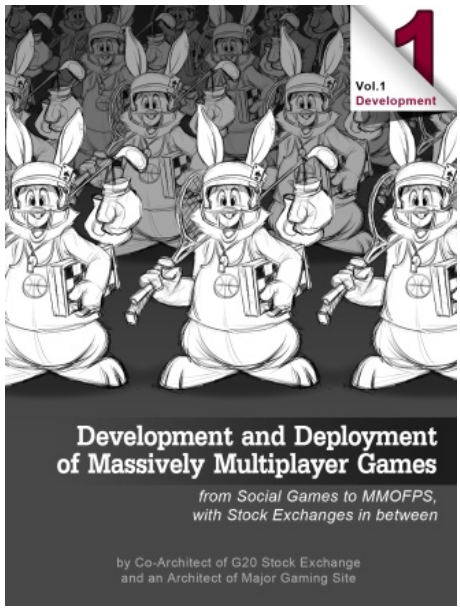
Chapter V(d). Modular Architecture: Client-Side. Client Architecture Diagram, Threads, and Game Loop

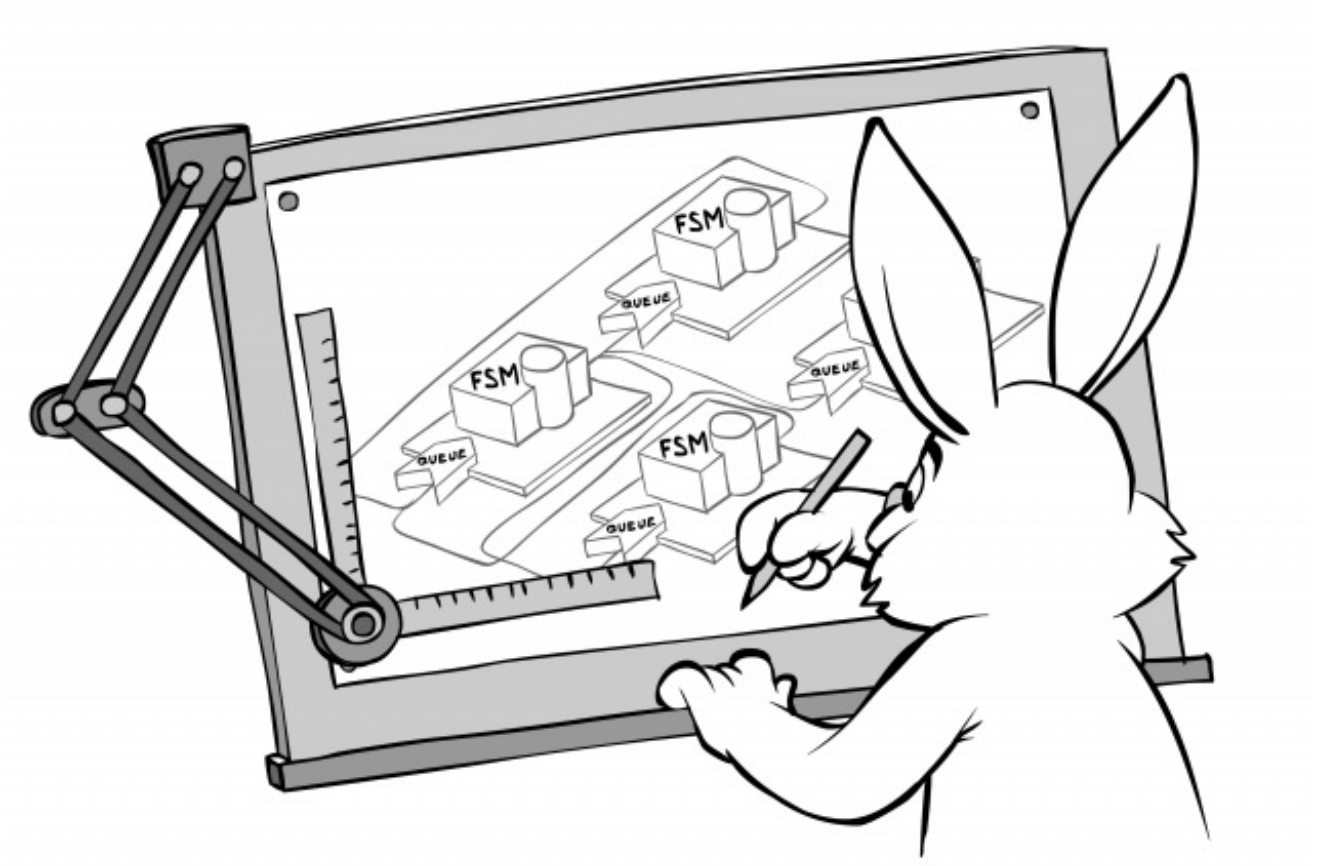
posted December 14, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

[[*This is Chapter V(d) from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further details see "Book Beta Testing". All the content published during Beta Testing, is subject to change before the book is published.*

To navigate through the book, you may want to use Development&Deployment of MMOG: Table of Contents.]]



After we've spent quite a lot of time discussing boring things such as deterministic logic and finite automata, we can go ahead and finally draw the architecture diagram for our MMO game client. Yahoo!



However, as the very last delay before that glorious and long-promised diagram, we need to define one term that we'll use in this section. Let's define "tight loop" as an *"infinite loop which goes over and over without delays, and is regardless of any input"*.[1] In other words, tight loop is bound to eat CPU cycles (and lots of them) regardless of doing any useful work.

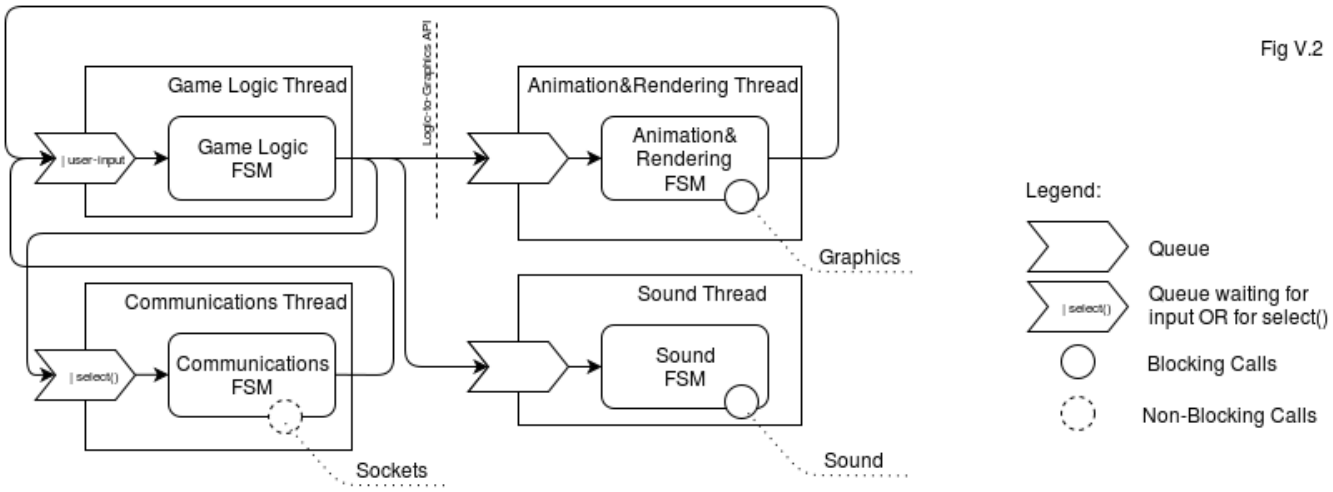
And now we're *really* ready for the diagram 🙂 .

---

[1] while different interpretations of the term "tight loop" exist out there, for our purposes this one will be the most convenient and useful

Queues-and-FSMs (QnFSM) Architecture: Generic Diagram

Fig. V.2 shows a diagram which describes a "generic" client-side architecture. It is admittedly more complicated than many of you will want or need to use; on the other hand, it represents quite a generic case, and many simplifications can be obtained right out of it by simple joining some of its elements.



