

asynchronous robotics programming

(ASYNCHRONOUS PROGRAMMING WITH COROUTINES IN C++)

JAVIER BLAZQUEZ

GOOD RESPONSIVENESS IS A DESIRABLE PROPERTY FOR MANY APPLICATIONS, BUT IT IS especially important in games, where reacting immediately to the player's input is paramount, and dropping frames is unacceptable. Thus, any long process or computation—such as file I/O—should be done asynchronously if at all possible.

However, the traditional model for asynchronous programming requires structuring code in a different way than for regular sequential code, complicating certain kinds of algorithms significantly. Since C++ doesn't provide any built-in constructs for dealing with asynchronous operations, game programmers have to rely on their operating system APIs, which typically provide only low-level mechanisms, such as handles or completion callbacks.

This article presents a method for simplifying asynchronous operations considerably through the use of coroutines.

PROGRAMMING ASYNCHRONOUSLY, THE HARD WAY

» Non-blocking I/O operations are commonly used in asynchronous programming, so let's use them as an example. When dealing with asynchronous I/O, the typical scenario is to initiate a non-blocking operation and then defer any processing on the data until it becomes available. To know when the data is available, we can wait for a callback or query the handle returned by the asynchronous API.

There are several drawbacks to both options. Relying on a callback forces us to artificially partition the code into multiple functions that perform only part of the computation. When we start an asynchronous request in the middle of an algorithm, we have to pass a pointer to the function that will perform the rest of the computation, and then return from the function immediately, basically delaying the rest of the algorithm while the data is in flight. Once the data is available, the new function will be called to continue the computation. If that function in turn performs a non-blocking operation, it has to similarly defer the rest of the code to another function and pass it as a callback. This approach of chaining functions together can quickly make the code unmanageable and difficult to understand.

Callbacks also have the problem of potentially requiring the code to be thread-safe, since they could fire at any time from another thread. On certain platforms, the operating system invokes these callbacks from the background thread that performs the I/O operations, forcing us to protect any shared data that our callback function might access. On other platforms, like Win32, the completion callbacks are called in the context of the thread that initiated the request, solving the multithreading problem, but they can only run when the thread is in an alertable wait state. If we are not careful to enter this state regularly, we could be delaying the reception of callbacks for a long time.

On the other hand, when polling regularly for completion, we are usually required to make the code maintain some kind of state to know what pieces of data we have processed already and which ones we are still waiting for, effectively turning our routine into a state machine. Again, this approach usually obscures the intent of the algorithm, and has us jumping through hoops to gather the data we need.

To illustrate the point, consider an entity class that represents an object in the world. These entities contain a reference to their 3D model and a list of textures. When loading an entity, we might have some kind of index file that describes it, containing the names of the

asynchronous programming

model and texture files that it uses. To create an entity, we start by reading the index file, extracting the file names of the other resources, and then loading the data from those files.

To load an entity asynchronously, we would need to write a function similar to that in Listing 1. This function needs to be called regularly (once every frame, for instance) to query the status of the pending asynchronous requests. Note that the function has to not only maintain a notion of current state, but also use member variables to hold the intermediate data used throughout the load, since it will be called over several frames and would otherwise be lost. This example is very simple and could be improved by tracking multiple requests at the same time, but it's meant to illustrate the problem of dealing with asynchronous requests that depend on the results of other requests.

The resulting code is far from ideal, since it turns simple sequential code into a switch statement, which maintains many pieces of state. It also makes error handling especially difficult should a problem arise on an intermediate step.

Given these drawbacks, it's worthwhile to consider alternatives to the traditional asynchronous model. I will show how to use coroutines to avoid dealing with these kinds of callback chains or state machines in the code.

WHAT ARE COROUTINES?

» Coroutines are basically subroutines that can be suspended during their execution and resumed at the same point later. While subroutines can only give control back to the calling code by returning, coroutines can yield at any point, effectively pausing their computation temporarily. When yielding, coroutines capture the values of local variables, which become available again and can be used normally when execution resumes.

Coroutines are first-class citizens in some languages, but C++ does not support them directly. However, many operating system APIs offer similar constructs, like fibers on Win32 and user contexts on POSIX.

