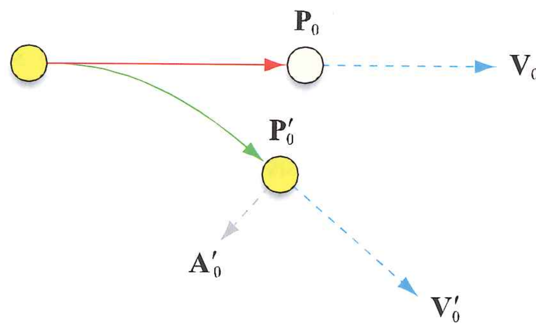


Figure 18.2. The next update creates two realities. The red line is the estimated path, and the green curve is the actual path.

Continuing our scenario, the opponent saw us, slowed his tank, and took a hard right. Soon, we receive a message updating his kinematic state. At this point, we have conflicting realities. The first reality is the position $\mathbf{Q}_t$ where we guessed he would be using the previous formula. The second reality is where he actually went, our new $\mathbf{P}'_0$, which we refer to as the *last known* state because it's the last thing we know to be correct. This dual state is the beginning of Bob's nightmares. Since there are two versions of each value, we use the prime notation (e.g., $\mathbf{P}'_0$) to indicate the last known, as shown in Figure 18.2.

To resolve the two realities, we need to create a believable curve between where we thought the tank would be, and where we estimate it will be in the future. Don't bother to path the remote tank through its last known position, $\mathbf{P}'_0$. Instead, just move it from where it is now, $\mathbf{P}_0$, to where we think it is supposed to be in the future, $\mathbf{P}'_1$.

Myth Busting—Discontinuities Are Not Minor

The human brain is amazing at recognizing patterns [Koster 2005] and, more importantly, changes in patterns, such as when the tiniest piece of fuzz moves past our peripheral vision. What this means is that players will notice subtle discontinuities in a vehicle path long before they realize the vehicle is in the wrong location. Therefore, discontinuities such as hops, warps, wobbles, and shimmies are the enemy.

18.3 Pick an Algorithm, Any Algorithm

If you crack open any good 3D math textbook, you'll find a variety of algorithms for defining a curve. Fortunately, we can discard most of them right away because they are too CPU intensive or are not appropriate (e.g., B-splines do not pass through the control points). For dead reckoning, we have the additional requirement that the algorithm must work well for a single segment of a curve passing through two points: our current location $\mathbf{P}_0$ and the estimated future location $\mathbf{P}'_1$. Given all these requirements, we can narrow the selection down to a few types of curves: cubic Bézier splines, Catmull-Rom splines, and Hermite curves [Lengyel 2004, Van Verth and Bishop 2008].

These curves perform pretty well and follow smooth, continuous paths. However, they also tend to create minor repetitive oscillations. The oscillations are relatively small, but noticeable, especially when the actor is making a lot of changes (e.g., moving in a circle). In addition, the oscillations tend to become worse when running at inconsistent frame rates or when network updates don't come at regular intervals. In short, they are too wiggly.

Projective Velocity Blending

Let's try a different approach. Our basic problem is that we need to resolve two realities (the current $\mathbf{P}_0$ and the last known $\mathbf{P}'_0$). Instead of creating a spline segment, let's try a straightforward blend. We create two projections, one with the current and one with the last known kinematic state. Then, we simply blend the two together using a standard linear interpolation (lerp). The first attempt looks like this:

$$\mathbf{P}_t = \mathbf{P}_0 + \mathbf{V}_0 T_t + \frac{1}{2}\mathbf{A}'_0 T_t^2 \qquad \text{(projecting from where we were)},$$

$$\mathbf{P}'_t = \mathbf{P}'_0 + \mathbf{V}'_0 T_t + \frac{1}{2}\mathbf{A}'_0 T_t^2 \qquad \text{(projecting from last known)},$$

$$\mathbf{Q}_t = \mathbf{P}_t + (\mathbf{P}'_t - \mathbf{P}_t)\hat{T} \qquad \text{(combined)}.$$

This gives $\mathbf{Q}_t$, the dead-reckoned location at a specified time. (Time values such as T_t and $\hat{T}$ are explained in Section 18.4.) Note that both projection equations above use the last known value of acceleration $\mathbf{A}'_0$. In theory, the current projection $\mathbf{P}_t$ should use the previous acceleration $\mathbf{A}_0$ to maintain C^2 continuity. However, in practice, $\mathbf{A}'_0$ converges to the true path much quicker and reduces oscillation.

