EmbryoGen layout

Vladimir Ulman
20th Nov 2019, CSBD Dresden

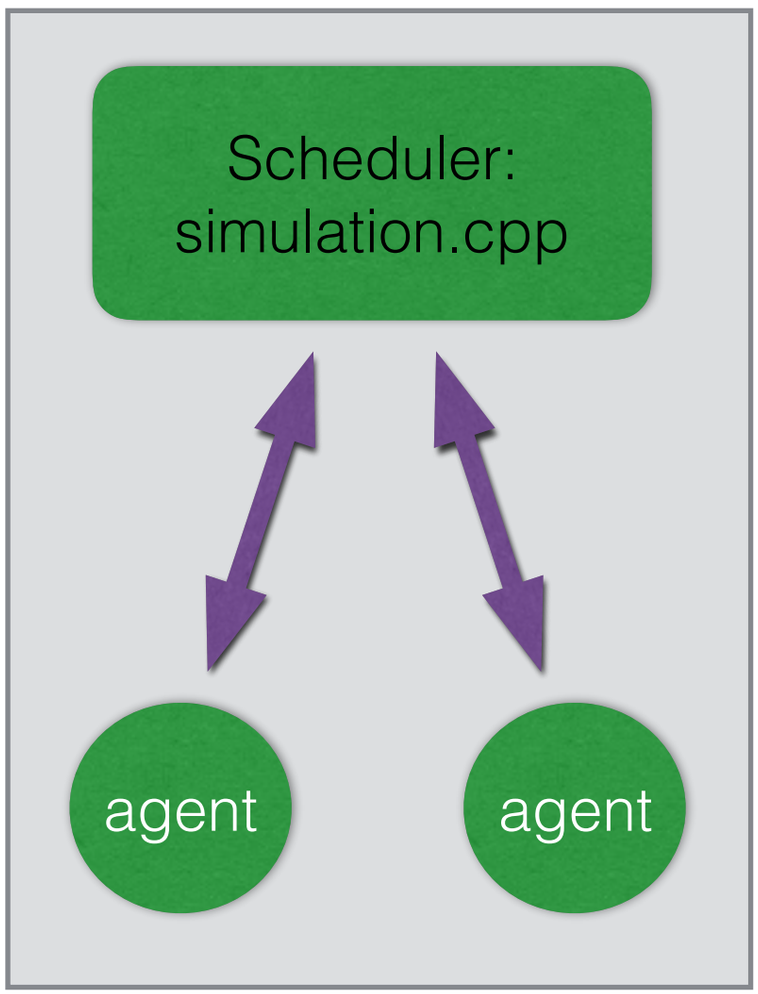
Single thread version (up to commit 880a7309)

- shared mem communication (direct calls)

- network communication MessagePassing (MPI)