The way these implementations work is by basically performing a context switch, much like what happens when a new thread is scheduled by the operating system. In essence, this context switch just copies out the value of the CPU registers (including the stack pointer) and restores the values for the new coroutine, resuming at the point where it last yielded execution. And because the stack pointers are switched, the resuming coroutine can access its local variables normally. Note that, unlike a real thread switch, this kind of context switch doesn't involve a call into the operating system kernel, making it faster.

For most cases, it's best to use the coroutine APIs provided by the operating system, which are usually quite lightweight. Implementing a homegrown context switching logic can be very tricky on certain platforms. It can be tempting to build a custom system so that we can, for instance, control where the stack for each coroutine is allocated or track its lifetime—however,

Listing 1

```
bool Entity::UpdateLoad()
{
    switch (mState)
    {
        case InitialState:
            mFile = File(mIndexFile);
            mRequest = mFile.Read(&mIndex, sizeof(mIndex));
            mState = ReadingIndex;
            break;

        case ReadingIndex:
            if (!mRequest.IsComplete()) break;

            // The index has finished loading, start reading model
            mFile = File(mIndex.mModelIndexFile);
            mModel = std::malloc(mIndex.mModelIndexSize);
            mRequest = mFile.Read(mModel, mIndex.mModelIndexSize);
            mState = ReadingModel;
            break;

        case ReadingModel:
            if (!mRequest.IsComplete()) break;

            // The model has finished loading, start reading textures
            mTexIndex = -1;
            mState = ReadingTextures;
            break;

        case ReadingTextures:
            if (!mRequest.IsComplete()) break;

            if (++mTexIndex == mIndex.mTextureCount) {
                mState = Finished;
                break;
            }

            mFile = File(mIndex.mTextureFiles[mTexIndex]);
            mTextures[mTexIndex] = std::malloc(mIndex.mTextureSizes[mTexIndex]);
            mRequest = mFile.Read(mTextures[mTexIndex],
                                mIndex.mTextureSizes[mTexIndex]);

            break;
    }

    return mState == Finished;
}
```

Listing 2

```
size_t File::Read(void* buffer, size_t size)
{
    OVERLAPPED overlapped = { 0, 0, { mOffset, 0 }, NULL };
    ReadFile(mHandle, buffer, size, NULL, &overlapped);

    while (!HasOverlappedIoCompleted(&overlapped))
    {
        ResourceManager::Yield();
    }

    DWORD bytesRead;
    GetOverlappedResult(mHandle, &overlapped, &bytesRead, FALSE);

    mOffset += bytesRead;
    return bytesRead;
}
```

one of the properties of a stack is that it grows automatically when the current call chain requires more storage. Because this process is usually controlled by the operating system, it's difficult for the application to hook in its own logic here. We would need to use guard pages or a similar mechanism to capture accesses past the end of the stack that must trigger an extension.

Also, allocating the stack manually can confuse the operating system and subtly break certain constructs. For instance, on Xbox 360, exception handling doesn't work properly for code that runs on a user-allocated stack. If the stack is not allocated by the operating system, then it refuses to run the exception handlers that have been set in the code. Exceptions can be thrown, but they cannot be caught. Even if you don't actually use exceptions in your game, there are certain operating system APIs that rely on them, and they would cease to function properly. `ReleaseSemaphore` is an example of a function that internally relies on exceptions for handling certain cases.

ENTER FIBERS

» The entity loading code presented earlier can be simplified substantially through the use of coroutines implemented on Win32 using fibers. Fibers are defined as lightweight units of execution that run cooperatively, that is, they are not preempted by the operating system but rather have to yield execution explicitly. Although I show here what an implementation for Windows and Xbox 360 could look like, the concepts are transferable to other platforms.

Consider a `ResourceManager` class that centralizes the creation and loading of resources such as an entity. We would call a `LoadResource<R>` function to kick-start the creation of a resource given its type `R` and the path to its index file. This would create a fiber that, when scheduled, will call a static `Create` function on our resource, which contains the logic for loading and instantiating that particular resource. Note that on a synchronous implementation, we would call the `Create` function immediately as part of the `LoadResource` function, and the function would block until the resource has been loaded, returning a pointer to it.