This technique actually works pretty well. It is simple and gives a nice curve between our points. Unfortunately, it has oscillations that are as bad as or worse than the spline techniques. Upon inspection, it turns out that with all of these techniques, the oscillations are caused by the changes in velocity ($\mathbf{V}_0$ and $\mathbf{V}'_0$). Maybe if we do something with the velocity, we can reduce the oscillations. So, let's try it again, with a tweak. This time, we compute a linear interpolation between the old velocity $\mathbf{V}_0$ and the last known velocity $\mathbf{V}'_0$ to create a new blended velocity $\mathbf{V}_b$. Then, we use this to project forward from where we were.

The technique, *projective velocity blending*, works like this:

$$\mathbf{V}_b = \mathbf{V}_0 + (\mathbf{V}'_0 - \mathbf{V}_0)\hat{T} \qquad \text{(velocity blending)},$$

$$\mathbf{P}_t = \mathbf{P}_0 + \mathbf{V}_b T_t + \frac{1}{2}\mathbf{A}'_0 T_t^2 \qquad \text{(projecting from where we were)},$$

$$\mathbf{P}'_t = \mathbf{P}'_0 + \mathbf{V}'_0 T_t + \frac{1}{2}\mathbf{A}'_0 T_t^2 \qquad \text{(projecting from last known)},$$

$$\mathbf{Q}_t = \mathbf{P}_t + (\mathbf{P}'_t - \mathbf{P}_t)\hat{T} \qquad \text{(combined)}.$$

And the red lines in Figure 18.3 show what it looks like in action.

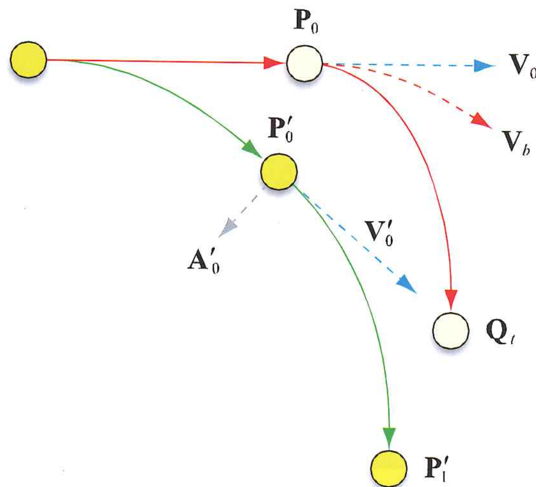


Figure 18.3. Dead reckoning with projective velocity blending shown in red.

In practice, this works out magnificently! The blended velocity and change of acceleration significantly reduce the oscillations. In addition, this technique is the most forgiving of both inconsistent network update rates and changes in frame rates.

Prove It!

So it sounds good in theory, but let's get some proof. We can perform a basic test by driving a vehicle in a repeatable pattern (e.g., a circle). By subtracting the real location from the dead-reckoned location, we can determine the error. The images in Figure 18.4 and statistics in Table 18.1 show the clear result. The projective velocity blending is roughly five to seven percent more accurate than cubic Bézier splines. That ratio improves a bit more when you can't publish acceleration. If you want to test it yourself, the demo application on the website has implementations of both projective velocity blending and cubic Bézier splines.

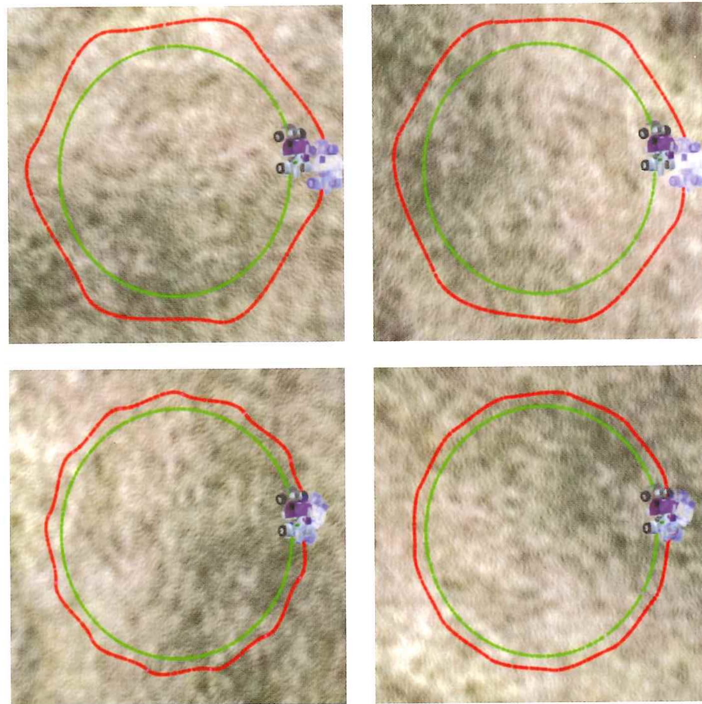


Figure 18.4. Cubic Bézier splines (left) versus projective velocity blending (right), with acceleration (top) and without acceleration (bottom).