- one process

- one MPI client



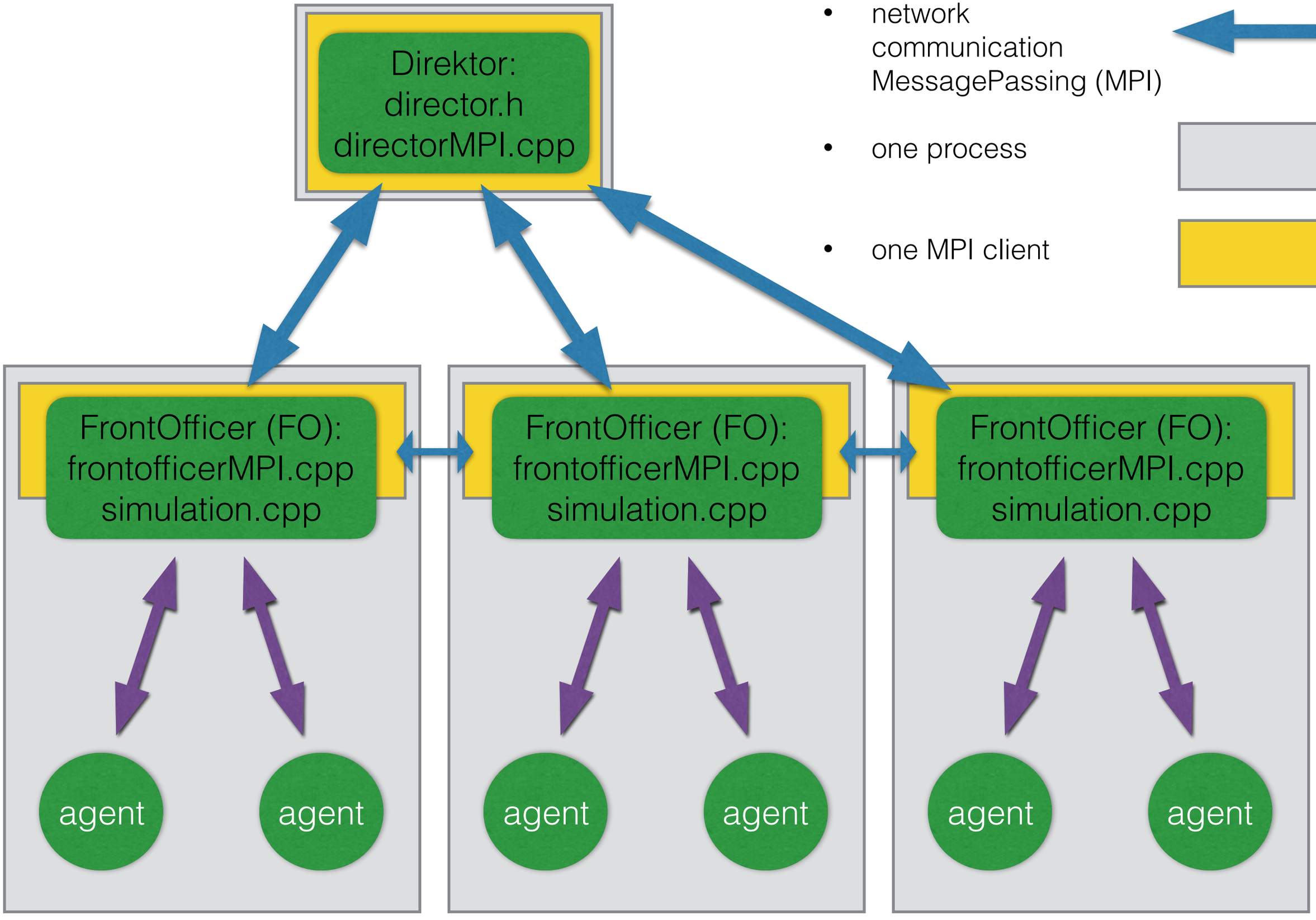
simulation.cpp

- just does everything: round-robin simulation stages, collecting images, providing service to agents, building output images

