Best Practices

Use Structs for `INPUT` and `STATE`

- Define `INPUT` and `STATE` as **structs** in C#, not classes.
- Structs are value types, which ensures predictable behavior during prediction and reconciliation.
- Example:

```
public struct MyInput : IPredictedData
{
    public float horizontal;
    public float vertical;
    public bool jump;
}

public struct MyState : IPredictedData<MyState>
{
    public Vector3 position;
    public Quaternion rotation;
    public bool isJumping;
}
```

Why?

- Structs are copied by value, making them ideal for storing snapshot data that needs to be reconciled.
- Avoids unintended side effects that can occur with reference types (classes).

---

2. Initialize State with `GetInitialState`

- Use the `protected override STATE GetInitialState()` method to define the default values for your `STATE` struct.
- This method is called when the entity is first created, ensuring that it starts with a valid initial state.

Example:

```
protected override MyState GetInitialState()
{
    return new MyState
    {
        position = Vector3.zero,
        rotation = Quaternion.identity,
        isJumping = false
    };
}
```

Why?

- Ensures that your entity starts with a consistent and predictable state.
- Avoids undefined behavior caused by uninitialized state variables.

---

3. Treat `STATE` as the Source of Truth

- Any data that affects the simulation should be part of the `STATE` struct.
- Use `STATE` to store:
  - Entity position, rotation, and velocity.
  - Flags or variables that control behavior (e.g., `isJumping`, `isShooting`).
- Avoid modifying Unity components directly (e.g., `Transform.position`) without synchronizing them with the `STATE`.

Why?

- The `STATE` struct is reconciled by the CSP system, ensuring consistency between the client and server.
- Directly modifying Unity components can lead to desynchronization and unpredictable behavior.

---

4. Use `GetUnityState` and `SetUnityState` for External Components

- If your `STATE` affects Unity components (e.g., `Transform`, `Rigidbody`), use these overrides to synchronize them:
  - **`protected override void GetUnityState(ref STATE state)`**:
    - Updates the `STATE` struct with data from Unity components (e.g., reading the `Transform.position`).
  - **`protected override void SetUnityState(STATE state)`**:
    - Applies the `STATE` to Unity components (e.g., setting the `Transform.position`).

Example:

```
protected override void GetUnityState(ref MyState state)
{
    state.position = transform.position;
    state.rotation = transform.rotation;
}

protected override void SetUnityState(MyState state)
{
    transform.position = state.position;
    transform.rotation = state.rotation;
}
```

Why?

- These methods ensure that Unity components are properly synchronized with the `STATE`, maintaining consistency during prediction and reconciliation.

---

5. Make `SerializeField` Constants Only

- Use `SerializeField` only for **constant values** that do not change during simulation (e.g., speed, prefab references).
- Avoid using `SerializeField` for variables that are part of the simulation logic (e.g., position, velocity).

Why?

- `SerializeField` variables are not reconciled by the CSP system, so changing them during simulation can lead to desynchronization.

---

6. Keep Simulation Logic Deterministic

- Ensure that all simulation logic (e.g., movement, physics) is deterministic and based on the `STATE` or `INPUT`.

Why?

- Deterministic logic ensures that the client and server produce the same results, even when running at different times or frame rates.

Summary of Best Practices

Practice	Why It Matters
Use structs for `INPUT` and `STATE`	Ensures predictable behavior and avoids side effects of reference types.
Use `GetInitialState` for defaults	Provides a consistent starting state for entities.
Treat `STATE` as the source of truth	Prevents desynchronization and maintains consistency between client and server.
Use `GetUnityState` and `SetUnityState`	Properly synchronizes Unity components with the `STATE`.
Make `SerializeField` constants only	Avoids desynchronization caused by unreconciled changes.

Previous
Predicted Hierarchy

Next
Input Handling

Last updated 6 months ago