Update Rate	Cubic Bézier (DR Error)	Projective Velocity (DR Error)	Improvement
1 update/sec	1.5723 m	1.4584 m	7.24% closer
3 updates/sec	0.1041 m	0.1112 m	6.38% closer
5 updates/sec	0.0574 m	0.0542 m	5.57% closer

Table 18.1. Improvement using projective velocity blending. Deck-reckoning (DR) error is measured in meters.

As a final note, if you decide to implement a spline behavior instead of projective velocity blending, you might consider the cubic Bézier splines [Van Verth and Bishop 2008]. They are slightly easier to implement because the control points can simply be derived from the velocities $\mathbf{V}_0$ and $\mathbf{V}'_0$. The source code on the website includes a full implementation.

18.4 Time for *T*

So far, we've glossed over time. That's okay for an introduction, but, once you begin coding, the concept of time gets twisted up in knots. So, let's talk about *T*.

What Time Is It?

The goal is to construct a smooth path that an actor can follow between two moments in time T_0 and T_1. These two times mark the exact beginning and end of the curve and are defined by locations $\mathbf{P}_0$ and $\mathbf{P}'_1$, respectively. The third time T_t is how much time has elapsed since T_0. The final time $\hat{T}$ represents how far the actor has traveled along the path as a normalized value, with $0.0 \le \hat{T} \le 1.0$.

T_0 is easy. It's the time stamp when the last known values were updated. Basically, it's "now" at the time of the update. If you've seen the movie *Spaceballs*, then T_0 is "now, now." When we process a new network update, we mark T_0 as now and set T_t back to zero. The slate is wiped clean, and we start a whole new curve, regardless of where we were.

If T_0 is now, then T_1 must be in the future. But how far into the future, T_Δ, should the projection go? Well, if we knew that the actor updates were coming at regular intervals, then we could just use the inverse update rate. So, for three updates per second, $T_\Delta = 0.333$ s. Even though network updates won't always be perfectly spaced out, it still gives a stable and consistent behavior. Naturally, the

update rate varies significantly depending on the type of game, the network conditions, and the expected actor behavior. As a general rule of thumb, an update rate of three per second looks decent and five or more per second looks great.

Time to Put It Together

From an implementation perspective, normalized time values from zero to one aren't terribly useful. In many engines, you typically get a time T_f since the last frame. We can easily add this up each frame to give the total time since the last update T_t. Once we know T_t, we can compute our normalized time $\hat{T}$ as follows:

$$T_t \leftarrow T_t + T_f$$

$$\hat{T} = \frac{T_t}{T_\Delta}.$$

Now we have all the times we need to compute the projective velocity blending equations. That leaves just one final wrinkle in time. It happens when we go past T_Δ (i.e., $T_t > T_\Delta$). This is a very common case that can happen if we miss an update, have any bit of latency, or even have minor changes in frame rate. From earlier,

$$\mathbf{Q}_t = \mathbf{P}_t + (\mathbf{P}'_t - \mathbf{P}_t)\hat{T}.$$

Because $\hat{T}$ is clamped at one, the $\mathbf{P}_t$ drops out, leaving the original equation

$$\mathbf{Q}_t = \mathbf{P}'_0 + \mathbf{V}'_0 T_t + \frac{1}{2}\mathbf{A}'_0 T_t^2.$$

The math simplifies quite nicely and continues to work for any value of $\hat{T} \geq 1.0$.

Just in Time Notes

Here are a few tips to consider:

- Due to the nature of networking, you can receive updates at any time, early or late. In order to maintain C^1 continuity, you need to calculate the instantaneous velocity between this frame's and the last frame's dead-reckoned position, $(\mathbf{P}_t - \mathbf{P}_{t-1})/T_f$. When you get the next update and start the new curve, use this instantaneous velocity for $\mathbf{V}_0$. Without this, you will see noticeable changes in velocity at each update.

- Actors send updates at different times based on many factors, including creation time, behavior, server throttling, latency, and whether they are moving. Therefore, track the various times separately for each actor (local and remote).
- If deciding your publish rate in advance is problematic, you could calculate a run-time average of how often you have been receiving network updates and use that for T_{Δ}. This works okay but is less stable than a predetermined rate.
- In general, the location and orientation get updated at the same time. However, if they are published separately, you'll need separate time variables for each.
- It is possible to receive multiple updates in a single frame. In practice, let the last update win. For performance reasons, perform the dead reckoning calculations later in the game loop, after the network messages are processed. Ideally, you will run all the dead reckoning in a single component that can split the work across multiple worker threads.
- For most games, it is not necessary to use time stamps to sync the clocks between clients/servers in order to achieve believable dead reckoning.