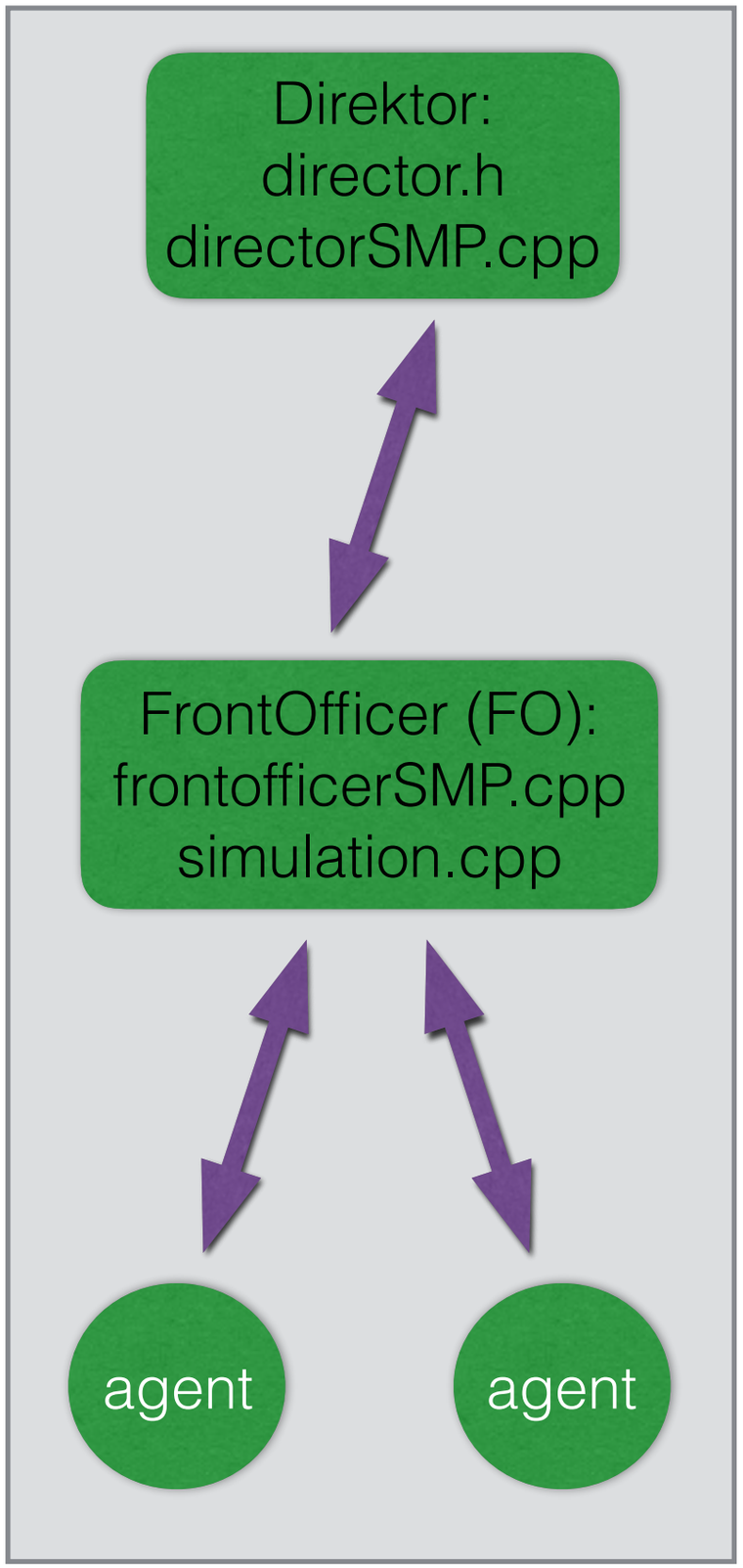
- is single-thread

Distributed version w/ MPI (planned)

- shared mem communication (direct calls)
- network communication MessagePassing (MPI)
- one process
- one MPI client



Special case: w/o MPI (planned)



- shared mem communication (direct calls)
- network communication MessagePassing (MPI)
- one process
- one MPI client

