Tutorial – Common Patterns

In this tutorial we will examine how applying multithreaded programming patterns to a game engine can influence the frame rates and playability of a game. We will focus on the “Pipeline Thread” pattern that was discussed during the presentation.

To simplify the tutorial, and to prevent you from having to code a full game engine from scratch in order to complete it, you have been provided with a ‘tutorial starter’ that simulates the different systems that may exist in a game engine.

Each of these systems has been given a running time that is approximately equal to the time it would be allocated in a full game engine:

|  |  |
| --- | --- |
| **System Name** | **Execution Time (seconds)** |
| Update Render Data | 0.001 |
| Culling | 0.0025 |
| Renderer | 0.0042 |
| Input | 0.0005 |
| Game Logic | 0.004 |
| Physics | 0.006 |
| Pathfinding | 0.2 (only runs 10% of the time) |

A single-threaded application:

Download and open the ThreadingPatterns project from portal.

If you compile and run the program, you will see the console displaying information on how often each system is currently running (and the resulting frame rate). Note that these values are the average time taken over 1 second.

At the moment, each value is exactly the same. This is because each system is running in a linear fashion, and cannot run again while the other systems are finishing executing.

You should also notice that the time fluctuates a large amount – this is the pathfinding system having a big impact on the overall average frame rate. In a real game, this sort of fluctuation would make the game unplayable (and unwatchable!)

Overall, the average frame-rate seems to hover between 20-30 frames per second, with the occasional spike in either direction.

If you navigate to the single\_threaded\_application class, and look at the tick() function, you can see how each system is currently executed:

/\* @brief Ordered custom tick() implementation.

\*/

void single\_threaded\_application::tick()

{

// Game mechanics

m\_input.execute();

m\_game\_logic.execute();

m\_physics.execute();

m\_pathfinding.execute();

//Must be called before rendering can occur!

m\_update\_render\_data.execute();

// Rendering

m\_culling.execute();

m\_renderer.execute();

}

Our task is to now break this execution into separate threads, to allow independent systems to Execute at the same time.

A multi-threaded application:

To start with, create a new multithreaded\_application class that derives from the base\_application class. You will need to remember to override the tick() function, though leave it empty for now.

To begin with, we will begin by breaking our execution into two threads:

**A Graphics processing thread**

Runs the culling and rendering systems. Because we always need to cull before we render, it makes sense to run the two systems together

**A Logic processing thread**

Runs the input, game logic, physics and pathfinding systems, as well as the update render data system once all logic processing has been completed. We could break this up a little more later if we needed to.

While this will work well, we also need to make sure that the Graphics processing thread does not execute while the render data is in the process of being updated. While it won’t do much in this example code, it would cause all sorts of issues (and crashes) if it was done in a full implementation.

Also, remember that we already have a thread to work with – the main thread that starts when the application does. For this example, we’ll use this as the **Logic processing thread**, and create a new thread to be our **Graphics processing thread.**

Because our main() thread will run the logic code, we can put call execute() on all the object systems in our “tick()” function:

void multithreaded\_application::tick()

{

// Game mechanics

m\_input.execute();

m\_game\_logic.execute();

m\_physics.execute();

}

If we update the main() to use the multithreaded\_application and run the program, you should see those systems being executed as expected and -1 for the disabled systems.

Next, we’ll create a thread that runs the graphics systems.

First, add a new thread member variable to our multithreaded\_application class. Remember you will need to include <thread> to be able to access the the std::thread class

private:

*std*::*thread* m\_renderThread;

We can now spawn a new thread in our constructor, and use a lambda which loops endlessly and executes our graphics functions:

multithreaded\_application::multithreaded\_application()

{

m\_render\_thread = std::thread([&]

{

while (true)

{

m\_culling.execute();

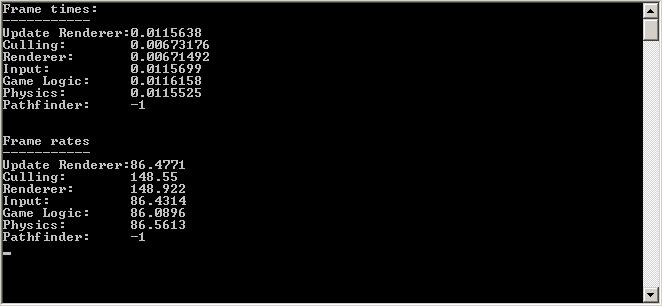
m\_renderer.execute();

}

});

}

If you run the program again, you should now see the culling and renderer systems are being updated – and running considerably faster than the logic systems.



As mentioned above though, we still have a problem that would break an actual engine – we are updating our render data and rendering at the same time!

To stop this, we will use a mutex to prevent the graphics thread from executing while the update render system is executing.

First, add a new mutex to the multithreaded\_application class. You will need to include <mutex> for the std::mutex class to be visible:

*std*::*mutex* m\_render\_data\_mutex;

We will then lock/unlock the mutex each time we render:

**m\_render\_data\_mutex.*lock*();**

m\_culling.execute();

m\_renderer.execute();

**m\_render\_data\_mutex.*unlock*();**

We then also need to place a lock around the Update Render data execute:

m\_render\_data\_mutex.*lock*();

m\_update\_render\_data.execute();

m\_render\_data\_mutex.*unlock*();

Run the application now – you should very quickly see that **something is going wrong** – while our graphics thread still seems to be updating correctly, the logic thread is running much slower than before.

We have an unforeseen **race condition** – the graphics thread is unlocking and locking the mutex so quickly that the logic thread doesn’t have a chance to get a lock – it becomes a race between who can lock the mutex first, and the renderer thread is winning most the time.

To prevent this from happening, we need to let the graphics thread know that the main thread is trying to get a lock, and to not re-lock just yet. This will give the logic thread priority in gaining the lock, and prevent it from getting stuck.

First, we’ll create a new **atomic** bool. Remember them as variable that is safe to read and write from without having to wrap it in a mutex first.

Start by including the <atomic> header, then add the following member variable to your class:

*std*::*atomic*<bool> m\_waiting\_to\_write;

Initialize its value to false in the constructor.

multithreaded\_application::multithreaded\_application() : m\_waiting\_to\_write(false)

In our graphics thread lambda, we need to update the code to check the variable to make sure we only lock when it isn’t set to true:

m\_render\_thread = *std*::*thread*([&]

{

while (true)

{

**if (!m\_waiting\_to\_write)**

**{**

m\_render\_data\_mutex.*lock*();

m\_culling.Execute();

m\_renderer.Execute();

m\_render\_data\_mutex.*unlock*();

**}**

}

});

And finally, we can set and unset the variable in our “tick()” function:

m\_waiting\_to\_write = true;

m\_render\_data\_mutex.*lock*();

m\_waiting\_to\_write = false;

m\_update\_render\_data.execute();

m\_render\_data\_mutex.*unlock*();

If we run the application again, everything should be executing properly – there is still quite a bit of fluctuation due to the pathfinding system, but it should be running at a reasonable speed (60+ frames)

Try to add a new thread yourself to handle the pathfinding system and separate it from the rest of the logic, and what effect it has on the execute times of the other systems.