AnySync Documentation

Links

[Online Documentation](#)
[Introduction Video](#)
[Forum](#)
[Asset Store](#)
[Support Email](#)

Introduction

This asset helps to synchronize movement through the network. It's creating accurate keyframed motion by pushing new positions into a buffer and interpolating through them. It focuses on minimizing latency and resisting lag spikes by using extrapolation.

AnySync works with any networking as long as you can write code. It contains examples for Photon, UNet and WebSockets (with a sample networking built on websocket-sharp to help you get started).
Movement sync is very accurate and made with flexible send rate in mind. Great for syncing physics, character controllers, vehicles - everything what moves. Works for both predictable and unpredictable movement.

Every game is different and might have special netcode requirements, which are unavailable with simple drag-and-drop solutions screaming “not a single line of code” in a bold underlined text. This manual will guide you through the integration process. Let’s get started.

Examples

Check out the introduction video linked above, then move on to example projects. Contact support If you wish to request an example project for your networking.

UNet examples

1. Open “UNet2DExampleScene” located in “UnetSimple” folder.
2. Make a standalone build with this scene and run it.
3. Press Play button in your editor and start server or host.
4. Start client in your standalone build.
5. When you’re done with the simple 2D example, you can explore advanced 3D example, located in “UNetAdvanced” folder.

Photon example

1. Make sure you have Photon Unity Networking imported and properly set up in your project.
2. Import "PhotonExamplePackage (PUN Required)" located in AnySync examples folder.
3. Open "PhotonExampleScene” from imported Photon example folder.
4. Make a standalone build with this scene and run it.
5. Press play button in Unity editor.

WebSocket example (using websocket-sharp)

1. Photon Unity Networking and this example both import websocket-sharp.dll library. Don’t use them together or manually remove the duplicate.
2. Import “WebSocketExamplePackage” located in AnySync examples folder.
3. Open "WebSocketExampleScene" from imported WebSocket example folder.
4. Make a standalone build with this scene and run it.
5. Press play button in your editor and click “Start Server” button, and then click “Start Client” as well (so you will be able to see other clients connected to the server).
6. Click “Start Client” button in your standalone build.
7. Notice that WebSocket server acts only as a message relay server (just like Photon). This makes easier to cut it from Unity and move to a standalone server application.

Integration

SyncBuffer is the main component that does everything AnySync have. You need to update it with deltaTime and feed position keyframes. In return, it gives you relevant position and rotation data for current point in time, based on TargetLatency setting.
Here is a minimal example. You can extend it with rotation, velocities and accelerations.

1. Add SyncBuffer component to a GameObject that you wish to synchronize through Add Component -> Scripts -> SyncBuffer.
2. Get it from another script responsible for syncing your object, like this.

```
private SyncBuffer _syncBuffer;
private void Awake()
{
      _syncBuffer = GetComponent<SyncBuffer>();
}
```

3. Send keyframe data through network and feed it with new keyframes. (UNet example)

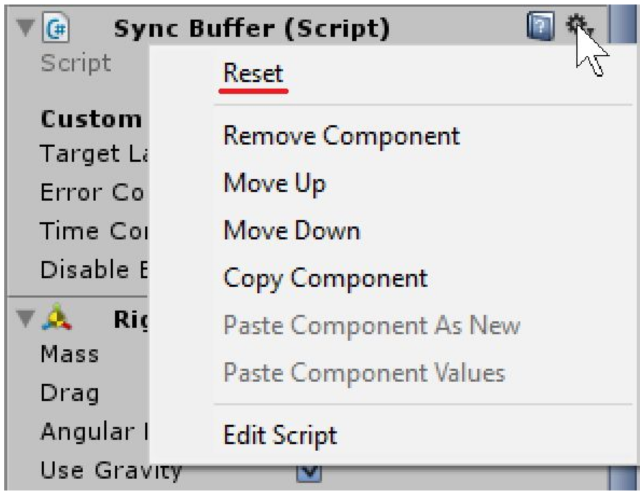
```
private float _timeSinceLastSync;
private void SendSyncMessage()
{
      CmdSync(_timeSinceLastSync, transform.position);
      _timeSinceLastSync = 0f;
}

[Command]
private void CmdSync(float interpolationTime, Vector3 position)
{
      _syncBuffer.AddKeyframe(interpolationTime, position);
}
```

4. Progress keyframe playback and apply new position data.

```
private void Update()
{
      _timeSinceLastSync += Time.deltaTime;
      if (_syncBuffer.HasKeyframes)
      {
            _syncBuffer.UpdatePlayback(Time.deltaTime);
            transform.position = _syncBuffer.Position;
      }
}
```

Settings

Before modifying any settings, you should learn what they do and what pitfalls they have. Whenever in doubt, leave all settings as defaults. Here is how to reset them.



Target Latency

- It's the most important setting that controls buffer length and extrapolation power.
- Default value is 0.075ms. It's designed around send rate of 10 messages per second, which equals to 0.1ms send interval. So it's interpolating for 0.075 milliseconds, then switches to extrapolation for remaining 0.025ms. This ensures low latency and barely introduce any extrapolation errors.
- Values above send interval increase resistance to bad network conditions, but add extra delay.
- Values below send interval compensate latency by using extrapolation, but it might introduce nasty overshooting issues (clipping through objects, floor, walls) and jerky movement during abrupt velocity changes.
- Setting TargetLatency to 0 ensures lowest possible input response delay. But It's still recommend to keep it equal to send interval +- 25% to reduce overshooting.
- To fully compensate the latency, you need to assign a negative TargetLatency value based on server ping time through script (It's not available through inspector for foolproofing reasons). It won't reduce response time because it's physically impossible, but only attempt to predict next position in the future. Do that only for racing games, because it will introduce severe extrapolation errors.