Let's name this architecture a "Queues-and-FSMs Architecture" for obvious reasons, or "QnFSM" in short. Of course, QnFSM is (by far) not the only possible architecture, and even not the most popular one, but its variations have been seen to produce games with extremely good reliability, extremely good decoupling between parts, and very good maintainability. On the minus side, I can list only a bit of development overhead due to message-based exchange mechanism, but from my experience it is more than covered with better separation between different parts and very-well defined interfaces, which leads to the development speed-ups even in the medium-run (and is even more important in the long-run to avoid spaghetti code). Throw in the ability of "replay debug" and "replay-based post-mortem" in production, and it becomes a solution for lots of real-world pre-deployment and post-deployment problems.

In short – I'm an extremely strong advocate of this architecture (and its variations described below), and don't know of any practical cases when it is not the best thing you can do. While it might look over-engineered at the first glance, it pays off in the medium- and long-run[2]

I hope that the diagram on Fig V.2 should be more or less self-explanatory, but I will elaborate on a few points which might not be too obvious:

- each of FSMs is a strictly-deterministic FSM as described in "Event-Driven Programming and Finite State Machines" section above
  - while being strictly-deterministic is not a strict requirement, implementing your FSMs this way will make your debugging and post-mortem much much much easier.
- all the exchange between different FSMs is message-based. Here "message" is a close cousin of a network packet; in other words – it is just a bunch of bytes formatted according to some convention between sending thread and receiving thread.
  - There can be different ways how to pass these messages around; examples include explicit message posting, or implementing non-blocking RPC calls instead. While the Big Idea behind the QnFSM architecture won't change because of the way how the messages are posted, convenience and development time may change quite significantly. Still, while important, this is only an implementation detail which will be further discussed in Chapter [[TODO]].
  - for the messages between Game Logic Thread and Animation&Rendering Thread, format should be along the lines of "Logic-to-Graphics API", described in "Logic-to-Graphics Layer" section above. In short: it should be all about logical changes in the game world, along the lines of "NPC ID=ZZZ is currently moving along the path defined by the set of points {(X0,Y0),(X1,Y1),…} with speed V" (with coordinates being game world coordinates, not screen coordinates), or "Player at seat #N is in the process of showing his cards to the opponents".[3]

"There can be different ways how to pass these messages around; examples include explicit message posting, or implementing non-blocking RPC calls instead

- each thread has an associated Queue, which is able to accept messages, and provides a way to wait on it as long as is the Queue is empty

- the architecture is “Share-Nothing”. It means that there is no data shared between threads, and the only way to exchange data between threads, is via Queues and messages-passed-via-the-Queues
  - “share-nothing” means no thread synchronization problems (there is no need for mutexes, critical sections, etc. etc. outside of your queues). This is a Really Good Thing™, as trying to handle thread synchronization with any frequently changeable logic (such as the one within at least some of the FSMs) inevitably leads to lots and lots of problems (see, for example, [NoBugs2015])
  - of course, implementation of the Queues still needs to use inter-thread synchronization, but this is one-time effort and it has been done many times before, so it is not likely to cause too much trouble; see Chapter [[TODO]] for further details on Queues in C++
  - as a nice side effect, it means that whenever you want it, you can deploy your threads into different processes without changing *any* code within your FSMs (merely by switching to an inter-process implementation of the Queue). In particular, it can make answering very annoying questions such as “who’s guilty for the memory corruption” much more easily
- Queues of Game Logic Thread and Communications Thread, are rather unusual. They’re waiting not only for usual inter-thread messages, but also for some other stuff (namely input messages for Game Logic Thread, and network packets for the Communications Thread).
  - In most cases, at least one of these two particular queues will be supported by your platform (see Chapter [[TODO]] for details)
  - For those platforms which don’t support such queues – you can always use your-usual-inter-thread-queue (once again, the specifics will be discussed in Chapter [[TODO]]), and have an additional thread which will get user input data (or call select()), and then feed the data into your-usual-inter-thread-queue as a yet another message. This will create a strict functional equivalent (a.k.a. “compliant implementation”) of two specific Queues mentioned above

“In most cases, at least one of these two particular queues will be supported by your platform

- all the threads on the diagram (with one possible exception being Animation&Rendering Thread, see below) are *not* tight-looped, and unless there is something in their respective Queue – they just wait on the Queue until some kind of message comes in (or select() event happens)
  - while “no-tight-loops” is not a strict requirement for the client-side, wasting CPU cycles in tight loops without Really Good Reason is rarely a good idea, and might hurt quite a few of your players (those with weaker rigs).

- Animation&Rendering Thread is a potentially special case, and MAY use tight loop, see "Game Loop" subsection below for details
- to handle delays in other-than-Animation&Rendering Thread, Queues should allow FSMs to post some kind of "timer message" to the own thread
- even without tight loops it is possible to write your FSM in an "almost-tight-loop" manner that is closely resembling real-world real-time control systems (and classical Game Loop too), but without CPU overhead. See more on it in [[TODO!! – add subsection on it to "FSM" section]] section above.

---

[2] As usual, "I don't know of any cases" doesn't provide guarantees of any kind, and your mileage may vary, but at least before throwing this architecture away and doing something-that-you-like-better, please make sure to read the rest of this Chapter, where quite a few of potential concerns will be addressed
[3] yes, I know I've repeated it for quite a few times already, but it is *that* important, that I prefer to risk being bashed for annoying you rather than being pounded by somebody who didn't notice it and got into trouble

Migration from Classical 3D Single-Player Game

If you're coming from single-player development, you may find this whole diagram confusing; this maybe especially true for inter-relation between Game Logic FSM and Animation&Rendering FSM. The idea here is to have 95% of your existing "3D engine as you know it", with all the 3D stuff, as a part of "Animation&Rendering FSM". You will just need to cut off game decision logic (which will go to the server-side, and maybe partially duplicated to Game Logic FSM too for client-side prediction purposes), and UI logic (which will go into Game Logic FSM). All the mesh-related stuff should stay within Animation&Rendering FSM (even Game Logic FSM should know absolutely nothing about meshes and triangles).

"If your existing 3D engine is too complicated to fit into single-

If your existing 3D engine is too complicated to fit into single-threaded FSM, it is ok to keep it multi-threaded as long as it looks "just as an FSM" from the outside (i.e. all the communications with Animation&Rendering FSM go via messages or non-blocking RPC calls, expressed in terms of Logic-to-Graphics Layer). For details on using FSMs for multi-threaded 3D engines, see "On Additional Threads and Task-Based Multithreading" section below. Note that depending on specifics of your existing 3D rendering engine, you MAY need to resort to Option C; while Option C won't provide you with FSM goodies for your rendering engine (sorry, my supply of magic powder is quite limited), you will still be able to enjoy all

the benefits (such as replay debugging and production post-mortem) for the other parts of your client.

threaded FSM, it is ok to keep it multi-threaded as long as it looks 'just as an FSM' from the outside

It is worth noting that Game Logic FSM, despite its name, can often be more or less rudimentary, and (unless client-side prediction is used) mostly performs two functions: (a) parsing network messages and translating them into the commands of Logic-to-Graphics Layer, (b) UI handing. However, if client-side prediction is used, Game Logic FSM can become much more elaborated.

Interaction Examples in 3D World: Single-Player vs MMO

Let's consider three typical interaction examples after migration from single-player game to an MMO diagram shown above.

