Projectile Prediction

Overview

All projectiles deriving from the base *AProjectile* class come with built-in support for projectile prediction. To use prediction when spawning projectiles, projectiles must be spawned with the "Spawn Predicted Projectile" task inside a Local Predicted gameplay ability.

Projectile prediction attempts to create a more responsive and more fair experience for all players, by applying a variety of prediction and reconciliation techniques on the owning client, server, and remote clients.

On the owning client, a "fake" projectile is spawned locally to improve responsiveness. On the server, the "real" projectile is fast-forwarded (aka "forward-predicted") to match where it would be on the owning client, to compensate for the owner's latency. On remote clients, the projectile is rewound and quickly resimulated, so the projectile doesn't appear to have jumped forward in time for them (even though it did). Over time, the projectiles are synchronized and reconciled to ensure they always look and behave the same on all machines.

Projectiles can also be used without prediction. To spawn a non-predicted projectile, use a normal "Spawn Actor from Class" node in the server's execution of a gameplay ability (or in any other blueprint/code). The authoritative projectile will be replicated to all clients and behave as expected.

Projectiles can also be spawned with the "Spawn Predicted Projectile" task on listen servers. In this situation, no predicted projectile is spawned. The authoritative projectile

is spawned instantly and will be replicated to clients. This just means you can have one "Spawn Predicted Projectile" task in your gameplay ability script, and it will function as expected on both clients and listen servers.

Spawning Projectiles

Predicted projectiles should only be spawned in Local Predicted gameplay abilities with the "Spawn Predicted Projectile" task, which is responsible for spawning, initializing, and reconciling projectiles, and linking them for synchronization.

Defined in the player controller's config is an arbitrary value called *MaxPredictionPing*, which determines the maximum amount of ping with which clients can forward-predict. In other words, how far ahead of the server the client's projectile is allowed to spawn.

On the client's locally predicted activation of the task, if the client's ping is less than *MaxPredictionPing*, they'll instantly spawn a "fake" projectile locally. This is the client's "predicted" version of the projectile. (If the server eventually rejects the prediction key for the ability's activation, the task will destroy the fake projectile. Using ability tasks allows us to hook into these prediction key events easily.)

On the server's activation of the task, it will spawn the authoritative (i.e. "real") projectile. The server will immediately "fast-forward" the spawned projectile (by ticking it forward) to place it closer to where it would be on the client's machine due to latency.

The amount of time that the server's projectile is ticked forward (aka "forward-prediction time") is calculated as:

ClientBiasPct * (client's ping - *PredictionLatencyReduction*)

See ACrashPlayerController::GetForwardPredictionTime

ClientBiasPct, defined in the player controller config, is a percentage value that determines how much to favor the client when placing the real projectile. If *ClientBiasPct* is 1.0, the client is favored completely, and the authoritative projectile is placed where it would be on the client's machine. If *ClientBiasPct* is 0.0, the server is favored completely, and the authoritative projectile is not forwarded at all. We use a value of 0.5, placing the projectile halfway between the client and the server, such that it still feels accurate to the client and fairly compensates them for their latency, while still being fair for everyone else.

PredictionLatencyReduction (aka "fudge factor"), also defined in the player controller config, is a small value used to smooth the server's estimate of the client's ping. Due to processing time, tick rate, etc., the server's perceived ping of a connected client tends to be 10-20ms higher than their real ping. We use this value to get a more accurate ping estimate on the server.

If the client's ping is GREATER than *MaxPredictionPing*, all of the above still occurs. The only difference is that we delay spawning the fake projectile such that we don't forward-predict it more than *MaxPredictionPing*. For example, if a client has 150ms of ping, and *MaxPredictionPing* is 100ms, we'll delay spawning the fake projectile 50ms, such that we forward-predict with the maximum allowed 100ms. This prevents projectiles from being forward-predicted too far by clients with extremely high latency, which would look terrible and feel unfair to other players (e.g. if we allowed clients with 1000ms of ping to forward-predict 1000ms, their projectiles would appear an entire second ahead of where they SHOULD be for everyone else).

