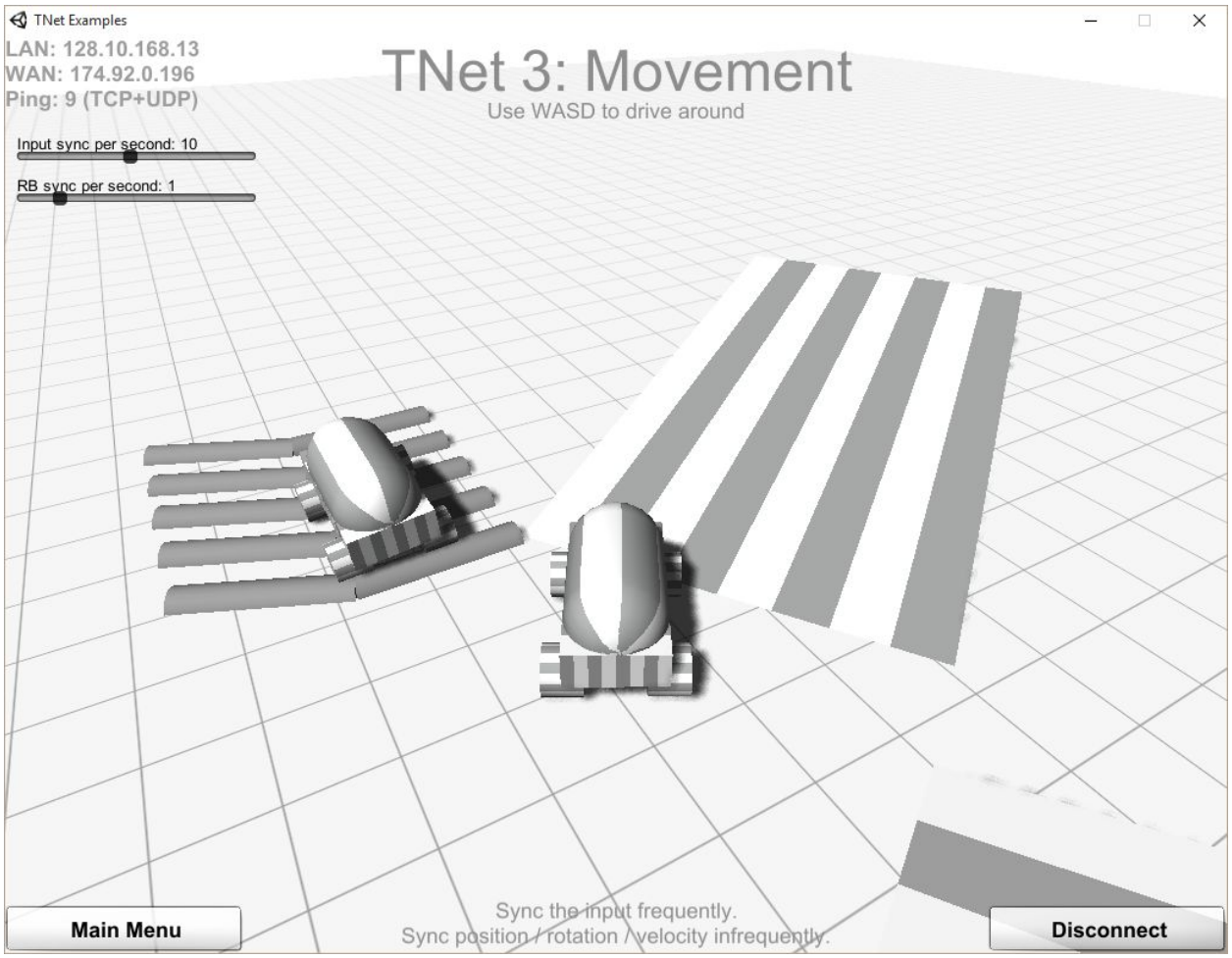
TNet 3: Movement Example In-Depth



The Movement example demonstrates how to handle sending character movement information over the network while keeping the transmitted information, and therefore lag, to a minimum.

With a naive approach of synchronizing the rigidbody frequently, the constant sync packets will not only cause visible jitter due to position updates, but can also run into a situation where players with high ping end up teleporting into a collision with other players, when in fact there wasn't one. This will often result in physics sending the two players flying out in opposite directions in a rather violent fashion. By synchronizing only the input frequently we let all clients run a local physics-based simulation that gets corrected infrequently, thus resulting in an overall smoother experience.

The scripts in the scene consists of a Chase Camera and a `TNAutoCreate` used to spawn the cars. The car it spawns is located in `\Resources\Car` and has the required `TNObject` script, as well as and a script named `ExampleCar` and `ExampleCarGUI`.

The chase camera has a very basic setup. The Main Camera is the child of the Chase Cam object and is offset providing a separation from the camera and the car. All the script needs to do is match the position of the car.

```
void FixedUpdate ()
    {
        if (target)
        {
              Vector3 forward = target.forward;
              forward.y = 0f;
              forward.Normalize();

              float delta = Time.deltaTime * 4f;
              mTrans.position = Vector3.Lerp(mTrans.position, target.position,
        delta * 4f);
              mTrans.rotation = Quaternion.Slerp(mTrans.rotation,
        Quaternion.LookRotation(forward), delta * 8f);
        }
}
```

If there is a target transform associated with this script then this GameObject attempts to match it's position exactly. The rotation of the "Chase Cam" is always pointing forward with a zeroed out Y axis to stop it from turning with the car. And the position and rotation are Lerp'd and Slerp'd over time to smooth out the movement and the chase camera is complete.