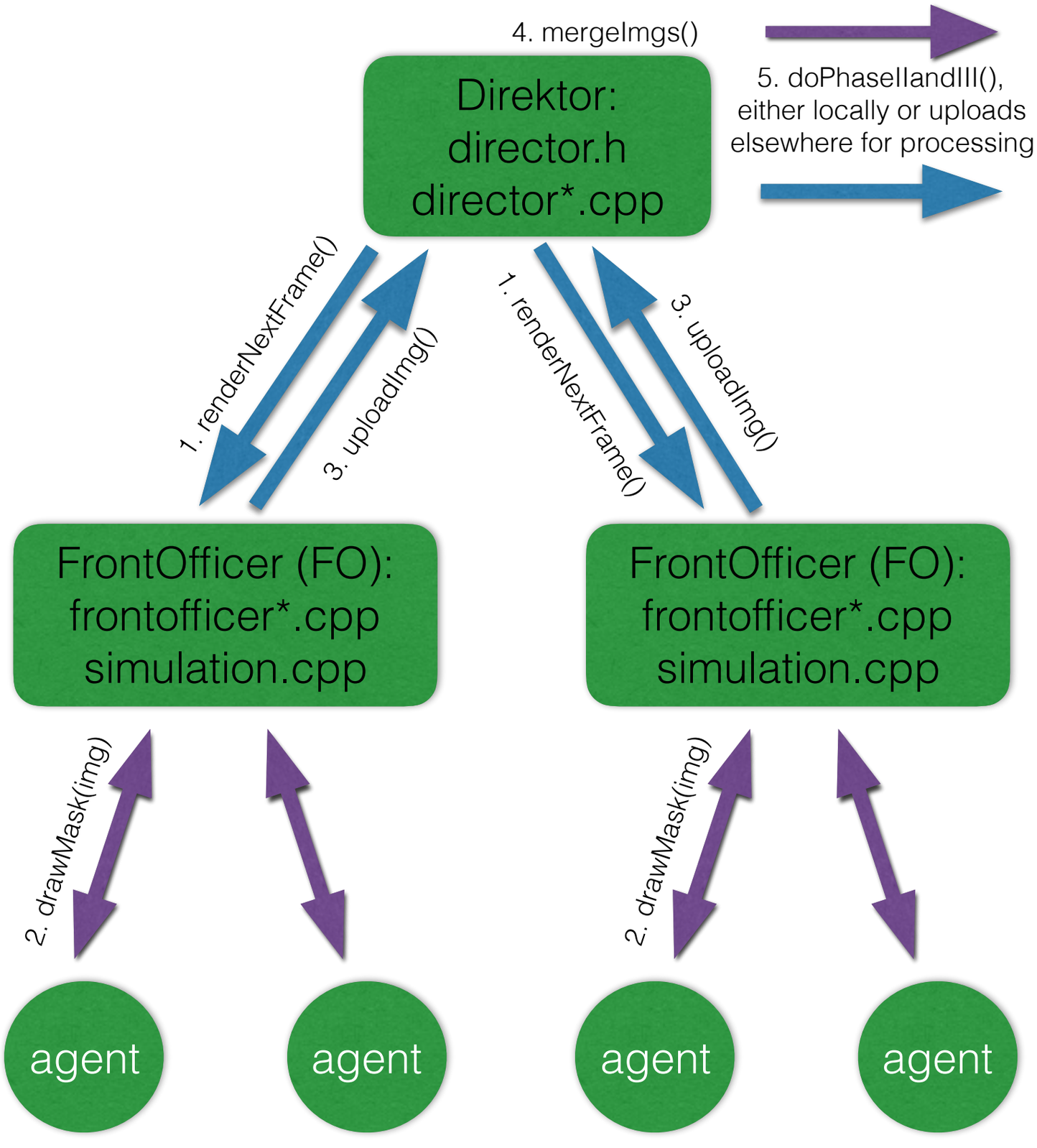
simulation.cpp
- provides basic simulation-oriented functionalities
- outsources technical/implementation details to frontofficer*.cpp, such as startNewAgent(), enableProducingOutput() etc.

frontofficer*.cpp, FrontOfficer (FO):
- forwards communication to/from the Direktor, e.g. tracks.txt metadata updating, rendering phantom imgs
- manages up-to-date list of all agents' AABBs, implements getNearbyAgents(), requests ShadowAgents from other FOs

Direktor:
- the main clock, instructs FOs to commence simulation stages
- decides when is the rendering time, collects phantom imgs
- manages tracks.txt metadata
- manages agents' IDs