18.5 Publish or Perish

So far, the focus has been on handling network updates for remote actors. However, as with most things, garbage in means garbage out. Therefore, we need to take a look at the publishing side of things. In this section, forget about the actors coming in over the network and instead focus on the locally controlled actors.

When to Publish?

Let's go back and consider the original tank scenario from the opponent's perspective. The tank is now a local actor and is responsible for publishing updates on the network. Since network bandwidth is a precious resource, we should reduce traffic if possible. So the first optimization is to decide *when* we need to publish. Naturally, there are times when players are making frequent changes in direction and speed and five or more updates per second are necessary. However, there are many more times when the player's path is stable and easy to predict. For instance, the tank might be lazily patrolling, might be heading back from a respawn, or even sitting still (e.g., the player is chatting).

The first optimization is to only publish when necessary. Earlier, we learned that it is better to have a constant publish rate (e.g., three per second) because it keeps the remote dead reckoning smooth. However, before blindly publishing every time it's allowed (e.g., every 0.333 s), we first check to see if it's neces-

```
bool ShouldForceUpdate(const Vec3& pos, const Vec3& rot)
{
    bool forceUpdateResult = false;
    if (enoughTimeHasPassed)
    {
        Vec3 posMoved = pos - mCurDeadReckoned_Pos;
        Vec3 rotTurned = rot - mCurDeadReckoned_Rot;

        if ((posMoved.length2() > mPosThreshold2) ||
            (rotTurned.length2() > mRotThreshold2))
        {
            // Rot.length2 is a fast approx (i.e., not a quaternion).
            forceUpdateResult = true;
        }

        // ... Can use other checks such as velocity and accel.
    }

    return (forceUpdateResult);
}
```

Listing 18.1. Publish—is an update necessary?

sary. To figure that out, we perform the dead reckoning as if the vehicle was remote. Then, we compare the real and the dead-reckoned states. If they differ by a set threshold, then we go ahead and publish. If the real position is still really close to the dead-reckoned position, then we hold off. Since the dead reckoning algorithm on the remote side already handles $T_t > T_\Delta$, it'll be fine if we don't update right away. This simple check, shown in Listing 18.1, can significantly reduce network traffic.

What to Publish

Clearly, we need to publish each actor's kinematic state, which includes the position, velocity, acceleration, orientation, and angular velocity. But there are a few things to consider. The first, and least obvious, is the need to separate the actor's real location and orientation from its last known location and orientation. Hopefully, your engine has an actor property system [Campbell 2006] that enables you to control which properties get published. If so, you need to be absolutely sure

you never publish (or receive) the actual properties used to render location and orientation. If you do, the remote actors will get an update and render the last known values instead of the results of dead reckoning. It's an easy thing to overlook and results in a massive one-frame discontinuity (a.k.a. blip). Instead, create publishable properties for the last known values (i.e., location, velocity, acceleration, orientation, and angular velocity) that are distinct from the real values.

The second consideration is partial actor updates, messages that only contain a few actor properties. To obtain believable dead reckoning, the values in the kinematic state need to be published frequently. However, the rest of the actor's properties usually don't change that much, so the publishing code needs a way to swap between a partial and full update. Most of the time, we just send the kinematic properties. Then, as needed, we send other properties that have changed and periodically (e.g., every ten seconds) send out a heartbeat that contains everything. The heartbeat can help keep servers and clients in sync.

Myth Busting—Acceleration Is Not Always Your Friend

In the quest to create believable dead reckoning, acceleration can be a huge advantage, but be warned that some physics engines give inconsistent (a.k.a. spiky) readings for linear acceleration, especially when looked at in a single frame as an instantaneous value. Because acceleration is difficult to predict and is based on the square of time, it can sometimes make things worse by introducing noticeable under- and overcompensations. For example, this can be a problem with highly jointed vehicles for which the forces are competing on a frame-by-frame basis or with actors that intentionally bounce or vibrate.

With this in mind, the third consideration is determining what the last known values should be. The last known location and orientation come directly from the actor's current render values. However, if the velocity and acceleration values from the physics engine are giving bad results, try calculating an instantaneous velocity and acceleration instead. In extreme cases, try blending the velocity over two or three frames to average out some of the sharp instantaneous changes.

Publishing Tips

Below are some final tips for publishing:

- Published values can be quantized or compressed to reduce bandwidth [Sayood 2006].