`TNAutoCreate` is on the "Spawn Car" GameObject and spawns the `\Resources\Car` prefab on start. Each client spawns their own car on start, which is how multiple cars wind up in the scene.

```
IEnumerator Start ()
{
      while (TNManager.isJoiningChannel) yield return null;
      if (channelID < 1) channelID = TNManager.lastChannelID;
      TNManager.Instantiate(channelID, "CreateAtPosition", prefabPath,
persistent, transform.position, transform.rotation);
      Destroy(gameObject);
}
```

The `Start()` function waits until `TNManager` says that the local client has connected to a channel. Once connected it Instantiates a non-persistent copy of the Car prefab using the custom RCC named `CreateAtPosition`. As a final step the `TNAutoCreate` script deletes the GameObject it's attached to stopping it from executing anymore.

```
[RCC]
static GameObject CreateAtPosition (GameObject prefab, Vector3 pos, Quaternion
rot)
```

```
{
	// Instantiate the prefab
	GameObject go = prefab.Instantiate();

	// Set the position and rotation based on the passed values
	Transform t = go.transform;
	t.position = pos;
	t.rotation = rot;
	return go;
}
```

`CreateAtPosition` is a simple implementation of an instantiate RCC. It creates a copy of the prefab, and adjusts its position and rotation before returning.

Now that the Car has been instantiated on all of the clients, take a look at the script which controls it, `ExampleCar.cs`.

```
public class ExampleCar : ExampleCarNoNetworking
```

`ExampleCar` has been split up into two classes to keep the networking code separated from everything else. `ExampleCarNoNetworking` moves the car without using any networking. It hooks up the chase camera target, the user's input read in and the movement of the physics is completed.

```
[System.NonSerialized]
public TNObject tno;
```

Due to `ExampleCar` inheriting from `ExampleCarNoNetworking` instead of `TNBehaviour`, the `tno` member variable will need to be setup manually.

```
protected override void Awake ()
{
	base.Awake();
	tno = GetComponent<TNObject>();
}
```

In the awake function `tno` gets populated with the `TNObject` component script and operates exactly the same as `TNBehaviour`.

In the `Update()` function, the car again checks `tno.mine` to see if it belongs to this client. If it doesn't it exits early. This stops all user input from being read in from the keyboard/mouse/joystick and also stops update delta's from counting down.

If however the car belongs to this client, then it looks at the update frequency of the inputs and rigidbody.

```
[Range(1f, 20f)]
public float inputUpdates = 10f;

[Range(0.25f, 5f)]
public float rigidbodyUpdates = 1f;
```

These values are used to calculate the delay between messages sent out over the network. The first one of the two to be used is `inputUpdates`.

```
float delay = 1f / inputUpdates;
```

The number of updates per second is converted to a time by dividing one second by the number of updates.

```
// The closer we are to the desired send time, the smaller is the deviation
required to send an update.
float threshold = Mathf.Clamp01(delta - delay) * 0.5f;

// If the deviation is significant enough, send the update to other players
if (Tools.IsNotEqual(mLastInput.x, mInput.x, threshold) ||
      Tools.IsNotEqual(mLastInput.y, mInput.y, threshold))
{
      mLastInputSend = time;
      mLastInput = mInput;
      tno.Send("SetAxis", Target.OthersSaved, mInput);
}
```

A threshold value between 0.0 .. 1.0 is calculated using the time since the last sent message and the the delay in seconds. That threshold value is then passed to the `Tools.IsNotEqual` helper function and is used to determine if either the horizontal or vertical controller inputs have changed enough to warrant another transmission of their value over the network.

```
static public bool IsNotEqual (float before, float after, float threshold)
{
      if (before == after) return false;
      if (after == 0f || after == 1f) return true;
      float diff = before - after;
      if (diff < 0f) diff = -diff;
      if (diff > threshold) return true;
      return false;
}
```

`IsNotEqual()` checks a couple different factors in determining if the two float values are different enough. Are they exactly the same? If they are, don't bother sending updated information. If they're not the same, is one of the values an extreme of 0 or 1? If it is, send that information out over the network. If it's not an extreme and they're not exactly the same, is the difference greater than the chosen threshold value? If it is, return 'true'.

```
[RFC] void SetAxis (Vector2 v) { mInput = v; }
```

If any of those cases winds up being true, then this `SetAxis` function is called on all clients in the current channel. All that function does is pass along the input axis information in a `Vector2` format.

```
if (mNextRB < time)
{
	mNextRB = time + 1f / rigidbodyUpdates;
	tno.Send("SetRB", Target.OthersSaved, mRb.position, mRb.rotation,
mRb.velocity, mRb.angularVelocity);
}
```

The sending of the precise location, rotation, velocity, and angularVelocity is a little simpler to send. The only check is to make sure that the amount of time which has passed is greater than the value set in `rigidbodyUpdates`. The reason this is sent less frequently is because the car's orientation should already be up to date based on the frequently sync'd input values. Plus, when the rigidbody gets sync'd, it may result in it teleporting to another position -- which can be noticeable. With infrequent updates, the teleport happens rarely and can easily be smoothed out by using interpolation (although this example doesn't delve into that).

From this point on, all of the information needed to move the car has been propagated over the network, and `ExampleCarNoNetworking` is free to use the `mInput` information in it's FixedUpdate to move the vehicle.

To recap, the exact position, rotation, velocity, and angular velocity of the car is adjusted every so often by the `SetRB` RFC. In between those updates, if the user's input has changed enough, then a smaller, more frequent update will be sent via the `SetAxis` RFC which mimics keyboard/joystick/mouse input and the car's physics is free to move itself.