Error Correction Speed

- This setting controls position and rotation error correction speed caused by extrapolation. It’s using smooth lerp algorithm which is unnoticeable on small errors, but ensures fast correction if severe errors occur.
- Default value is 10. It’s not too harsh, but gets the job done in time.
- This feature is not available through inspector because it’s not supposed to be tweaked when using this asset for the first time.
- Correction algorithm behaves like this (but it’s more advanced behind the scenes)

```
Vector3.Lerp(oldPos, newPos, ErrorCorrectionSpeed * deltaTime);
```

Time Correction Speed

- SyncBuffer is always trying to keep up with relevant time according to remaining keyframes and TargetLatency. Sometimes network messages are lost altogether (packet loss), or delivered earlier/later than expected (packet jitter).
- Default value is 2. This ensures balanced behavior that works good under significant packet jitter and rare packet loss.
- This feature is not available through inspector because it’s not supposed to be tweaked when using this asset for the first time.
- If you expect to have severe packet jitter like on 3G mobile networks, then you might want to use lower value to prevent objects from noticeably slowing down and speeding up.
- But if you are more likely to get a packet loss, then reducing it might help to keep up with the time faster.

Disable Extrapolation

- Sometimes you can’t use extrapolation. If you enable this setting, an object will just stop when end of keyframe buffer is reached, then lerp to new position when new packet finally arrives (using ErrorCorrectionSpeed).
- By default, extrapolation is enabled, because allows using low TargetLatency and makes network lag spikes less noticeable.
- This feature is not available through inspector because it’s not supposed to be tweaked when using this asset for the first time.
- While extrapolation is disabled, you have to use high TargetLatency in order to make network issues less noticeable. About twice a send interval.

Advanced features

It’s worth noting that everything in SyncBuffer script is public, protected and virtual to allow users to extend the functionality by making a custom class that inherits from it. This will make updating to a new version much easier compared to dealing with modified source code.
Try these once you’re done with the basics and ready to unleash full potential of AnySync.

Skipping sync while idle

- To fit within tight network limitations or save bandwidth, you can send nothing while object is not moving.
- Before stopping to send further sync messages, last keyframe must explicitly state that object velocity is 0.
- If you don’t sync velocity over the network, then it automatically calculates velocity required to transition between 2 positions in specified interpolation time. You must override it with 0 before going idle, otherwise you might get a drifting effect while object is supposed to stay still.
- See example projects for implementation samples.

Teleporting

- To do that, you need to send a teleport position keyframe with 0 interpolation time on top of regular keyframe. Do it by sending 2 sync messages at the same time, or duplicating last keyframe like this.

  ```
  _syncBuffer.AddKeyframe(keyframeTime, lastKeyframe.Position);
  _syncBuffer.AddKeyframe(0f, teleportPosition);
  ```

- When adding special keyframes for teleportation, make sure total interpolation time stays the same, otherwise this will introduce time drift.
- See example projects for implementation samples. UNet and Photon examples use different techniques to achieve that.

Improving extrapolation accuracy

- The Easiest way to significantly improve extrapolation accuracy is to provide velocity with every keyframe.
- Providing acceleration will improve accuracy even further, but might lead to bad results during packet loss and lag spikes.
- Be aware that TargetLatency values lower than send interval might introduce overshooting (clipping through objects). Read settings chapter to learn more.
- Every game is different. If you really need to compensate a lot of latency, you will have to implement overshooting prevention yourself. Use LastReceivedKeyframe and IsExtrapolating properties of SyncBuffer to do it.

Common mistakes

Here are the symptoms and how to cure them.

Object is clipping through walls or floor

- It’s using too much extrapolation. You need to increase TargetLatency back to 0.075

Object is sliding when supposed to stay still

- This is also an issue of extrapolation. Increase TargetLatency to fix this it.

Laggy movement

- Make sure you are not pushing same keyframe into a buffer multiple times. You might accidentally run into that issue on host if you're using UNet HLAPI.
- Another source of the issue might be related to using rigidbody.position inside Update(). Please use FixedUpdate() to read rigidbody.position, or read transform.position instead.
- Make sure you don't send keyframes with 0 interpolation time on accident. Those are used for teleportation (they ignore smooth error correction).

Change log

v1.05

- Added simple 2D UNet example to help with the learning curve.
- Fixed issues in example when PhotonView has multiple observed components.
- Removed all inspector settings except TargetLatency to avoid confusion. Everything is still accessible from the code.
- Allowed greater flexibility by using protected fields and virtual methods.

v1.04

- Restricted TargetLatency inspector range to 0.0 - 0.5 in order to avoid confusion. Use scripts to assign a negative value in the future (if you really need to).
- Fixed UNet and WebSocket examples to send 10 messages per second instead of 8.

v1.03

- Added WebSocket example.
- Fixed more issues in Photon example.
- Fixed teleporting while object is idle.

v1.02

- New documentation.
- Fixed some potential issues on teleporting in Photon example.

v1.01

- Added an example package for Photon Unity Networking.