- If an actor isn't stable at speeds near zero due to physics, consider publishing a zero velocity and/or acceleration instead. The projective velocity blend will resolve the small translation change anyway.
- If publishing regular heartbeats, be sure to sync them with the partial updates to keep the updates regular. Also, try staggering the heartbeat time by a random amount to prevent clumps of full updates caused by map loading.
- Some types of actors don't really move (e.g., a building or static light). Improve performance by using a static mode that simply teleports actors.
- In some games, the orientation might matter more than the location, or vice versa. Consider publishing them separately and at different rates.
- To reduce the bandwidth using `ShouldForceUpdate()`, you need to dead reckon the local actors in order to check against the threshold values.
- Evaluate the order of operations in the game loop to ensure published values are computed correctly. An example order might include: handle user input, tick local (process incoming messages and actor behaviors), tick remote (perform dead reckoning), publish dead reckoning, start physics (background for next frame), update cameras, render, finish physics. A bad order will cause all sorts of hard-to-debug dead reckoning anomalies.
- There is an optional damping technique that can help reduce oscillations when the acceleration is changing rapidly (e.g., zigzagging). Take the current and previous acceleration vectors and normalize them. Then, compute the dot product between them and treat it as a scalar to reduce the acceleration before publishing (shown in the `ComputeCurrentVelocity()` function in Listing 18.2).
- Acceleration in the up/down direction can sometimes cause floating or sinking. Consider publishing a zero instead.

The Whole Story

When all the pieces are put together, the code looks roughly like Listing 18.2.

```
void OnTickRemote(const TickMessage& tickMessage)
{
    // This is for local actors, but happens during Tick Remote.
    double elapsedTime = tickMessage.GetDeltaSimTime();
    bool forceUpdate = false, fullUpdate = false;

    Vec3 rot = GetRotation();
    Vec3 pos = GetTranslation();
```

```
    mSecsSinceLastUpdateSent += elapsedTime;
    mTimeUntilHeartBeat -= elapsedTime;

    // Have to update instant velocity even if we don't publish.
    ComputeCurrentVelocity(elapsedTime, pos, rot);

    if ((mTimeUntilHeartBeat <= 0.0F) || (IsFullUpdateNeeded()))
    {
        fullUpdate = true;
        forceUpdate = true;
    }
    else
    {
        forceUpdate = ShouldForceUpdate(pos, rot);
        fullUpdate = (mTimeUntilHeartBeat < HEARTBEAT_TIME * 0.1F);
    }

    if (forceUpdate)
    {
        SetLastKnownValuesBeforePublish(pos, rot);
        if (fullUpdate)
        {
            mTimeUntilHeartBeat = HEARTBEAT_TIME;  // +/- random offset
            NotifyFullActorUpdate();
        }
        else
        {
            NotifyPartialActorUpdate();
        }

        mSecsSinceLastUpdateSent = 0.0F;
    }
}

void SetLastKnownValuesBeforePublish(const Vec3& pos, const Vec3& rot)
{
    SetLastKnownTranslation(pos);
    SetLastKnownRotation(rot);
    SetLastKnownVelocity(ClampTinyValues(GetCurrentVel()));
    SetLastKnownAngularVel(ClampTinyValues(GetCurrentAngularVel()));
```

```
    // (OPTIONAL!) ACCELERATION dampen to prevent wild swings.
    // Normalize current accel. Dot with accel from last update. Use
    // the product to scale our current Acceleration.
    Vec3 curAccel = GetCurrentAccel();
    curAccel.normalize();

    float accelScale = curAccel * mAccelOfLastPublish;
    mAccelOfLastPublish = curAccel;     // (pre-normalized)
    SetLastKnownAccel(GetCurrentAccel() * Max(0.0F, accelScale));
}

void ComputeCurrentVelocity(float deltaTime, const Vec3& pos,
                                   const Vec3& rot)
{
    if ((mPrevFrameTime > 0.0F) && (mLastPos.length2() > 0.0F))
    {
        Vec3 prevComputedLinearVel = mComputedLinearVel;
        Vec3 distanceMoved = pos - mLastPos;
        mComputedLinearVel = distanceMoved / mPrevFrameTime;
        ClampTinyValues(mComputedLinearVel);

        // accel = the instantaneous differential of the velocity.
        Vec3 deltaVel = mComputedLinearVel - prevComputedLinearVel;
        Vec3 computedAccel = deltaVel / mPrevDeltaFrameTime;
        computedAccel.z() = 0.0F;    // up/down accel isn't always helpful.

        SetCurrentAcceleration(computedAccel);
        SetCurrentVelocity(mComputedLinearVel);
    }
    mLastPos = pos;
    mPrevFrameTime = deltaTime;
}
```

Listing 18.2. Publish—the whole story.