MMOFPS interaction example (shooting). Let's consider an MMOFPS example when Player A presses a button to shoot with a laser gun, and game logic needs to perform a raycast to see where it hits and what else happens. In single-player, all this usually happens within a 3D engine. For an MMO, it is more complicated:

- Step 1. button press goes to our authoritative server as a message
- Step 2. authoritative server receives message, performs a raycast, and calculates where the shot hits.
- Step 3. our authoritative server expresses "where it hits" in terms such as "Player B got hit right between his eyes"[4] and sends it as a message to the client (actually, to *all* the clients).
- Step 4. this message is received by Game Logic FSM, and translated into the commands of Logic-to-Graphics Layer (still without meshes and triangles, for example, "show laser ray from my gun to the point right-between-the-eyes-of-Player B", and "show laser hit right between the eyes of Player B"), which commands are sent (as messages) to Animation&Rendering FSM.
- Step 5. Animation&Rendering FSM can finally render the whole thing.[5]

While the process is rather complicated, most of the steps are inherently inevitable for an MMO; the only thing which you could theoretically save compared to the procedure described above, is merging step 4 and step 5 together (by merging Game Logic FSM and Animation&Rendering together), but I advise against it as such merging would introduce too much coupling which will hit you in the long run. Doing such different things as parsing network messages and rendering within one tightly coupled module is rarely a good idea, and it becomes even worse if there is a chance that you will ever want to use some other Animation&Rendering FSM (for example, a newer one, or the one optimized for a different platform).

MMORPG interaction example (ragdoll). In a typical MMORPG example, when an NPC is hit for 93th time and dies as a result, ragdoll physics is activated. In a typical single-player game, once again, the whole thing is usually performed within 3D engine. And once again, for a MMO the whole thing will be more complicated:

Ragdoll physics

In computer physics engines, ragdoll physics is a type of procedural animation that is often used as a replacement for traditional static death animations in video games and animated films.

— Wikipedia —

- Step 1. button press (the one which will cause NPC death) goes to authoritative server
- Step 2. server checks attack radius, calculates chances to hit, finds that the hit is successful, decreases health, and find that NPC is dead
- Step 3. server performs ragdoll simulation in the server-side 3D world. However, it doesn't need to (neither it really can) send it to clients as a complete triangle-based animation. Instead, the server can usually send to the client only a movement of "center of gravity" of NPC in question (calculated as a result of 3D simulation). This movement of "center of gravity" is sent to the client (either as a single message with the whole animation or as a series of messages with "current position" each)
  - as an interesting side-effect: as the whole thing is quite simple, there may be no real need to calculate the whole limb movement, and it may suffice to calculate just a simple parabolic movement of the "center of gravity", which MAY save you quite a bit of resources (both CPU and memory-wise) on the server side (!)
- Step 4. Game Logic FSM receives the message with "center of gravity" movement and translates it into Logic-to-Graphics commands. This doesn't necessarily need to be trivial; in particular, it may happen that Game Logic stores larger part of the game world than Animation&Rendering FSM. In this latter case, Game Logic FSM may want to check if this specific ragdoll animation is within the scope of the current 3D world of Animation&Rendering FSM.
- Step 5. Animation&Rendering FSM performs some ragdoll simulation (it can be pretty much the same simulation which has already been made on the server side, or something completely different). If ragdoll simulation is the same, then the process of ragdoll simulation on the client-side will be quite close to the one on the server-side; however, if there are any discrepancies due to not-so-perfect determinism – client-side simulation will correct coordinates so that "center of gravity" is adjusted to the position sent by server. In case of non-deterministic behaviour between client and server, the movement of the limbs on the client and the server may be different, but for a typical RPG it doesn't matter (what is really important is where the NPC eventually lands – here or over the edge of the cliff, but this is guaranteed to

be the same for all the clients as “center of gravity” comes from the server side).

UI interaction example. In a typical MMORPG game, a very common task is to show object properties when the object is currently under cursor. For the diagram above, it should be performed as follows:

- Step 1. Game Logic FSM sends a request to the Animation&Rendering FSM: “what is the object ID at screen coordinates (X,Y)?” (where (X,Y) are cursor coordinates)
- Step 2. Animation&Rendering FSM processes this (trivial) request and returns object ID back
- Step 3. Game Logic FSM finds object properties by ID, translates them into text, and instructs Animation&Rendering FSM to display object properties in HUD

While this may seem as an overkill, the overhead (both in terms of developer’s time and run time) is negligible, and good old rule of “the more cleanly separated parts you have – the easy is further development is” will more than compensate for the complexities of such separation.

---

[4] this is generally preferable to player-unrelated “laser hit at (X,Y,Z)” in case of client-side prediction; of course, in practice you’ll use some coordinates, but the point is that it is usually better to use player-related coordinates rather than absolute game world coordinates

[5] I won’t try to teach you how to render things; if you’re from 3D development side, you know much more about it than myself

FSMs and their respective States

The diagram on Fig. V.2 shows four different FSMs; while they all derive from our FiniteStateMachineBase described above, each of them is different, has a different function, and stores a different state. Let’s take a closer look at each of them.

Game Logic FSM

Game Logic FSM is the one which makes most of decisions about your game world. More strictly, these are not exactly decisions about the game world in general (this one is maintained by our authoritative server), but about *client-side copy* of the game world. In some cases it can be almost-trivial, in some cases (especially when client-side prediction is involved) it can be very elaborated.

In any case, Game Logic FSM is likely to keep a copy of the game world (or of relevant portion of the game world) from the game server, as a part of it's state. This copy has normally nothing to do with meshes, and describes things in terms such as "there is a PC standing at position (X,Y) in the game world coordinates, facing NNW", or "There are cards AS and JH on the table".

"Game Logic FSM is likely to keep a copy of the game world from the game server, as a part of it's state.

Game Logic FSM & Graphics

Probably the most closely related to Game Logic FSM is Animation&Rendering one. Most of the interaction between the two goes in the direction from Game Logic to Animation&Rendering, using Logic-to-Graphics Layer commands as messages. Game Logic FSM should instruct Animation&Rendering FSM to construct a portion of its own game copy as a 3D scene, and to update it as its own copy of the game world changes.

In addition, Game Logic FSM is going to handle (but not render) UI, such as HUDs, and various UI dialogs (including the dialogs leading to purchases, social stuff, etc.); this UI handling should be implemented in a very cross-platform manner, via sending messages to Animation&Rendering Engine. These messages, as usual, should be expressed in very graphics-agnostic terms, such as "show health at 87%", or "show the dialog described by such-and-such resource".

To handle UI, Game Logic FSM MAY send a message to Animation&Rendering FSM, requesting information such as "what object (or dialog element) is at such-and-such screen position" (once again, the whole translation between screen coordinates into world objects is made on the Animation&Rendering side, keeping Game Logic FSM free of such information); on receiving reply, Game Logic FSM may decide to update HUD, or to do whatever-else-is-necessary.

Other messages coming from Animation&Rendering FSM to Game Logic FSM, such as "notify me when the bullet hits the NPC", MAY be necessary for the client-side prediction purposes (see Chapter [[TODO]] for further discussion). On the other hand, it is very important to understand that these messages are non-authoritative by design, and that their results can be easily overridden by the server.

As you can see, there can be quite a few interactions between Game Logic FSM and Animation&Rendering FSM. Still, while it may be tempting to combine Game Logic FSM with Animation&Rendering FSM, I would advise against it at least for the games with many platforms to be supported, and for the games with Undefined Life Span; having these two FSMs separate (as shown on Fig V.2) will ensure much cleaner

separation, facilitating much-better-structured code in the medium- to long-run. On the other hand, having these two FSM running within the same thread is a very different story, is generally ok and can be done even on a per-platform basis; see "Variations" section below.

> "Having these two FSMs separate will ensure much cleaner separation, facilitating much-better-structured code in the medium- to long-run.

Game Logic FSM: Miscellaneous

There are two other things which need to be mentioned with regards to Game Logic FSM:

- **You MUST keep your Game Logic FSM truly platform-independent.** While all the other FSMs MAY be platform-specific (and separation between FSMs along the lines described above, facilitates platform-specific development when/if it becomes necessary), you should make all the possible effort to keep your Game Logic the same across all your platforms. The reason for it has already been mentioned before, and it is all about Game Logic being the most volatile of all your client-side code; it changes so often that you won't be able to keep several code bases reasonably in sync.
- If by any chance your Game Logic is that CPU-consuming that one single core won't cope with it – in most cases it can be addressed without giving up the goodies of FSM-based system, see "Additional Threads and Task-Based Multi-Threading" section below.