Once the authoritative projectile has been replicated to the owning client, the client fast-forwards their version of that authoritative projectile, to make up for the time it took to replicate the projectile to the client. (E.g. if the client has 60ms of ping, it takes 30ms for the server to replicate the authoritative projectile to them, meaning that that projectile will be 30ms behind where it currently is on the server.) As a result, the client's version of the authoritative projectile will be in the same location as it currently is on the server. This helps ensure that, when using the authoritative projectile to reconcile the fake one, we have an accurate representation of where the real projectile really is. Otherwise, we would be trying to reconcile the fake projectile with a version of the authoritative projectile that's considerably behind.
See AProjectile::CatchupTick

Once both projectiles have been spawned, they're linked together by the client's task and begin synchronization and reconciliation to make up for their differences on each machine.

Synchronization & Reconciliation for Predicting Client

The owning client (the one predicting the projectile) never sees the authoritative projectile. Instead, the client fully simulates with their fake projectile, and hides their version of the authoritative projectile once it's replicated. The owning client uses a variety of reconciliation techniques to ensure their fake projectile behaves the same as the real one. Fake projectiles can predict visuals (VFX, audio, etc.), but do not predict gameplay effects (damage, knockback, etc.).

As the projectiles travel, the owning client's fake projectile will continuously interpolate towards the authoritative projectile to try to synchronize with it. However, this is usually

only done for slow-moving projectiles, like rockets. It can be toggled on/off per projectile with the *bCorrectFakeProjectilePositionOverTime* option.

The event we're most concerned with when reconciling projectiles is the "detonation" event. This is triggered when a projectile stops moving (it hit a wall, stopped bouncing, etc.), or when it impacts a valid target. Because it runs its own simulation locally, the fake projectile will predict this event. If the *bPredictFX* option is enabled, the fake projectile will trigger the corresponding visuals upon detonating (explosion FX, impact sounds, etc.). If *bPredictFX* is disabled, the fake projectile will not trigger any detonation visuals. Instead, the authoritative projectile's visuals will be used once IT detonates. Generally, we don't want to predict FX for projectiles with AOE effects, like explosive rockets, but we can predict FX for smaller projectiles that only apply direct impact effects.

Projectiles stop simulating and eventually destroy themselves after detonation, so it's crucial that this event is synchronized across each machine, and reconciled in the event of missed predictions. On the owning client, there are a number of potential opportunities for missed predictions that we account for:

- After the fake projectile detonates, it sets a short timer (determined by the player's ping). If the linked authoritative projectile hasn't detonated when the timer ends, then that means the fake projectile hit something that the real projectile did not, causing it to detonate. To reconcile this, we destroy the fake projectile and switch to using the authoritative one. This will result in detonation FX being replayed at the correct location once the authoritative projectile DOES detonate.
  See AProjectile::SwitchToAuthTimer
  See AProjectile::SwitchToRealProjectile

- When the authoritative projectile detonates, if the fake projectile has NOT detonated yet, that means the fake projectile missed something that was hit by the real projectile. To reconcile this, we destroy the fake projectile and switch to the authoritative projectile. Note that this isn't necessarily a missed prediction, because the fake projectile might have JUST been about to detonate, but ended up slightly behind the authoritative projectile. This can happen if our ping estimate is slightly off, which caused us to fast-forward the authoritative projectile a little too much, causing it to detonate right before the fake one. However, swapping out our authoritative projectile for the fake one when it comes to detonation effects is completely fine, so this isn't an issue.
  See AProjectile::ShouldAuthProjDetonateOnOwner

- When the authoritative projectile detonates, if the fake projectile HAS already detonated, we check to see WHERE it detonated. If the fake projectile detonated a considerable distance from where the authoritative projectile detonated, that means it likely hit the wrong thing. To reconcile this, we destroy the fake projectile and use the authoritative projectile's detonation FX. This will result in detonation FX being replayed at the correct location.
  See AProjectile::ShouldAuthProjDetonateOnOwner

- When the authoritative projectile detonates, if the fake projectile can't even be found, that means that either the fake projectile detonated so prematurely that it's already been destroyed, or the authoritative projectile detonated so early (likely on spawn) that it hasn't been linked yet.