18.6 Ground Clamping

No matter how awesome your dead reckoning algorithm becomes, at some point, the problem of ground clamping is going to come up. The easiest way to visualize the problem is to drop a vehicle off of a ledge. When it impacts the ground,

the velocity is going to project the dead-reckoned position under the ground. Few things are as disconcerting as watching a tank disappear halfway into the dirt. As an example, the demo on the website allows mines to fall under ground.

Can We Fix It?

As with many dead reckoning problems, there isn't one perfect solution. However, some simple ground clamping can make a big difference, especially for far away actors. Ground clamping is adjusting an actor's vertical position and orientation to make it follow the ground. The most important thing to remember about ground clamping is that it happens *after* the rest of the dead reckoning. Do everything else first.

The following is one example of a ground clamping technique. Using the final dead reckoned position and orientation, pick three points on the bounding surface of the actor. Perform a ray cast starting above those points and directed downward. Then, for each point, check for hits and clamp the final point if appropriate. Compute the average height H of the final points $\mathbf{Q}_0$, $\mathbf{Q}_1$, and $\mathbf{Q}_2$, and compute the normal $\mathbf{N}$ of the triangle through those points as follows:

$$H = \frac{(\mathbf{Q}_0)_z + (\mathbf{Q}_1)_z + (\mathbf{Q}_2)_z}{3}$$

$$\mathbf{N} = (\mathbf{Q}_1 - \mathbf{Q}_0) \times (\mathbf{Q}_2 - \mathbf{Q}_0).$$

Use H as the final clamped ground height for the actor and use the normal to determine the final orientation. While not appropriate for all cases, this technique is fast and easy to implement, making it ideal for distant objects.

Other Considerations

- Another possible solution for this problem is to use the physics engine to prevent interpenetration. This has the benefit of avoiding surface penetration in all directions, but it can impact performance. It can also create new problems, such as warping the position, the need for additional blends, and sharp discontinuities.
- Another way to minimize ground penetration is to have local actors project their velocities and accelerations into the future before publishing. Then, damp the values as needed so that penetration will not occur on remote actors (a method known as predictive prevention). This simple trick can improve behavior in all directions and may eliminate the need to check for interpenetration.

- When working with lots of actors, consider adjusting the ground clamping based on distance to improve performance. You can replace the three-point ray multicast with a single point and adjust the height directly using the intersection normal for orientation. Further, you can clamp intermittently and use the offset from prior ground clamps.
- For character models, it is probably sufficient to use single-point ground clamping. Single-point clamping is faster, and you don't need to adjust the orientation.
- Consider supporting several ground clamp modes. For flying or underwater actors, there should be a "no clamping" mode. For vehicles that can jump, consider an "only clamp up" mode. The last mode, "always clamp to ground," would force the clamp both up and down.

18.7 Orientation

Orientation is a critical part of dead reckoning. Fortunately, the basics of orientation are similar to what was discussed for position. We still have two realities to resolve: the current drawn orientation and the last known orientation we just received. And, instead of velocity, there is angular velocity. But that's where the similarities end.

Hypothetically, orientation should have the same problems that location had. In reality, actors generally turn in simpler patterns than they move. Some actors turn slowly (e.g., cars) and others turn extremely quickly (e.g., characters). Either way, the turns are fairly simplistic, and oscillations are rarely a problem. This means C^1 and C^2 continuity is less important and explains why many engines don't bother with angular acceleration.

Myth Busting—Quaternions

Your engine might use HPR (heading, pitch, roll), *XYZ* vectors, or full rotation matrices to define an orientation. However, when it comes to dead reckoning, you'll be rotating and blending angles in three dimensions, and there is simply no getting around quaternions [Hanson 2006]. Fortunately, quaternions are easier to implement than they are to understand [Van Verth and Bishop 2008]. So, if your engine doesn't support them, do yourself a favor and code up a quaternion class. Make sure it has the ability to create a quaternion from an axis/angle pair and can perform spherical linear interpolations (slerp). A basic implementation of quaternions is provided with the demo code on the website.

```
Vec3 angVelAxis(mLastKnownAngularVelocityVector);

// normalize() returns length.
float angVelMagnitude = angVelAxis.normalize();

// Rotation around the axis is magnitude of ang vel * time.
float rotationAngle = angVelMagnitude * actualRotationTime;
Quat rotationFromAngVel(rotationAngle, angVelAxis);
```

Listing 18.3. Computing rotational change.