Animation&Rendering FSM

Animation&Rendering FSM is more or less similar to the rendering part of your usual single-player game engine. If your game is a 3D one, then in the diagram above,

> **it is Animations&Rendering FSM which keeps and cares about all the meshes, textures, and animations; as a Big Fat Rule of Thumb, nobody else in the system (including Game Logic FSM) should know about them.**

At the heart of the Animation&Rendering FSM there is a more or less traditional Game Loop.

Game Loop

Most of single-player games are based on a so-called Game Loop. Classical game loop looks more or less as follows (see, for example,

[GameProgrammingPatterns.GameLoop]):

```
while(true) {
  process_input();
  update();
  render();
}
```

Usually, Game Loop doesn't wait for input, but rather polls input and goes ahead regardless of the input being present. This is pretty close to what is often done in real-time control systems.

For our diagram on Fig V.2 above, within our Animation&Rendering Thread we can easily have something very similar to a traditional Game Loop (with a substantial part of it going within our Animation&Rendering FSM). Our Animation&Rendering Thread can be built as follows:

- Animation&Rendering Thread (outside of Animation&Rendering FSM) checks if there is anything in its Queue; unlike other Threads, it MAY proceed even if there is nothing in the Queue
- it passes whatever-it-received-from-the-Queue (or some kind of NULL if there was nothing) to Animation&Rendering FSM, alongside with any time-related information
- within the Animation&Rendering FSM's process_event(), we can still have process_input(), update() and render(), however:
  - there is no loop within Animation&Rendering FSM; instead, as discussed above, the Game Loop is a part of larger Animation&Rendering Thread
  - process_input(), instead of processing user input, processes instructions coming from Game Logic FSM
  - update() updates only 3D scene to be rendered, and not the game logic's representation of the game world; all the decision-making is moved at least to the Game Logic FSM, with most of the decisions actually being made by our authoritative server
  - render() works exactly as it worked for a single-player game
- after Animation&Rendering FSM processes input (or lack

"all the decision-making is moved at least to the Game Logic FSM, with most of the

thereof) and returns, Animation&Rendering Thread may conclude Game Loop as it sees fit (in particular, it can be done in any classical Game Loop manner mentioned below)

- then, Animation&Rendering Thread goes back to the very beginning (back to checking if there is anything in its Queue), which completes the infinite Game Loop.

decisions actually being made by our authoritative server

All the usual variations of Game Loop can be used within the Animation&Rendering Thread – including such things as fixed-time-step with delay at the end if there is time left until the next frame, variable-time-step tight loop (in this case a parameter such as elapsed_time needs to be fed to the Animation&Rendering FSM to keep it deterministic), and fixed-update-time-step-but-variable-render-time-step tight loop. Any further improvements (such as using VSYNC) can be added on top. I don't want to elaborate further here, and refer for further discussion of game loops and time steps to two excellent sources: [GafferOnGames.FixYourTimestep] and [GameProgrammingPatterns.GameLoop].

One variation of the Game Loop that is not discussed there, is a simple event-driven thing which you would use for your usual Windows programming (and without any tight loops); in this case animation under Windows can be done via WM_TIMER,[6] and 2D drawing – via something like BitBlt(). While usually woefully inadequate for any serious frames-per-second-oriented games, it has been seen to work very well for social- and casino-like ones.

However, the best thing about our architecture is that the architecture as such doesn't really depend on time step choices; you can even make different time step choices for different platforms and still keep the rest of your code (beyond Animation&Rendering Thread) intact, though Animation&Rendering FSM may need to be somewhat different depending on the fixed-step vs variable-step choice.[7]

"The best thing about our architecture is that the architecture as such doesn't really depend on time step choices; you can even make different time

Animation&Rendering FSM: Running from Game Logic Thread

For some games and/or platforms it might be beneficial to run Animation&Rendering FSM within the same thread as Game Logic FSM. In particular, if your game is a social game running on Windows, there may be no real need to use two separate CPU cores for Game Logic and Animation&Rendering, and the whole thing will be quite ok running within one single thread. In this case, you'll have one thread, one Queue, but two FSMs, with thread code outside of the FSMs deciding which of the

FSMs incoming message belongs to.

However, even in this case I still urge you to keep it as two separate FSMs with a very clean message-based interface between them. First, nobody knows which platform you will need to port your game next year, and second, clean well-separated interfaces at the right places tend to save lots of trouble in the long run.

step choices for different platforms and still keep the rest of your code intact

---

[6] yes, this does work, despite being likely to cause ROFLMAO syndrome for any game developer familiar with game engines

[7] of course, technically you may write your Animation&Rendering FSM as a variable-step one and use it for the fixed-step too, but there is a big open question if you really need to go the variable-step, or can live with a much simpler fixed-step forever-and-ever

Communications FSM

Another FSM, which is all-important for your MMOG, is Communications FSM. The idea here is to keep all the communications-related logic in one place. This may include very different things, from plain socket handling to such things as connect/reconnect logic[8], connection quality monitoring, encryption logic if applicable, etc. etc. Also implementations of higher-level concepts such as generic publisher/subscriber, generic state synchronization, messages-which-can-be-overridden etc. (see Chapter [[TODO]] for further details) also belong here.

"For most of (if not 'all') the platforms, the code of Communications FSM can be kept the same

For most of (if not "all") the platforms, the code of Communications FSM can be kept the same, with the only things being called from within the FSM, being your own wrappers around sockets (for C/C++ – Berkeley sockets). Your own wrappers are nice-to-have just in case if some other platform will have some peculiar ideas about sockets, or to make your system use something like OpenSSL in a straightforward manner. They are also necessary to implement "call interception" on your FSM (see "Implementing Strictly-Deterministic Logic: Strictly-Deteministic Code via Intercepting Calls" section above), allowing you to "replay test" and post-mortem of your Communications FSM.

The diagram of Fig. V.2 shows an implementation of the Communications FSM that uses non-blocking socket calls. For client-side it is perfectly feasible to keep the code of Communications FSM exactly the same, but to deploy it in a different manner, simulating non-blocking sockets via two additional

threads (one to handle reading and another to handle writing), with these additional threads communicating with the main Communications Thread via Queues (using Communication Thread's existing Queue, and one new Queue per new thread).[9]

One more thing to keep in mind with regards to blocking/non-blocking Berkeley sockets, is that getaddrinfo() function (as well as older gethostbyname() function) used for DNS resolution, is inherently blocking, with many platforms having no non-blocking counterpart. However, for the client side in most cases it is a non-issue unless you decide to run your Communications FSM within the same thread as your Game Logic FSM. In the latter case, calling a function with a potential to block for minutes, can easily freeze not only your game (which is more or less expected in case of connectivity problems), but also game UI (which is not acceptable regardless of network connectivity). To avoid this effect, you can always introduce yet another thread (with its own Queue) with the only thing for this thread to do, being to call getaddrinfo() when requested, and to send result back as a message, when the call is finished.[10]

Communications FSM: Running from Game Logic Thread

For Communications FSM, running it from Game Logic Thread might be possible. One reason against doing it, would be if your communications are encrypted, *and* your Game Logic is computationally-intensive.

And again, as with Animation&Rendering FSM, even if you run two FSMs from one single thread, it is much better to keep them separate. One additional reason to keep things separate (with this reason being specific to Communications FSM) is that Communications FSM (or at least large parts of it) is likely to be used on the server-side too.

---

[8] BTW, connect/reconnect will be most likely needed even for UDP

[9] for the server-side, however, these extra threads are not advisable due to the performance overhead. See Chapter [[TODO]] for more details

[10] Alternatively, it is also possible to create a new thread for each getaddrinfo() (with such a thread performing getaddrinfo(), reporting result back and terminating). This thread-per-request solution would work, but it would be a departure from QnFSM, and it can lead to creating too many threads in some fringe scenarios, so I usually prefer to keep a specialized thread intended for getaddrinfo() in a pure QnFSM model

Sound FSM

Sound FSM handles, well, sound. In a sense, it is somewhat similar to Animation&Rendering FSM, but for sound. Its interface (and as always with QnFSM,

interfaces are implemented over messages) needs to be implemented as a kind of "Logic-to-Sound Layer". This "Logic-to-Sound Layer" message-based API should be conceptually similar to "Logic-to-Graphics Layer" with commands going from the Game Logic expressed in terms of "play this sound at such-and-such volume coming from such-and-such position within the game world".