    - The former situation is already handled by the fake projectile, in the situation mentioned first on this list: after detonating, the fake projectile will have detected that the authoritative projectile still hasn't detonated (via the *SwitchToAuthTimer* timer) and switched to using the authoritative projectile before destroying itself. All we have to do at this point is replay

the detonation with the real projectile.
See AProjectile::ShouldAuthProjDetonateOnOwner

- The latter situation is handled via *TornOff*. We don't allow replicated projectiles to detonate if they haven't been fully spawned. Instead, we wait until they get torn off (which occurs after they've detonated on the server, AFTER *BeginPlay*), and then try to detonate them again. The second detonation should be successful, and will either be a successful prediction (if the fake projectile we've now linked to already detonated, and did so in the correct location), or will be a missed prediction that will be caught by one of the above reconciliation methods.
  See AProjectile::ShouldAuthProjDetonateOnOwner
  See AProjectile::TornOff

Synchronization & Reconciliation for Remote Clients

On remote clients (i.e. simulated proxies that are not the owner of the projectile, and thus don't have a fake projectile, and are instead using the replicated authoritative one), we also have to perform a small amount of reconciliation for detonation. To save network resources and create smoother movement, projectile movement is not replicated. Remote clients perform their projectile simulation locally, so it's possible that their detonation occurs at a different time or place than the server's. To account for this, we use the *bTearOff* property.

When the authoritative projectile detonates on the server, it tears off, which will replicate to the client to trigger the *TornOff* function. From here, if the client has not already been detonated, it will do so. Before tearing off, the authoritative projectile will also send one final movement replication update, to ensure the client's projectile

detonates in the correct location. This also happens to the owning client's authoritative projectile, which helps catch missed predictions (as explained above), since we can guarantee that our version of the authoritative projectile detonated where it was supposed to.
See AProjectile::TornOff

The reason we use tear-off (instead of something like a "DetonateIfNotDetonated" RPC) is primarily because of replication timing. TornOff will always be triggered after *BeginPlay*, and after the actor's initial replication update, which helps account for situations where the projectile detonates on spawn (i.e. when a player shoots something right in front of them).

On remote clients, due to latency and forward-prediction, projectiles will appear a noticeable distance ahead of where they spawned—or, sometimes, not at all. To fix this, when a projectile is initially replicated to a remote simulated proxy, it is rewound back to its spawn position and resimulated. When the projectile detonates on the server, the projectile is also detonated on the proxy (via *TornOff*), but done so with a short delay to allow the projectile enough time to catch up to where it detonated on the server, since it will be slightly behind due to rewinding. This process allows non-owning simulated proxies to witness the full lifetime of projectiles, while still showing their correct final effects.

Because of the delay before detonation, projectiles will sometimes detonate on clients shortly after they detonated on the server. This delay, however, is not noticeable and reasonably negligible considering that, even without the injected delay, clients would still witness a late detonation simply due to replication time.

Debugging

There are a wide range of debugging options available in the "GAS Developer Settings" editor settings category. Tooltips are available to view an explanation of each setting.

For debugging projectile movement, enable a "Projectile Debug Mode" to view the path of each projectile (the server's authoritative projectile, the client's replicated version of the authoritative projectile, and the client's fake projectile). The client's projectiles are drawn at each time stamp to show their exact difference in position. The server's projectile is not synchronized, but is drawn at a higher rate to show a continuous depiction of the projectile's position. Note that the draw frequency (which can also be modified in the settings) is limited by frame rate.

For debugging projectile spawns, initial replication, and fast-forwarding, enable "Draw Spawn Position" to draw each projectile's initial location (AFTER any forward-prediction/fast-forwarding).

For debugging projectile detonation, enable "Draw Final Position" to draw the location, direction, and radius of each projectile's detonation. ("Projectile Debug Mode" does NOT have to be enabled to use "Draw Spawn Position" or "Draw Final Position".)
See UAbilityTask_SpawnPredictedProjectile
See ACrashPlayerController