With this in mind, dead reckoning the orientation becomes pretty simple: project both realities and then blend between them. To project the orientation, we need to calculate the rotational change from the angular velocity. Angular velocity is just like linear velocity; it is the amount of change per unit time and is usually represented as an axis of rotation whose magnitude corresponds to the rate of rotation about that axis. It typically comes from the physics engine, but it can be calculated by dividing the change in orientation by time. In either case, once you have the angular velocity vector, the rotational change $R'_{\Delta t}$ is computed as shown in Listing 18.3.

If you also have angular acceleration, just add it to `rotationAngle`. Next, compute the two projections and blend using a spherical linear interpolation. Use the last known angular velocity in both projections, just as the last known acceleration was used for both equations in the projective velocity blending technique:

$$R'_{\Delta t} = \mathrm{quat}\left(R'_{\mathrm{mag}} T_t, R'_{\mathrm{dir}}\right) \qquad \text{(impact of angular velocity)},$$

$$R_t = R'_{\Delta t} R_0 \qquad \text{(rotated from where we were)},$$

$$R'_t = R'_{\Delta t} R'_0 \qquad \text{(rotated from last known)},$$

$$S_t = \mathrm{slerp}\left(\hat{T}, R_t, R'_t\right) \qquad \text{(combined)}.$$

This holds true for $\hat{T} < 1.0$. Once again, $\hat{T}$ is clamped at one, so the math simplifies when $\hat{T} \geq 1.0$:

$$S_t = R'_{\Delta t} R'_0 \qquad \text{(rotated from last known)}.$$

Two Wrong Turns Don't Make a Right

This technique may not be sufficient for some types of actors. For example, the orientation of a car and its direction of movement are directly linked. Unfortunately, the dead-reckoned version is just an approximation with two sources of error. The first is that the orientation is obviously a blended approximation that will be behind and slightly off. But, even if you had a perfect orientation, the remote vehicle is following a dead-reckoned path that is already an approximation. Hopefully, you can publish fast enough that neither of these becomes a problem. If not, you may need some custom actor logic that can reverse engineer the orientation from the dead-reckoned values; that is, estimate an orientation that would make sense given the dead-reckoned velocity. Another possible trick is to publish multiple points along your vehicle (e.g., one at front and one in back). Then, dead reckon the points and use them to orient the vehicle (e.g., bind the points to a joint).

18.8 Advanced Topics

This last section introduces a variety of advanced topics that impact dead reckoning. The details of these topics are generally outside the scope of this gem, but, in each case, there are specific considerations that are relevant to dead reckoning.

Integrating Physics with Dead Reckoning

Some engines use physics for both the local and the remote objects. The idea is to improve believability by re-creating the physics for remote actors, either as a replacement for or in addition to the dead reckoning. There are even a few techniques that take this a step further by allowing clients to take ownership of actors so that the remote actors become local actors, and vice versa [Feidler 2009]. In either of these cases, combining physics with dead reckoning gets pretty complex. However, the take away is that even with great physics, you'll end up with cases where the two kinematic states don't perfectly match. At that point, use the techniques in this gem to resolve the two realities.

Server Validation

Dead reckoning can be very useful for server validation of client behavior. The server should always maintain a dead-reckoned state for each player or actor. With each update from the clients, the server can use the previous last known state, the current last known state, and the ongoing results of dead reckoning as

input for its validation check. Compare those values against the actor's expected behavior to help identify cheaters.

Who Hit Who?

Imagine player A (local) shoots a pistol at player B (remote, slow update). If implemented poorly, player A has to "lead" the shot ahead or behind player B based on the ping time to the server. A good dead reckoning algorithm can really help here. As an example, client A can use the current dead-reckoned location to determine that player B was hit and then send a hit request over to the server. In turn, the server can use the dead-reckoned information for both players, along with ping times, to validate that client A's hit request is valid from client A's perspective. This technique can be combined with server validation to prevent abuse. For player A, the game feels responsive, seems dependent on skill, and plays well regardless of server lag.

Articulations

Complex actors often have *articulations*, which are attached objects that have their own independent range of motion and rotation. Articulations can generally be lumped into one of two groups: real or fake. Real articulations are objects whose state has significant meaning, such as the turret that's pointing directly at you! For real articulations, use the same techniques as if it were a full actor. Fortunately, many articulations, such as turrets, can only rotate, which removes the overhead of positional blending and ground clamping. Fake articulations are things like tires and steering wheels, where the state is either less precise or changes to match the dead-reckoned state. For those, you may need to implement custom behaviors, such as for turning the front tires to approximate the velocity calculated by the dead reckoning.

Path-Based Dead Reckoning

Some actors just need to follow a specified path, such as a road, a predefined route, or the results of an artificial intelligence plan. In essence, this is not much different from the techniques described above. Except, instead of curving between two points, the actor is moving between the beginning and end of a specified path. If the client knows how to recreate the path, then the actor just needs to publish how far along the path it is, $\hat{T}$, as well as how fast time is changing, T_v. When applicable, this technique can significantly reduce bandwidth. Moyer and Speicher [2005] have a detailed exploration of this topic.