Sound FSM: Running from Game Logic Thread

For Sound FSM running it from the same thread as Game Logic FSM makes sense quite often. On the other hand, on some platforms sound APIs (while being non-blocking in a sense that they return before the sound ends) MAY cause substantial delays, effectively blocking while the sound function finds and parses the file header etc.; while this is still obviously shorter than waiting until the sound ends, it might be not short enough depending on your game. Therefore, keeping Sound FSM in a separate thread MAY be useful for fast-paced frame-per-second-oriented games.

And once again – even if you decide to run two FSMs from the same thread – do yourself a favour and keep the FSMs separate; some months down the road you'll be very happy that you kept your interfaces clean and different modules nicely decoupled.[11]

"Once again – even if you decide to run two FSMs from the same thread – do yourself a favour and keep the FSMs separate

[11] Or you'll regret that you didn't do it, which is pretty much the same thing

Other FSMs

While not shown on the diagram on Fig V.2, there can be other FSMs within your client. For example, these FSMs may run in their own threads, but other variations are also possible.

One practical example of such a client-side FSM (which was implemented in practice) was "update FSM" which handled online download of DLC while making sure that the gameplay delays were within acceptable margins (see more on client updates in general and updates-while-playing in Chapter [[TODO]]).

In general, any kind of entity which performs mostly-independent tasks on the client-side, can be implemented as an additional FSM. While I don't know of practical examples of extra client-side FSMs other than "update FSM" described above, it doesn't mean that your specific game won't allow/require any, so keep your eyes open.

On Additional Threads and Task-Based Multithreading

If your game is very CPU-intensive, and either your Game Logic Thread, or Animation&Rendering Thread become overloaded beyond capabilities of one single CPU core, you might need to introduce an additional thread or five into the picture. This is especially likely for Animation&Rendering Thread/FSM if your game uses serious 3D graphics. While complexities threading model of 3D graphics engines are well beyond the scope of this book, I will try to provide a few hints for those who're just starting to venture there.

As usually with multi-threading, if you're not careful, things can easily become ugly, so in this case:

IIRC
abbv for If I Recall Correctly
— Urban Dictionary
—

- first of all, take another look if you have some Gross Inefficiencies in your code; it is usually much better to remove these rather than trying to parallelize. For example, if you'd have calculated Fibonacci numbers recursively, it is much better to switch to non-recursive implementation (which is IIRC has humongous $O(2^N)$ advantage over recursive one[12]) than to try getting more and more cores working on unnecessary stuff.
- From this point on, to the best of my knowledge you have about three-and-a-half options:
  - **Option A.** The first option is to split the whole thing into several FSMs running within several threads, dedicating one thread per one specific task. In 3D rendering world, this is known as "System-on-a-Thread", and was used by Halo engine (in Halo, they copy the whole game state between threads[GDC.Destiny], which is equivalent to having a queue, so this is a very direct analogy of our QnFSM).
  - **Option B.** The second option is to "off-load" some of the processing to a different thread, with this new thread being just as all the other threads on Fig V.2; in other words, it should have an input queue and a FSM within. This is known as "Task-Based Multithreading" [GDC.TaskBasedMT]. In this case, after doing its (very isolated) part of the job a.k.a. "task", the thread may report back to the whichever-thread-has-requested-its-services. This option is really good for several reasons, from keeping all the FSM-based goodies (such as "replay testing" and post-mortem) for all parts of your client, to encouraging multi-threading model with very few context switches (known as "Coarse-grained parallelism"), and context switches are damn expensive on all general-purpose CPUs.[13] The way how "task off-loading" is done, depends on the implementation. In some implementations, we MAY use data-driven pipelines (similar to those described in [GDC.Destiny]) to enable dynamic task balancing, which allows to optimize core utilization

on different platforms. Note that in pure "Option B", we still have shared-nothing model, so each of the FSMs has it's own exclusive state. On the other hand, for serious rendering engines, due to the sheer size of the game state, pure "shared-nothing" approach MIGHT BE not too feasible.

- **Option B1.** That's the point where "task-off-loading-with-an-immutable-shared-state" emerges. It improves[14] over a basic Option B by allowing for a *very-well-controlled* use of a shared state – namely, *sharing is allowed only when the shared state is guaranteed to be immutable*. It means that, in a limited departure from our shared-nothing model, in addition to inter-thread queues in our QnFSM, we MAY have a shared state. However, to avoid those nasty inter-thread problems, we MUST guarantee that while there is more than one thread which can be accessing the shared state, the shared state is constant/immutable (though it may change outside of "shared" windows). At the moment, it is unclear to me whether Destiny engine (as described in [GDC.Destiny]) uses Option B1 (with an immutable game state shared between threads during "visibility" and "extract" phases) – while it *looks* likely, it is not 100% clear. In any case, both Option B and Option B1 can be described more or less in terms of QnFSM (and most importantly – both eliminate all the non-maintainable and inefficient tinkering with mutexes etc. within your logic). From the point of view of determinism, Option B1 is equivalent to Option B, provided that we consider that immutable-shared-state as one of our inputs (as it is immutable, it is indistinguishable from an input, though delivered in a somewhat different way); while such a game sharing would effectively preclude from applying recording/replay in production (as recording the whole game state on each frame would be too expensive), determinism can still be used for regression testing etc.

- **Option C.** To throw away "replay debug" and post-mortem benefits *for this specific FSM,* and to implement it using multi-thread in-whatever-way-you-like (i.e. using traditional inter-thread synchronization stuff such as mutexes, semaphores, or Dijkstra forbid – memory fences etc. etc.).
  - This is a very dangerous option, and it is to be avoided as long as possible. However, there are some cases when clean separation between the main-thread-data and data-necessary-for-the-secondary-thread is not feasible, usually because of the piece of data to be used by both parallel processes, being too large; it is these cases (and to the best of my knowledge, *only* these cases), when you need to choose Option C. And even in these cases, you might be able to

"If you need Option C for your Game Logic – think

stay away from handling fine-grained thread synchronization, see Chapter [[TODO]] for some hints in this direction.

twice, and then twice more.

- Also, if you need Option C for your Game Logic – think twice, and then twice more. As Game Logic is the one which changes *a damn lot,* with Option C this has all the chances of becoming unmanageable (see, for example, [NoBugs2015]). It is *that* bad, that if you run into this situation, I would seriously think whether the Game Logic requirements are feasible to implement (and maintain) at all.
- On the positive side, it should be noted that even in such an unfortunate case you should be losing FSM-related benefits (such as "replay testing" and post-mortem) only for the FSM which you're rewriting into Option C; all the other FSMs will still remain deterministic (and therefore, easily testable).

- In any case, your multi-threaded FSM SHOULD look as a normal FSM from the outside. In other words, multi-threaded implementation SHOULD be just this – implementation detail *of this particular FSM*, and SHOULD NOT affect the rest of your code. This is useful for two reasons. First, it decouples things and creates a clean well-defined interface, and second, it allows you to change implementation (or add another one, for example, for a different platform) without rewriting the whole thing.

---

[12] that is, if you're not programming in Haskell or something similar
[13] GPGPUs is the only place I know where context switches are cheap, but usually we're not speaking about GPGPUs for these threads
[14] or "degrades", depending on the point of view

On Latencies

One question which may arise for queue-based architectures and fast-paced games, is about latencies introduced by those additional queues (we *do* want to show the data to the user as fast as possible). My experience shows that[15] then we're speaking about additional latency[16] of the order of single-digit microseconds. Probably it can be lowered further into sub-microsecond range by using less trivial non-blocking queues, but this I'm not 100% sure of because of relatively expensive allocations usually involved in marshalling/unmarshalling; for further details on implementing high-performance low-latency queues in C++, please refer to Chapter [[TODO]]. As this single-digit-microsecond delay is at least 3 orders of magnitude smaller than inter-frame delay of 1/60 sec or so, I am arguing that nobody will ever notice the difference, even for single-player or LAN-based games; for Internet-based MMOs where the absolutely best we can hope for is 10ms delay,[17] makes it even less relevant.

