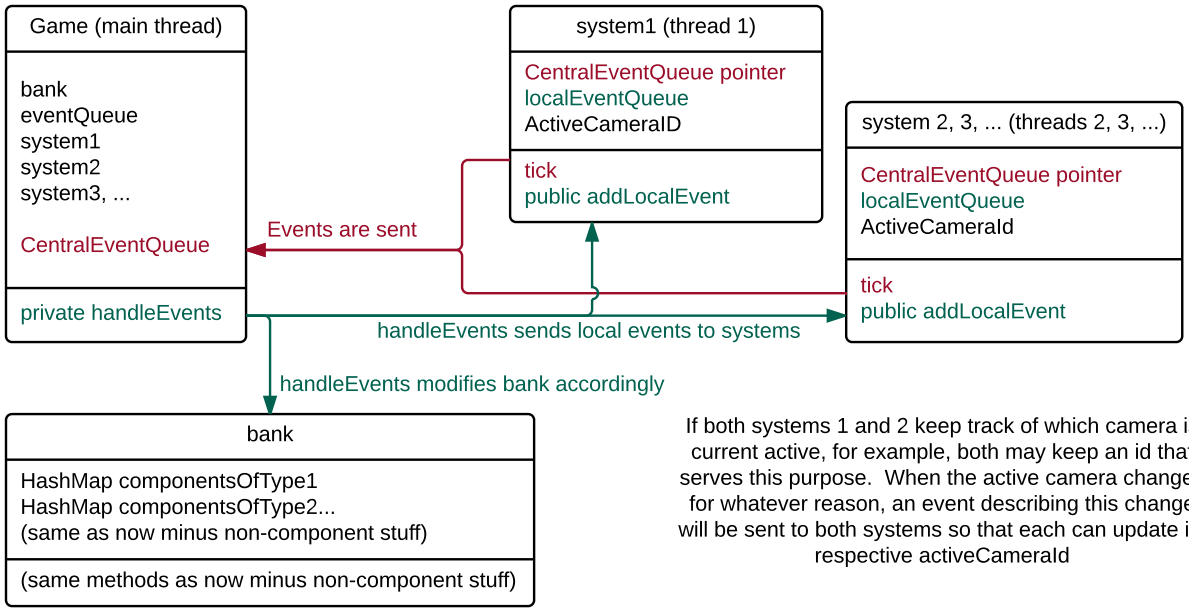

If both systems 1 and 2 keep track of which camera is current active, for example, both may keep an id that serves this purpose. When the active camera changes for whatever reason, an event describing this change will be sent to both systems so that each can update its respective activeCameraId

Overflow/underflow mechanic for CEQ - event handling loop repeats without events = underflow -> decrement physics (and other) thread's wait times. Event queue size keeps getting bigger every event handling loop = overflow = increment physics (and other) thread wait time. Increment/Decrement will be done via local events. This will handily serve as a performance regulator for the entire game. Better computer = better physics simulation.

After thought, let's scrap the idea of sending "update" events to the physics systems or "draw" events to the graphics systems. Let them run unregulated as they currently are. Instead, let's implement timestamp-based interpolation in the model rendering system so that no two frames are ever alike. This invloves storing a timestamp in every position and orientation component, then, given those and velocity info, the graphics system can render an interpolated version of each entity's position, giving a smoother effect overall and taking care of the redundant frame problem.

There is still the question of how to allow systems to read from the component collections, which is thread safe and must be done very quickly, but should not be done while the same components are being written. One way would be to allow systems to access the bank's getters. Doing this should require locks, but since it's a video game, it's not the end of the world when one frame looks a little weird. Until it is. Anyway, this solution would be easiest. On the other hand, each system could keep copies of the component data that it cares about and update those copies based on events received, but that's a ton of events and sound unnessecarily cumbersome.