Delayed Dead Reckoning

The first myth this gem addresses is that there is no ground truth. However, one technique, delayed dead reckoning, can nearly re-create it, albeit by working in the past. With delayed dead reckoning, the client buffers network updates until it has enough future data to re-create a path. This eliminates the need to project into the future because the future has already arrived. It simplifies to a basic curve problem. The upside is that actors can almost perfectly re-create the original path. The obvious downside is that everything is late, making it a poor choice for most real-time actors. This technique can be useful when interactive response time is not the critical factor, such as with distant objects (e.g., missiles), slow-moving system actors (e.g., merchant NPCs), or when playing back a recording. Note that delayed dead reckoning can also be useful for articulations.

Subscription Zones

Online games that support thousands of actors sometimes use a subscription-zoning technique to reduce rendering time, network traffic, and CPU load [Cado 2007]. Zoning is quite complex but has several impacts on dead reckoning. One significant difference is the addition of dead reckoning modes that swap between simpler or more complex dead reckoning algorithms. Actors that are far away or unimportant can use a low-priority mode with infrequent updates, minimized ground clamping, quantized data, or simpler math and may take advantage of delayed dead reckoning. The high-priority actors are the only ones doing frequent updates, articulations, and projective velocity blending. Clients are still responsible for publishing normally, but the server needs to be aware of which clients are receiving what information for which modes and publish data accordingly.

18.9 Conclusion

Dead reckoning becomes a major consideration the moment your game becomes networked. Unfortunately, there is no one-size-fits-all technique. The games industry is incredibly diverse and the needs of a first-person MMO, a top-down RPG, and a high-speed racing game are all different. Even within a single game, different types of actors might require different techniques.

The underlying concepts described in this gem should provide a solid foundation for adding dead reckoning to your own game regardless of the genre. Even so, dead reckoning is full of traps and can be difficult to debug. Errors can occur anywhere, including the basic math, the publishing process, the data sent over the

network, or plain old latency, lag, and packet issues. Many times, there are multiple problems going on at once and they can come from unexpected places, such as bad values coming from the physics engine or uninitialized variables. When you get stuck, refer back to the tips in each section and avoid making assumptions about what is and is not working. Believable dead reckoning is tricky to achieve, but the techniques in this gem will help make the process as easy as it can be.

Acknowledgements

Special thanks to David Guthrie for all of his contributions.

References

[Aronson 1997] Jesse Aronson. "Dead Reckoning: Latency Hiding for Networked Games." *Gamasutra*, September 19, 1997. Available at http://www.gamasutra.com/view/feature/3230/dead_reckoning_latency_hiding_for_.php.

[Cado 2007] Olivier Cado. "Propagation of Visual Entity Properties Under Bandwidth Constraints." *Gamasutra*, May 24, 2007. Available at http://www.gamasutra.com/view/feature/1421/propagation_of_visual_entity_.php.

[Campbell 2006] Matt Campbell and Curtiss Murphy. "Exposing Actor Properties Using Nonintrusive Proxies." *Game Programming Gems 6*, edited by Michael Dickheiser. Boston: Charles River Media, 2006.

[Feidler 2009] Glenn Fiedler. "Drop in COOP for Open World Games." *Game Developer's Conference*, 2009.

[Hanson 2006] Andrew Hanson. *Visualizing Quaternions*. San Francisco: Morgan Kaufmann, 2006.

[Koster 2005] Raph Koster. *A Theory of Fun for Game Design*. Paraglyph Press, 2005.

[Lengyel 2004] Eric Lengyel. *Mathematics for 3D Game Programming & Computer Graphics*, Second Edition. Hingham, MA: Charles River Media, 2004.

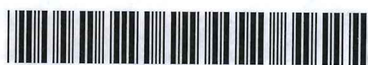
[Moyer and Speicher 2005] Dale Moyer and Dan Speicher. "A Road-Based Algorithm for Dead Reckoning." *Interservice/Industry Training, Simulation, and Education Conference*, 2005.

[Sayood 2006] Khalid Sayood. *Introduction to Data Compression*, Third Edition. San Francisco: Morgan Kaufmann, 2006.

[Van Verth and Bishop 2008] James Van Verth and Lars Bishop. *Essential Mathematics in Games and Interactive Applications: A Programmer's Guide*, Second Edition. San Francisco: Morgan Kaufmann, 2008.

Game Engine Gems 2

Edited by Eric Lengyel



2075419374

TEM-TSP - Médiathèque