In short – I don't think this additional single-digit-microsecond delay can possibly have any effect which is visible to end-user.

---

[15] assuming that the thread is not busy doing something else, and that there are available CPU cores
[16] introduced by a reasonably well-designed message marshalling/unmarshalling + reasonably well-designed inter-process single-reader queue
[17] see Chapter [[TODO]] for conditions when such delays are possible before hitting me too hard

Variations

The diagram on Fig V.2 shows each of the FSMs running within it's own thread. On the other hand, as noted above, each of the FSMs can be run in the same thread as Game Logic FSM. In the extreme case it results in the system where all the FSMs are running within single thread with a corresponding diagram shown on Fig V.3:

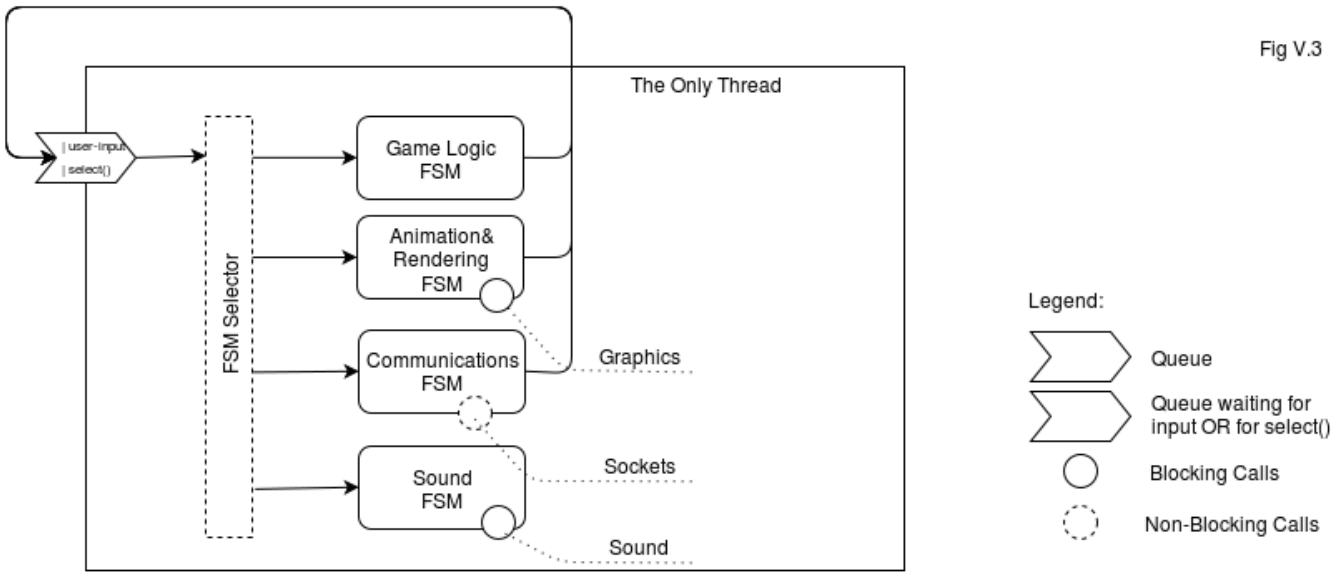
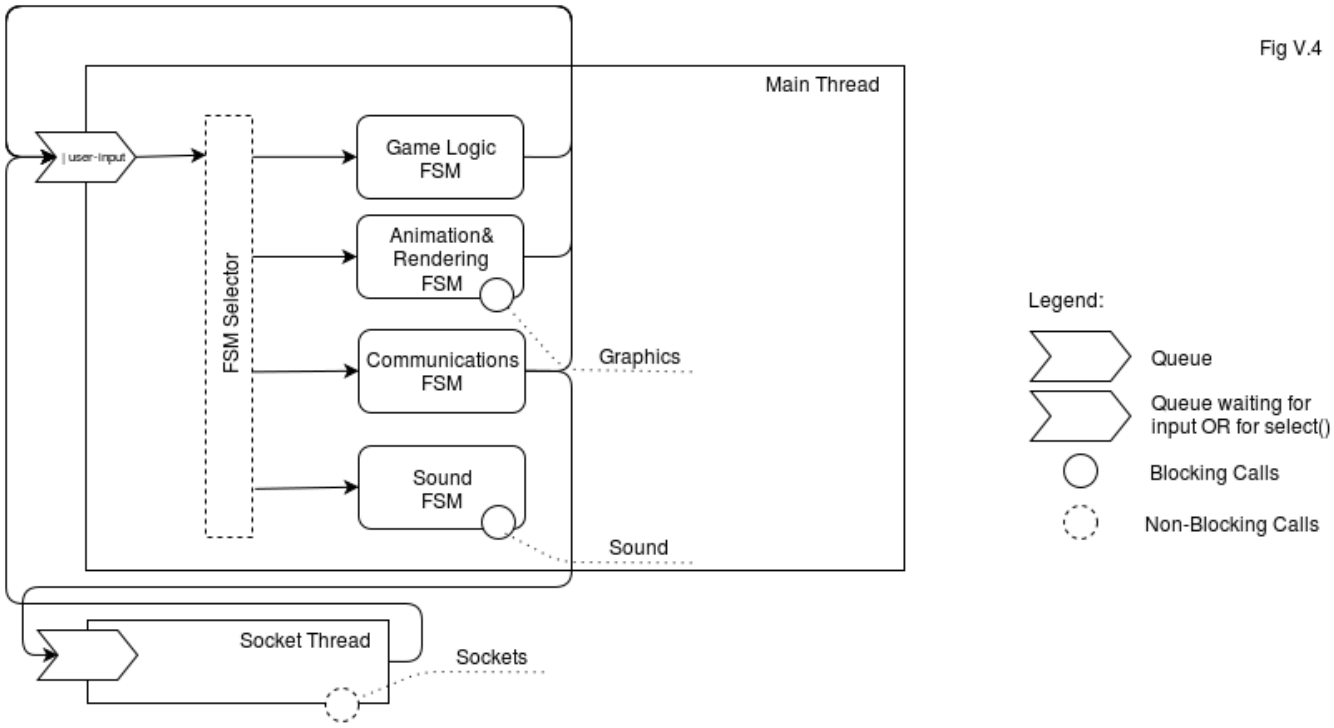