However, in the asynchronous case, the `LoadResource` function just creates the fiber and returns immediately. Since the resource doesn't exist yet, we cannot return a pointer to it. We have to return some kind of handle or descriptor that will let the client retrieve the actual resource when it's ready. The handle could be as simple as an integer ID that can be resolved later to retrieve the resource pointer. A more interesting handle type would be some kind of strongly-typed proxy object that can be queried to see if the resource is available, as well as provide reference counting semantics and regular pointer syntax for accessing the resource.

Listing 3

```
Entity* Entity::Create(const char* filename)
{
    // Load index
    File indexFile(filename);

    EntityIndex index;
    indexFile.Read(&index, sizeof(index));

    // Load model
    File modelFile(index.mModelFile);

    void* modelData = std::malloc(index.mModelSize);
    modelFile.Read(modelData, index.mModelSize);
    Model* model = new Model(modelData);

    // Load textures
    std::vector<Texture*> textures(index.mTextureCount);
    for (size_t i = 0; i < index.mTextureCount; ++i)
    {
        File textureFile(index.mTextureFiles[i]);

        void* textureData = std::malloc(index.mTextureSizes[i]);
        textureFile.Read(textureData, index.mTextureSizes[i]);
        textures[i] = new Texture(textureData);
    }

    return new Entity(model, textures);
}
```

After requesting the creation of one or more resources, we would need to call an `Update` function on the resource manager at some regular interval, for example, every frame. This `Update` function is responsible for processing the pending resource loads, and it does so by basically scheduling the existing fibers in turn, giving them a chance to execute part of the resource creation code. Note that the resource manager cannot preempt the running fiber, so it has to wait until it yields explicitly.

The way each update step works is as follows. First, we call `ConvertThreadToFiber` so that we can start scheduling fibers on the current thread. Next, we switch to the first fiber on the list. The fiber will then resume execution at the point where it last yielded. If it is a newly created fiber, it will start at the main fiber routine, which will immediately call the static `Create` function on the resource to run the loading logic for that resource. Eventually, this function will either return

Coroutines are first-class citizens in some languages, but C++ does not support them directly. However, many operating system APIs offer similar constructs, like fibers on Win32 and user contexts on POSIX.

(meaning that the load is complete and the resource is ready) or yield by calling a `Yield` function on the resource manager, allowing the next fiber to be scheduled. Note that fibers don't return from their main routine—they just mark themselves as finished and yield one last time, which effectively prevents the fiber from being scheduled again in the future.

The manager keeps activating the next available fiber until the last one yields, after which the `Update` function cleans up by calling `ConvertFiberToThread` and then returns. The call to `Update` is therefore a single pass over the list of fibers on which they are given an opportunity to progress and maybe finish the load process.

This description might seem to imply that the resource loading code must have explicit knowledge of fibers and yield manually at certain points during its execution. This is not necessarily the case, and I will show how this behavior can be abstracted so it doesn't introduce such requirements for the loading code.

asynchronous programming

SEQUENTIAL ASYNCHRONY

» The key enabling feature of fibers here is the opportunity for resource loading code to be written in a sequential manner even though its operations are all asynchronous. Since fibers can yield at any point and be resumed later, they can effectively block on an asynchronous I/O operation until it has completed. Note that this doesn't imply blocking the whole thread of execution—other code can and will run in the meantime on the same thread.

The behavior can be encapsulated in a simple fiber-aware `File` class that presents a synchronous I/O API to the resource loading code but performs asynchronous operations under the hood. For the user, a call to the `Read` function appears to block until the data is available, and indeed it will only return when the data has been read. In the meantime, it will yield to give other fibers a chance to execute, and it will keep doing so until it has determined that the asynchronous request has completed. See Listing 2 for the implementation of `File::Read`.

With this infrastructure in place, we can now write the code for loading entities in a much more straightforward way, as in Listing 3. The new version of the code resembles very much what one would write for loading data synchronously. The code relies on the `File` object to perform the reads, allowing it to be completely oblivious to the fact that it's actually running on a fiber and being scheduled out every time it performs a read.

Additionally, this code can be run unchanged for performing both synchronous and asynchronous resource loads, if we want to give this option to users of the resource manager. All we would need to do is make the `ResourceManager::Yield` function check whether it's being called from outside of a fiber. If it is, it will yield the thread by calling `Sleep(0)`, effectively turning reads into blocking operations.