Building phantom & mask images: centralized version

NOT USED



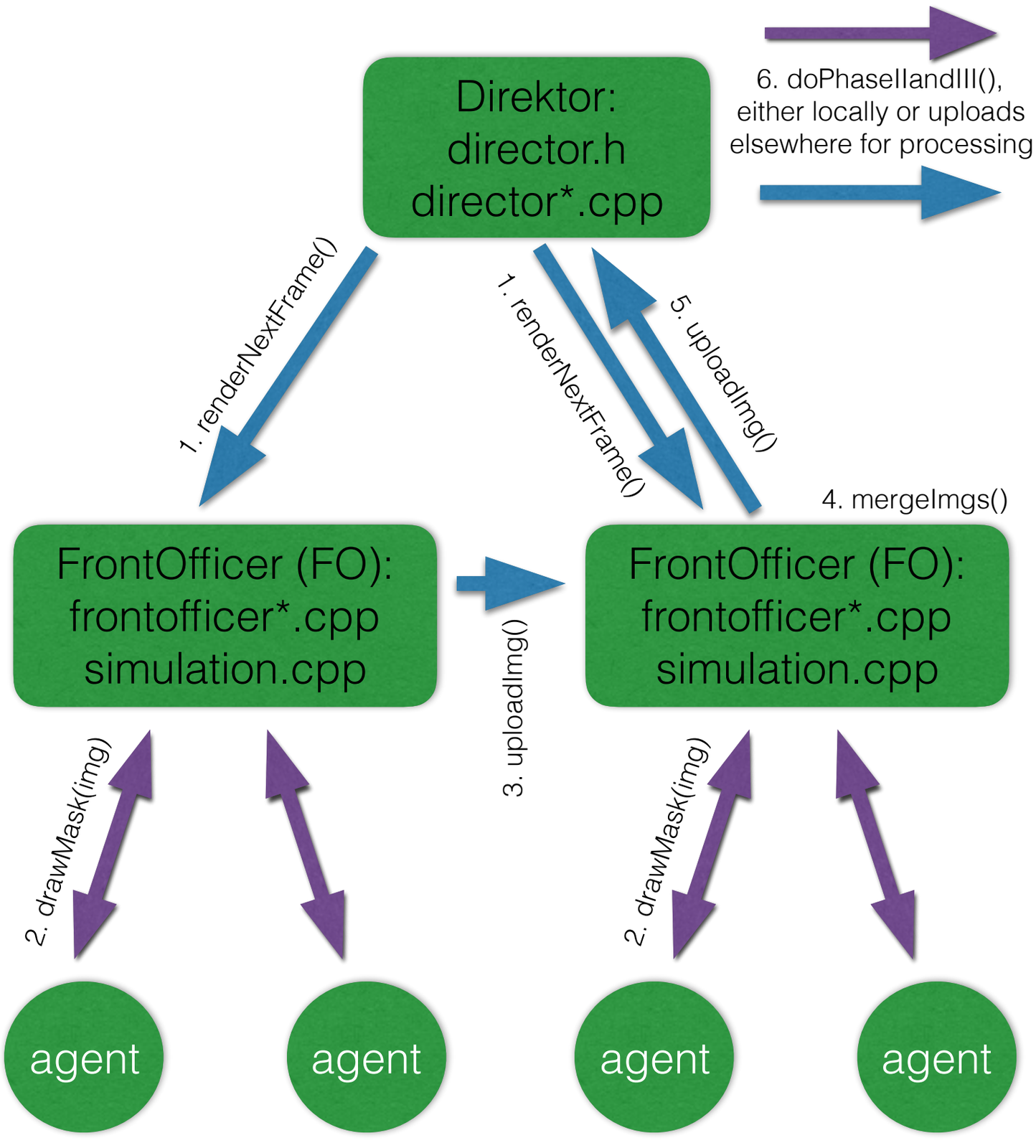
locally: ideally in another thread calls doPhaseIIandIII(), and saves the images (or sends them away)

remotely: sends away on someone who does doPhaseIIandIII() in his space, and saves, or displays, or sends further away

FO gathers all drawMasks() from his agents, uploads the result to Direktor, **Direktor** fuses the results, the displayed objects (masks, phantoms) shall not overlap, results images will likely be the same offset, size & res