Fig V.3

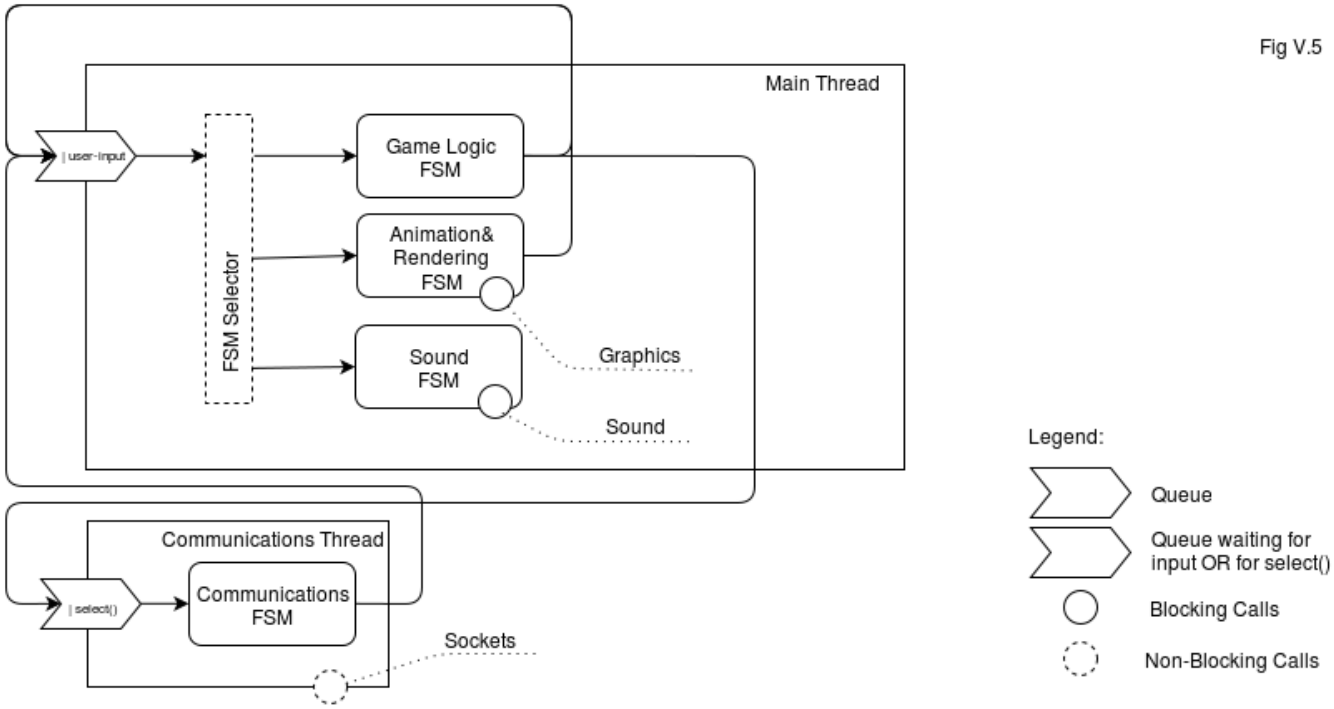
Each and every of FSMs on Fig V.3 is exactly the same as an FSM on Fig V.2; moreover, logically, these two diagrams are exactly equivalent (and "recording" from one can be "replayed" on another one). The only difference on Fig V.3 is that we're using the same thread (and the same Queue) to run all our FSMs. FSM Selector here is just a very dumb selector, which looks at the *destination-FSM* field (set by whoever-sent-the-message) and routes the message accordingly.

This kind of threading could be quite practical, for example, for a casino or a social game. However, not all the platforms allow to wait for the select() in the main graphics loop, so you may need to resort to the one on Fig V.4:



Here *Sockets Thread* is very simple and doesn't contain any substantial logic; all it does is just pushing whatever-it-got-from-Queue to the socket, and pushing whatever-it-got-from-socket – to the Queue of the *Main Thread;* all the actual processing will be performed there, within *Communications FSM.*

Another alternative is shown on Fig V.5:



Both Fig V.4 and Fig V.5 will work for a social or casino-like game on Windows.[18]

On the other end of the spectrum, lie such heavy-weight implementations as the one shown on Fig V.6:

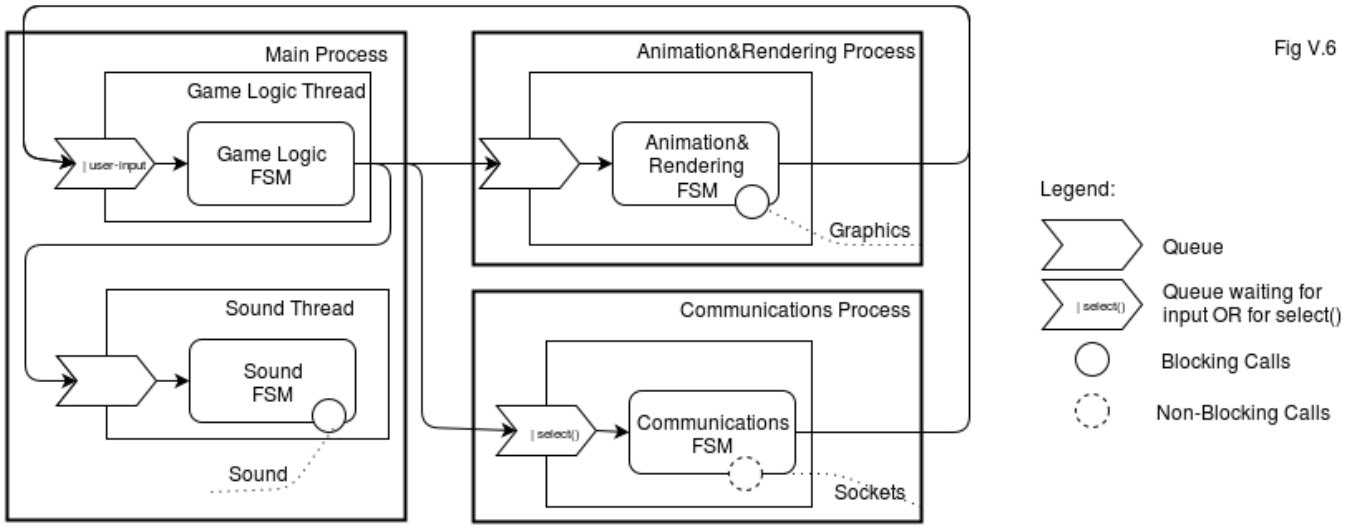


Fig V.6

Here, Animation&Rendering FSM, and Communications FSM run in their own processes. This approach might be useful during testing (in general, you may even run FSMs on different developer's computers if you prefer this kind of interactive debugging). However, for production it is better to avoid such configurations, as inter-process interfaces may help bot writers.

Overall, an exact thread (and even process) configuration you will deploy is not that important and may easily be system-dependent (or even situation-dependent, as in "for the time being, we've decided to separate this FSM to a separate process to debug it on respective developer's machines"). What really matters is that

as long as you're keeping your development model FSM-based, you can deploy it in any way you like without any changes to your FSMs.

In practice, this property has been observed to provide quite a bit of help in the long run. While this effect has significantly more benefits on the server-side (and will be discussed in Chapter [[TODO]]), it has been seen to aid client-side development too; for example, different configurations for different platforms do provide quite a bit of help. In addition, situation-dependent configurations have been observed to help a bit during testing.

---

[18] While on Windows it is possible to create both "|select()" and "|user-input" queues, I don't know how to create one single queue which will be both "|select()" and "|user-input" simultaneously, without resorting to a 'dumb' extra thread; more details on these and other queues will be provided in Chapter [[TODO]]

On Code Bases for Different Platforms

As it was noted above, you MUST keep your Game Logic FSM the same for all the platforms (i.e. as a single code base). Otherwise, given the frequent changes to

Game Logic, all-but-one of your code bases will most likely start to fall behind, to the point of being completely useless.

But what about other FSMs? Do you need to keep them as a single code base? The answer here is quite obvious:

while the architecture shown above allows you to make non-Game-Logic FSMs platform-specific, it makes perfect sense to keep them the same as long as possible

"If your game is graphics-intensive, there can be really good reasons to have your Animation&Rendering FSM different for different platforms

For example, if your game is graphics-intensive, there can be really good reasons to have your Animation&Rendering FSM different for different platforms; for example, you may want to use DirectX on some platforms, and OpenGL on some other platforms (granted, it will be quite a chunk of work to implement both of them, but at least it is possible with the architecture above, and it becomes a potentially viable business choice, especially as OpenGL version and DirectX version can be developed in parallel).

On the other hand, chances that you will need the platform-specific Communications FSM, are much lower.[19] Even if you're writing in C/C++, usable implementations of Berkeley sockets exist on most (if not on *all*) platforms of interest. Moreover, the behavior of sockets on different platforms is quite close from game developer's point of view (at least with regards to those things which we are able to affect).

So, while all such choices are obviously specific to your specific game, statistically you should have much more Animation&Rendering FSMs than Communications FSMs 🙂 .

---

[19] I don't count conditional inclusion of WSAStartup() etc. as being really platform-specific

QnFSM Architecture Summary

Queues-and-FSMs Architecture shown on Fig V.2 (as well as its variations on Fig V.3-Fig V.6) is quite an interesting beast. In particular, while it does ensure a clean separation between parts (FSMs in our case), it tends to go against commonly used patterns of COM-like components or even usual libraries. The key difference here

is that COM-like components are essentially based on blocking RPC, so after you called a COM-like RPC[20], you're blocked until you get a reply. With FSM-based architecture from Fig V.2-V.6, even if you're requesting something from another FSM, you still can (and usually should) process events coming while you're waiting for the reply. See in particular [[TODO!! add subsection on callbacks to FSM]] section above.