Another advantage of fibers is that they all execute in the context of a single thread. They do not run concurrently with any other code that executes on that thread. Therefore, the resource loading code can freely access any other data or systems without having to make them thread-safe necessarily. Also, the call to `Update` on the resource manager happens at a predefined place in the game loop, making it easier to reason about

the state of other objects at that point. Compare this with the non-deterministic behavior of callbacks that can be invoked at any time on a different thread or whenever the thread happens to enter an alertable wait state.

PAUSE AND CANCELLATION

» One of the interesting advantages of performing resource loads using fibers is that they can be interrupted at certain points, meaning we don't have to wait for the whole process to be completed in a single call. We can leverage this feature to easily add support for pausing and canceling resource loads, which can be valuable for systems that deal with many resources at the same time and must prioritize or discard some of them for memory, performance, or game-related reasons.

Implementing pause is straightforward. We can extend the resource manager interface with two functions to support pausing and resuming ongoing resource loads. Considering that the resource loading code will yield at some point (when performing a read, for example), we can use that opportunity to check whether the user has requested the fiber to be paused. If this is the case, we can mark the fiber as paused and avoid scheduling it again until somebody requests it to be resumed. Hence, we can effectively delay a certain operation indefinitely.

On the other hand, to properly support cancellation we have to extend the resource loading code so that it regularly checks for this condition. We could provide a function that the loading code can call to check whether it has been canceled and is supposed to return. This function could also act as a checkpoint by yielding execution so that the fiber doesn't run for too long when performing a complex operation. By inserting calls to this function throughout the code, the resource loading function can further improve the responsiveness of the game by reducing the maximum amount of time that the main thread is blocked. These calls can be thought of as preemption points for the fiber.

To implement cancellation support, the code must check the return value of the function and act accordingly. If it determines that it has been canceled, it must clean up all intermediate resources and return a null pointer from the `Create` function. Note that just removing the fiber from the list of running fibers when it has been canceled is not a proper solution because it prevents the resource loading code from cleaning up any intermediate objects, resulting in memory leaks.

An alternative solution would be to throw an exception when a fiber yields and we detect that it has been marked as canceled, and then catching that exception on the main fiber routine. This solution would force the resource loading function to return, cleaning up all intermediate resources as long as the user has properly followed the Resource Acquisition Is Initialization (RAII) pattern, which ensures that resources are cleaned up when they go out of scope by releasing them in the destructor of stack objects. The advantage of this alternative is that it doesn't require the user to add explicit checks for cancellation throughout the code, but relies instead on the proper application of the RAII pattern and having compiled support for exceptions, which is commonly disabled for games.

THE COMMENDABLE COROUTINE

» I have shown how to leverage coroutines with little effort to simplify code that deals with asynchronous processes, such as file I/O. The resulting code is much easier to understand since it retains its sequential nature and avoids multi-threading issues that can happen when we rely on callbacks.

There are several drawbacks to this approach, of course. The main problem is that coroutines are collaborative by nature. They have to voluntarily yield execution for other coroutines to have a chance to run. Since there is no preemption, a coroutine that's behaving badly can hold the thread of execution indefinitely, a problem that can't happen with the common multi-threaded approach, since the operating system scheduler makes sure that all threads have a chance to run (unless they have different priorities).

There's also a risk of introducing too much latency on I/O operations or reducing their throughput considerably, seeing as coroutines are scheduled out as soon as they initiate an I/O operation, which could complete long before the coroutine is scheduled again. This risk can be partially mitigated by scheduling the coroutines more frequently and performing certain I/O operations synchronously when they are deemed small enough not to block for a significant amount of time 🕒

JAVIER BLAZQUEZ works as a core engineer at Lucas Arts, where he deals mainly with systems-level libraries for the in-house game engine. Previously he worked at Pyro Studios, where he was responsible for the overall game architecture implementation. Email him at jbblazquez@gdmag.com.

resources

Stackless Python

www.stackless.com

Hawes, David, "Snakes on a Seamless Living World," *Game Developer*, February 2009

Richter, Jeffrey, "Concurrent Affairs" series, <http://msdn.microsoft.com/en-us/magazine/cc501041.aspx>