+ parallel renderings
+ handling smaller images
- Direktor receives images multiple times

Building phantom & mask images: round-robin version **USED**



locally: ideally in another thread calls doPhaseIIandIII(), and saves the images (or sends them away)

remotely: sends away on someone who does doPhaseIIandIII() in his space, and saves, or displays, or sends further away

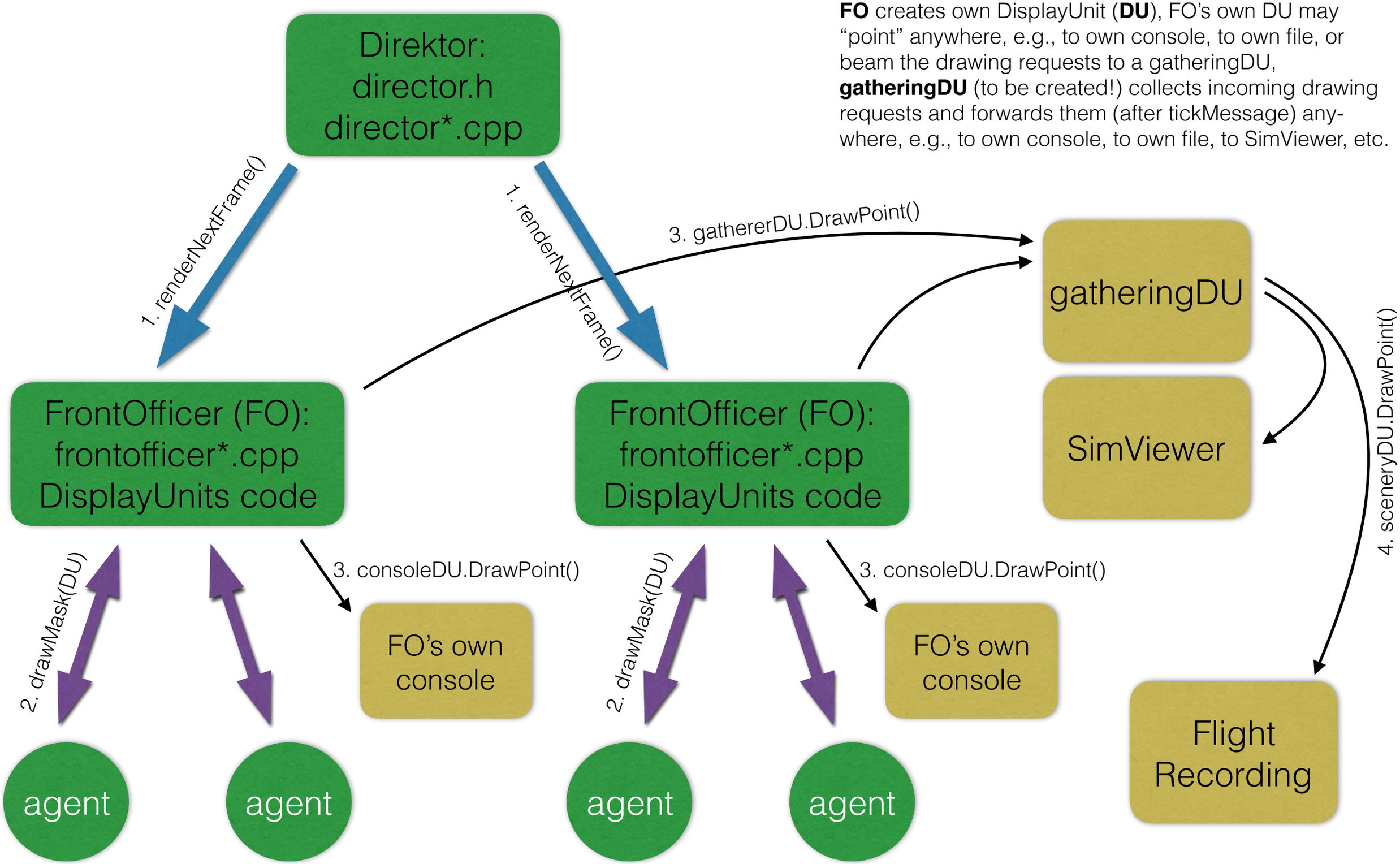
FO gathers all drawMasks() from his agents, passes the result on next FO, next FO merges this with its own results, the last FO passes the results on Direktor, **Direktor** receives the final fused results