From my experience, while developers usually see this kind of FSM-based programming as somewhat more cumbersome than usual procedure-call-based programming, most of them agree that it is beneficial in the medium- to long-run. This is also supported by experiences of people writing in Erlang, which has almost exactly the same approach to concurrency (except for certain QnFSM's goodies, see also "Relation to Erlang" section below). As advantages of QnFSM architecture, we can list the following:

"Most of developers agree that FSM-based programming is beneficial in the medium- to long-run.

- very good separation between different modules (FSMs in our case). FSMs and their message-oriented APIs tend to be isolated very nicely (sometimes even a bit too nicely, but this is just another side of the "somewhat more cumbersome" negative listed above).
- "replay testing" and post-mortem analysis. See "Strictly-Deterministic Logic: Benefits" section above.
- very good performance. While usually it is not *that* important for client-side, it certainly doesn't hurt either. The point here is that with such an architecture, context switches are kept to the absolute minimum, and each thread is working without any pauses (and without any overhead associated with these pauses) as long as it has something to do. On the flip side, it doesn't provide inherent capabilities to scale (so server-side scaling needs to be introduced separately, see Chapter [[TODO]]), but at least it is substantially better than having some state under the mutex, and trying to lock this mutex from different threads to perform something useful.

We will discuss more details on this Queues-and-FSMs architecture as applicable to the server-side, in Chapter [[TODO]], where its performance benefits become significantly more important.

Relation to Actor Concurrency

NB: this subsection is entirely optional, feel free to skip it if theory is of no interest to you

Actor Concurrency Model

The actor model

From theoretical point of view QnFSM architecture can be

seen as a system which is pretty close to so-called "Actor Concurrency Model" (that is, until Option C from "Additional Threads and Task-Based Multithreading" is used), with QnFSM's deterministic FSMs being Actor Concurrency's 'Actors'. However, there is a significant difference between the two, at least perceptionally. Traditionally, Actor concurrency is considered as a way to ensure concurrent calculations; that is, the calculation which is considered is originally a "pure" calculation, with all the parameters known in advance. With games, the situation is very different because we don't know everything in advance (by definition). This has quite a few implications.

Most importantly, system-wide determinism (lack of which is often considered a problem for Actor concurrency when we're speaking about calculations) is not possible for games.[21] In other words, games (more generally, *any distributed interactive system which produces results substantially dependent on timing;* dependency on timing can be either absolute, like "whether the player pressed the button before 12:00", or relative such as "whether player A pressed the button before player B") are inherently non-deterministic when taken as a whole. On the other hand, each of the FSMs/Actors can be made completely deterministic, and this is what I am arguing for in this book.

In other words – while QnFSM is indeed a close cousin of Actor concurrency, quite a few of the analysis made for Actor-concurrency-for-HPC type of tasks, is not exactly applicable to inherently time-dependent systems such as games, so take it with a big pinch of salt.

in computer science is a mathematical model of concurrent computation that treats 'actors' as the universal primitives of concurrent computation: in response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received.

— Wikipedia —

---

[20] also DCE RPC, CORBA RPC, and so on; however, game engine RPCs are usually very different, and you're *not* blocked after the call, in exchange for not receiving anything back from the call

[21] the discussion of this phenomenon is out of scope of this book, but it follows from inherently distributed nature of the games, which, combined with Einstein's light cone and inherently non-deterministic quantum effects when we're organizing transmissions from client to server, mean that very-close events happening for different players, may lead to random results when it comes to time of arrival of these events to server. Given current technologies, determinism is not possible as soon as we have more than one independent "clock domain" within our system (non-deterministic behaviour happens at least due to metastability problem on inter-clock-domain data paths), so at the very least any real-world multi-device game cannot be made fully deterministic in any practical sense.

Relation to Erlang Concurrency and Akka Actors

On the other hand, if looking at Erlang concurrency (more specifically, at *!* and *receive* operators), or at Akka's Actors, we will see that QnFSM is pretty much the same thing.[22] There are no shared states, everything goes via message passing, et caetera, et caetera, et caetera.

The only significant difference is that for QnFSM I am arguing for determinism (which is not guaranteed in Erlang/Akka, at least not without "call interception"; on the other hand, you can write deterministic actors in Erlang or Acca the same way as in QnFSM, it is just an additional restriction you need to keep in mind and enforce). Other than that, and some of those practical goodies in QnFSM (such as recording/replay with all the associated benefits), QnFSM is extremely close to Erlang's concurrency (as well as to Akka's Actors which were inspired by Erlang) from developer's point of view.

Which can be roughly translated into the following observation:

to have a good concurrency model, it is not strictly necessary to program in Erlang or to use Akka

[22] While both Erlang and Akka zealots will argue ad infinitum that their favourite technology is much better, from our perspective the differences are negligible

Akka
is... simplifying the construction of concurrent and distributed applications on the JVM. Akka... emphasizes actor-based concurrency, with inspiration drawn from Erlang.
— Wikipedia —

Erlang
Erlang is a general-purpose, concurrent, garbage-collected programming language and runtime system.
— Wikipedia —

Bottom Line for Chapter V

Phew, it was a long chapter. On the other hand, we've managed to provide a 50'000-feet (and 20'000-word) view on my favorite MMOG client-side architecture. To summarize and re-iterate my recommendations in this regard:

- Think about your graphics, in particular whether you want to use pre-rendered 3D or whether you want/need dual graphics (such as 2D+3D); this is one of the most important questions for your game client;[23] moreover, client-side 3D is not always the best choice, and there are quite a few MMO games out there which have rudimentary graphics
    - if your game is an MMOFPS or an MMORPG, most likely you do need

fully-fledged client-side 3D, but even for an MMORTS the answer can be not that obvious

- when choosing your programming language, think twice about resilience to bot writers, and also about those platforms you want to support. While the former is just one of those things to keep in mind, the latter can be a deal-breaker when deciding on your programming language
  - Usually, C++ is quite a good all-around candidate, but you need to have pretty good developers to work with it
- Write your code in a deterministic event-driven manner (as described in "Strictly-Deterministic Logic" and "Event-Driven Programming and Finite State Machines" sections), it helps, and helps a lot
  - This is not the only viable architecture, so you may be able to get away without it, but at the very least you should consider it and understand why you prefer an alternative one
  - The code written this way magically becomes a deterministic FSM, which has lots of useful implications
  - Keep all your FSMs perfectly self-contained, in a "Share-Nothing" model. It will help in quite a few places down the road.
  - Feel free to run multiple FSMs in a single thread if you think that your game and/or current platform is a good fit, but keep those FSMs separate; it can really save your bacon a few months later.
  - Keep one single code base for Game Logic FSM. For other FSMs, you may make different implementations for different platforms, but do it only if it becomes really necessary.

> **"Write your code in a deterministic event-driven manner, it helps, and helps a lot**

---

[23] yes, I know I'm putting on my Captain Obvious' hat once again here

[[To Be Continued…

This concludes beta Chapter V(d) from the upcoming book "Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)". Stay tuned for beta Chapter VI, "Modular Architecture: Server-Side. Naive and Classical Deployment Architectures.]]

[–] References

[NoBugs2015] 'No Bugs' Hare, “Multi-threading at Business-logic Level is Considered Harmful”, Overload #128
[GameProgrammingPatterns.GameLoop] Robert Nystrom, “Game Programming Patterns”
[GafferOnGames.FixYourTimestep] Glenn Fiedler, “Fix Your Timestep!”, Gaffer On Games
[GDC.Destiny] Natalya Tatarchuk, “Destiny's Multithreaded Rendering Architecture”, GDC2015
[GDC.TaskBasedMT] Ron Fosner, “Task-based Multithreading - How to Program for 100 cores”, GDC2010

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *Chapter V(c). Modular Architecture: Client-Side. On Debuggin..*

Chapter VI(a). Server-Side MMO Architecture. Naïve, Web-Ba... »

Filed Under: Distributed Systems, Programming, System Architecture
Tagged With: client, game, multi-player, Multithreading