Learning Goals:
1. To understand why and when dead reckoning is necessary.
2. To understand how it solves certain problems.
3. To get a practical understanding of how dead reckoning and convergence algorithms works.
4. To understand the difference between different convergence algorithms.

Assignment Lab 5:
You are supplied with example code that set up several identical clients, who all adds themselves to the scene. The code is using shared memory (Streep repository) so that each client can share some common data (position of objects on other clients) necessary for creating a network simulation. Each client has three different motion types (linear, circular & spline) by which they can animate their object. A network-latency (default 500ms) is simulated so that each player only gets position information of its ‘opponents’ at this time interval. This introduces the problem of how to update the graphical representation of the ‘opponents’ during the 500ms when their exact position is not known.

Your task is to:
- Modify the code such that dead_reckoning_type-values are shared instead of stored locally.
- Implement linear dead reckoning with zero-order convergence.
- Implement linear dead reckoning with first-order convergence.

Extra assignment:
- Make a simple game using shared states, dead reckoning, collision detection, object animation, etc. Have fun!
Dead reckoning

Dead reckoning is necessary when e.g. network latency causes information about shared transformations to be received only at certain intervals, and with a certain delay.

In order to understand the details of dead reckoning, study the figure found here: [http://esscience.anu.edu.au/lecture/ivr/sharedState/convergenceAlgo.en.html](http://esscience.anu.edu.au/lecture/ivr/sharedState/convergenceAlgo.en.html)

Linear dead reckoning is achieved by simply, making a linear transformation of each shared objects, from the last known position. That is, to keep the last known direction and speed until a new position is received. This should be done every frame, and can be calculated e.g. by the following pseudo-code:

```pseudo
For each shared object
{
    PredictedPathVector = Last_KnownPos - Last_Last_KnownPos
    DeltaTime = Last_KnownTime - Last_LastKNOWNTime
    DeltaVelocity = PredictedPathVector / DeltaTime
    NewDeltaTime = CurrentTime - LastLastKnownTime
    NewPos = LastKnownPos + NewDeltaTime * DeltaVelocity
}
```
Convergence algorithms

The dead reckoning described above results in a non-continuous movement. Since the shared objects regularly gets updated with a correct position, after a period of dead reckoning it will instantly jump to the new correct position. This is called zero-order convergence (a.k.a. snap-convergence). Such instant changes of object positions are typically an unwanted behavior. Thus, first order & higher order convergence algorithms are used to prevent this. When a new position is received, instead of jumping to it, a convergence algorithm is used to partly move the object towards the new position. Typically, the object will never reach this new position, because soon an even newer position will be received causing the object to move towards it instead. On a nonlinear movement, the calculated position of the shared object will hardly ever coincide with the real position, but instead follow another similar but smooth path close to the real objects path. First-order convergence can be calculated e.g. by the pseudo-code below. Study each path of the figure carefully, to understand how it works:


For each shared object
{
    StartDeltaTime = Last_KnownTime[0] - Last_KnownTime[2]
    PrevVelocity = PrevPredictedPathVector / PrevDeltaTime
    StartPos = Last_KnownPos[2] + StartDeltaTime * PrevVelocity
    CurrentDeltaTime = Now - Last_KnownTime[0]
    PredictedPathVector = Last_KnownPos[0] - Last_KnownPos[1]
    DeltaTime = Last_KnownTime[0] - Last_KnownTime[1]
    EndPos = Last_KnownPos[0] + CurrentDeltaTime * PredictedPathVector / DeltaTime
    LinearConvergenceVector = EndPos - StartPos
    Velocity = LinearConvergenceVector / DeltaTime
    NewPos = StartPos + CurrentDeltaTime * Velocity
}