C29 SD:ST Assignment 4 – Threads, Jobs, and Portals

Prof. Eiserloh

# **Requirements / Grading Rubric**

The assignment offers up to 105 points, with a maximum grade of 105 (taken out of 100).

## Doomenstein features (30 points)

* (2) A **Portal** class which derives from Entity, and represents a teleporter within a map or to a location on a different map.

*While parsing EntityTypes.xml during Initialization:*

* (2) All <Portal> root child elements of EntityTypes.xml are parsed; each element defines a type of portal (**EntityDef**) that can be placed in maps.
* (2) As with other entity types, each portal type has a <Physics> sub-element that defines the radius= and height= of its z-aligned cylinder.
* (2) As with other entity types, each portal type MAY have an <Appearance> sub-element that defines its (possibly animated) billboard sprite appearance and behavior. Any portal type without an <Appearance> sub-element is invisible (unless debug draw is on, showing its cylinder body).

*While parsing and creating Maps from map .xml files during post-Initialization:*

* (2) In any map .xml file, in the <Entities> section, each <Portal> element causes a **Portal** object of the named type to spawn in the map.
* (2) Each **Portal** constructs from an **EntityDef** (const reference) and **Map**\* (to the *current/starting map* in which the portal itself exists).
* (4) Each Portal instance stores the following per-instance data, parsed from various .xml attributes:
  + pos= position and yaw= Yaw in the current map, stored in its base (Entity) class;
  + destMap= destination map (name, or pointer); can be null / same as current map;
  + destPos= destination position (on destination map) where teleported entities are sent;
  + destYawOffset= yaw offset; this is added to the teleported entity’s current yaw when teleporting (e.g. 0 preserves current yaw as-is).

*At runtime:*

* (3) If any non-Portal Entity overlaps (2.5D z-aligned cylinder vs. cylinder) with a Portal, it may be teleported, based on the following rules:
* (3) For a Portal with no destination map (or the destination map is the same as the current map), all non-Portal Entities are teleported;
* (3) For a Portal with a destination map (that differs from the current map), only a player-possessed Entity is teleported (and the map is changed).
* (2) Any entity being teleported should have its position instantly changed, but its velocity preserved.
* (3) A teleported entity’s orientation pitch and roll should be preserved; its yaw is also preserved, but with the Portal’s destYawOffset added.

## Engine features (75 points)

**Note: these features must be demonstrated** in your .Exe and submitted video; this can be in your Doomenstein, Protogame, TestGame, or any other (SD-Engine) demo app. Suggested test job types include loading large binary files and reversing or xor-ing all the bytes in them, calculating a whole bunch of math, or some other type of “busywork”.

* (2) A new JobSystem engine subsystem (singleton-like). Can be owned (created, managed, destroyed) by each game’s App, or by the engine itself.
* (5) The ability for engine or game code to request the creation of any number of **WorkerThread** objects (which are owned by the JobSystem).
* (5) Each WorkerThread will run a standard “worker thread main” function/method, provided as the entry function to the std::thread constructor.
* (5) Once created, worker threads do not terminate (and are not destroyed/recreated) until requested, or until the JobSystem shuts down.
* (2) A Job abstract base class, with virtual destructor, pure virtual (required override) Execute() method, and virtual CallbackOnComplete() method.
* (2) One or more demonstration Job subclasses, in game or engine code, which demonstrates the system’s functionality.
* (2) Game/engine “client” code can instantiate Job subclass object (e.g. new MyCustomJob) and pass it into the JobSystem’s QueueJob() method.
* (3) A (private) thread-safe list of “queued jobs” that have been submitted to the JobSystem for processing.
* (1) A (private) thread-safe list of “running jobs” that have been claimed by worker threads for execution.
* (3) A (private) thread-safe list of “finished jobs” that are ready to be claimed by the main thread in game/engine “client” code.
* (5) Each worker thread tries to acquire and remove the next available queued Job (in a thread-safe way), and then call Execute() on that Job.
* (3) If a worker thread fails to acquire a job, it sleeps for 1-100 (your choice) microseconds, then tries again.
* (3) Worker threads do not “hog” shared mutexes by keeping them locked while calling Execute() on an acquired Job.
* (10) std::mutex (and/or std::atomic) is used to ensure thread-safety of all operations on shared/contested data.
* (5) When game/engine “client” code (usually in the main thread) calls CallFinishedJobCallbacksAndDelete() or equivalent on the JobSystem, all Jobs previously completed have their CallbackOnComplete() methods called – ***from the calling thread (main thread)*** – and then the Job objects are automatically deleted. Note that this must be done atomically, and without tying up the mutex that protects the “finished jobs” list while making each of the callback calls.
* (4) When the JobSystem is told to ShutDown(), it signals (using an std::atomic<bool> m\_isQuitting, or equivalent) that all worker threads should finish their current Jobs and stop taking any new jobs. It then **blocks until all threads have completed** (using the join() method on std::thread), and then finally deleting each WorkerThread object (whose destructor should, in turn, delete its std::thread object).
* (5) When running at least 12 WorkerThreads, and feeding them more than enough Jobs to keep them busy, your CPU should show near 100% busy in the Task Manager.
* (5) When running at least 12 WorkerThreads, but with no Jobs queued up (all workers are idle), your CPU should show less than 10% busy in the Task Manager (closer to 0% if your main thread is self-limiting its framerate).

*Bonus “stretch” features:*

* (1) WorkerThreads may take a set of bitflags for the types of Jobs they are allowed to process (default to 0xffffffff for “any job”); Jobs are also created with one (or more) of these bitflags set on them. WorkerThreads only process compatible Jobs, i.e. Jobs whose bitflags are contained in those of the WorkerThread.
* (1) Jobs may have a priority number (with a reasonable default); WorkerThreads always try to acquire the queued Job with the highest priority (ignoring those it cannot process, if job-worker compatibility bitflags are also used, per the previous bullet point).
* Additional methods on JobSystem which offer the ability to:
  + (1) WaitForJob( jobID )
  + (1) WaitForNextJobOfType( jobType / jobFlags )
  + (1) WaitForAllJobsOfType( jobType / jobFlags )

# **Submission**

*See “Submitting Assignments” in the course syllabus (available in Canvas) for details on assignment submission.*