Efficient, Event-Based Simulations

David "Rez" Graham

1 Introduction

Every game has a finite amount of time to spend for updating the frame during the main game loop. Games are often required to have minimum framerates of 60 fps. This leaves you a measly 16.7ms in which to update all positions, run physics, render everything to the screen, process all AI agents, and do any other housekeeping you need to do every frame. You have quite a bit of help from the hardware, with GPUs and multicore CPUs being standard even on lower-end machines, but this often isn't enough to run the simulation at the fidelity we would prefer.

This chapter will discuss one technique for turning those per-frame updates into an event-driven update system so that you're only truly simulating what you have to, when you have to. The rest is mathematically derived on demand.

2 Motivation

The most common method for updating game objects is to call an update function every frame, like Listing 1.

Listing 1 Typical update function

```
void Update(float deltaMs)
{
    for (SimulationObj& simObj : simObjects)
    {
        simObj.Update(deltaMs);
    }
}
```

This is perfectly suitable for cases in which it is used on only a small number of objects, but it can quickly get very expensive for objects that truly need to update every frame. For example, you might have a sensor system on enemies that detects for the presence of the player. It needs to cast rays to simulate vision and check for sound-producing objects in a radius, etc. This is fine for small numbers of objects, but it quickly gets very expensive.

One solution to this problem would be to time-slice the updates so that you only process part of the `simObjects` list each frame. This allows you to spread out the processing, which is good at reducing spikes. Implementing this solution is rather straight-forward, as can be seen in Listing 2. The function will continuously update objects

until it either runs out of time or it runs out of objects, the former determined by the `kTimeSlice` constant.

Listing 2 A time-sliced update function

```
void TimeSlicedUpdate(float deltaMs)
{
    constexpr float kTimeSlice = 2.f;  // 2 ms
    static size_t index = 0;  // persist across calls
    float startTime = GetCurrentTime();

    // Update all object until we either run out of time, or
    // we reach the end of the list.
    while (GetCurrentTime() + startTime < kTimeSlice &&
           index < simObjects.size());
    {
        simObjects[index].Update(deltaMs);
    }

    // Reset the index if necessary.
    if (index >= simObjects.size())
        index = 0;
}
```

This solution has the advantage of being easy to implement. This may be perfect for your use-case, yet there are several issues that must be pointed out. The first and perhaps most obvious is that updates will be inconsistent. For example, if you have a torch you need to update, it's possible that it will last longer than intended. If it's only a few frames it doesn't matter, but this becomes a huge issue if you have more expensive processes. In the extreme case, you can suffer from starvation.

A more insidious issue is that you are incurring a cost for every object in the list. Time-slicing doesn't magically solve performance issues; all it does is spread out those issues across multiple frames. If your frames are consistently too long – a good bet if you're having issues with per-frame updates – than time-slicing will only hurt performance. Time-slicing reduces spikes; it doesn't improve overall framerate.

Often times, you will have updates that really don't need the attention of the CPU every frame. For example, think about the update function on the torch from the previous example. If all it needs to do is wait until it's spent, it will essentially be busy-waiting until that time has elapsed, as shown in Listing 3.

Listing 3 Torch busy-wait

```
void Torch::Update(float deltaMs)
{
    if (m_lifeSpan > 0)
```

```
    {
        m_lifeSpan -= deltaMs;
        if (m_lifeSpan <= 0)
            TurnOffTorch();
    }
}
```

The vast majority of its time, the torch is running a test, subtracting a value, and then running another test. What if there are 100 torches in the game? How much time will be wasted just updating torches and not doing any real work?
Finally, time-slicing is rarely as simple as splitting up the processing of a list. Most objects can be created and destroyed during the main game loop, therefore the population of `simObjects` will likely change while the loop is in the middle of processing. It's important that you account for the list changing out from under you as a result. Furthermore, many AI systems have a number of object dependencies that will complicate this even more. The torch example is simple, but this kind of busy-wait anti-pattern can be seen in various game development situations. For example, loot drops that disappear, timed quests, NPC AI schedule updates, items like food that decay in the world and weather events can all be affected.

3 Event-Based Updates

Most of the time, the simulation is just updating a time value and checking to see if it's time to do the actual work. That means we're doing $O(n)$ updates, where n is number of `simObjects` in the list. We can reduce the value of n by shoving all time-based simple updates into a priority queue based on when they need to be updated. Every frame, we look at the front of the queue and check to see if it's time to call that update. If it is, we call it, pop the update, and move to the next one. If not, we're done and don't even have to look at any other items nor update any other time variables.