+parallel renderings
+ one image on Direktor
-fixed order in FO communication
-handling big images

DisplayUnits communication:

DUs must be made re-entrant capable when drawMask(DU) are called in SMP-parallel

FO creates own DisplayUnit (**DU**), FO's own DU may "point" anywhere, e.g., to own console, to own file, or beam the drawing requests to a gatheringDU, **gatheringDU** (to be created!) collects incoming drawing requests and forwards them (after tickMessage) anywhere, e.g., to own console, to own file, to SimViewer, etc.



Recap and Startup:

Direktor, FOs,
simulation code (incl. agents, textures, geometries etc.),
and utils (DUs and the utils folder)
are (well… should be) generic to be used as-is for any simulation.

Scenario is the class that needs to be tailored in order to define various simulations,
this includes definition of own agents (number of them, and their initial geometries - shape of what is simulated), of the behaviour of “standard” (or skeleton, if you will) agents if one builds upon them, and what shall be the outcomes of the simulation (what is saved on hard disc, or what is online visualized).

Direktor will likely borrow the doPhaseIIandIII() from it, FOs will call initializeScenario(args) with the right args to have their relevant portions of the scene initialized.

The main.cpp always instantiates the Scenario-derived object and passes the reference on it to the Direktor and all FOs — so the common simulation settings are known to all of them.

The main.cpp always instantiates just the Direktor or just one FO (when DISTRIBUTED macro is defined), or both of them (when simulator will run only on a single machine, without MPI).
The control from the main.cpp is then given inside the Direktor to drive the simulation.
If DISTRIBUTED and MPI_rank != 0, the main.cpp gives control to the FO.

Recap and Startup:

- director.h is the interface for the Direktor,
  it will have two implementations: directorMPI.cpp and directorSMP.cpp
- frontofficer.h is the interface for a FO, again with two implementations
- CMake will define DISTRIBUTED macro and will configure the build process to take exclusively either the *MPI.cpp or *SMP.cpp sources

- In the non-DISTRIBUTED case:
  - DirectorSMP and exactly one FO will be instantiated
  - DirectorSMP class encapsulates the original Director class,
    and contains a reference on the FO through which calls to FO will be realized
  - FO will be instructed to create 1/1 portion of the scene/scenario
    and to submit outcome images to the Direktor (nextFOsID = -1)

- In the DISTRIBUTED case:
  - Director and N-1 FOs will be instantiated, where N is the number of MPI nodes/clients
  - Director and FOs will have to communicate via the MPI calls
  - each FO will be instructed to create 1/(N-1) portion of the scene/scenario, and a chain (or sub-chains!) of the outcome images passing among FOs will be defined, last one on the chain will pass to the Direktor

Recap and Startup:

- a scenario overrides:
  - ~~initializeScenario()~~ - to be called from FOs, only at init (and there's nothing for runtime),
    - broken down into:
    - initializeAgents() - called from FOs, only at init
    - initializeScene() - called from FOs and Direktor, common settings, only at init
  - updateScene() - called from FOs and Direktor, at runtime
    - synces all of them w/o communication
  - doPhaseIIandIII() - to be used from Direktor, at runtime
    - initializePhaseIIandIII() - to be called from Direktor, only at init

- *PhaseIIandIII():
  - args only images, take care solely of converting dig. phantom to final image

- *Scene():
  - no args, initializeScene() updates mutable and immutable scene params
  - updateScene() may modify on-the-go the mutable scene params
  - (im)mutable scene params as (const) reference on params object