This will work for the 100 torches in the game, but what about something more complex? Say you have an enemy warlock that has magic points which are constantly recovering over time. As the warlock casts spells, these magic points deplete. This depletion of available magic acts a cooldown on the warlock's power. The simple implementation would be to update the warlock's current magic points every frame based on the delta time and then run the AI tick, which would choose the appropriate spell to cast based on current magic points and the world state. The problem is that this is not a simple cooldown, rather there are likely several spells all tied to the same value.

A more optimal solution would be to stop thinking about updates as per-frame incremental changes. Instead, think of the updates as discrete events that occur along a linear timeline. This timeline is in turn represented by the stat we're tracking. In the torch example, there is only a single event, which is when the lifespan of the torch reaches 0. We can trivially calculate how long it will be until that event occurs.

The warlock's magic points are the same, except that there are several more events

that need to occur to accurately simulate their recovery. Let's say the warlock has three spells: fireball, lightning, and ice. This could be translated into three events, each one signaling that a particular spell is ready to be cast. Figure 1 shows how this might look on a timeline. The 'X' marks the position or status of the warlock's current magic points. In this example, we would need three items in the priority queue, one for each spell. The ice event would trigger in 15 seconds, the fireball event would trigger in 30 and the lighting event would trigger in 45 seconds. The triggering of these events signifies that that particular spell is ready to cast.

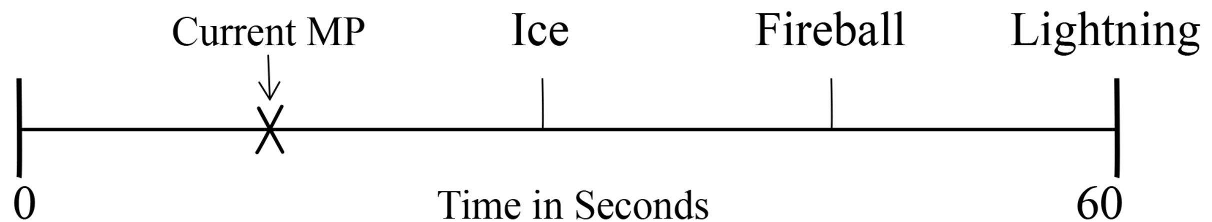

Figure 1 Warlock spells as a timeline

The reason I chose time in Figure 1 rather than the value of magic points is because it's possible for to have non-linear mappings from value to time. For example, the warlock's magic point regeneration might follow a logarithmic curve. It doesn't matter for our purposes, as long as we can calculate the amount of time until an event needs to be triggered.

4 Implementation

The first step is to encapsulate all of the data we need to represent a single callback, which you can see in Listing 4.

Listing 4 DelayedCallback struct

```
struct DelayedCallback
{
    typedef std::function<void()> Callback;

    unsigned int m_id;
    Callback m_callback;
    double m_timeToCall;
};
```

`m_id` is a unique id, which is used to reference this particular delayed callback in case it needs to be removed or updated. `m_callback` is the callback itself, which can be any kind of callable that conforms to the signature defined in the template. This is often a

lambda or functor that references the appropriate object. Finally, `m_timeToCall` is the game time for when this callback needs to be called. It's worth noting that this is a simplified version of the real struct. I have omitted things like constructors & assignment operators for clarity.

The next step is to create a class for managing these delayed callbacks. The class definition is shown in Listing 5.

Listing 5 DelayedCallbackManager class

```
class DelayedCallbackManager
{
    // This vector is treated like a heap.
    std::vector<DelayedCallback> m_callbacks;
    double m_currTime;

public:
    void Update(double deltaMs);

    unsigned int AddCallback(
        DelayedCallback::Callback&& callback,
        double delayUntilCallMs);

    bool RemoveCallback(unsigned int id);
    void ChangeTime(unsigned int id, double newTime);

private:
    // comparison function for the priority queue
    static bool HeapCompare(
        const DelayedCallback& left,
        const DelayedCallback& right)
    {
        return left.m_timeToCall > right.m_timeToCall;
    }
};
```

Again, I have omitted most of the boilerplate. At the top is the `m_callbacks` array, which is treated as a heap. This is the priority queue of all callbacks. The reason this is a vector instead of a `std::priority_queue` is to make it easier to manipulate. For example, we can't get into the guts of a `priority_queue` in order to remove a callback from the middle of the queue. With an array, we can delete anywhere we want and fix up the heap properties manually (Cormen 2009).

The API has an update function which is called every frame to update the current time and check to see if any callbacks need to be called. There are also functions for adding and removing callbacks, as well as a function to change the time of an existing callback.

Finally, we have the comparison function used for the heap.

The `Update()` function is very simple and is shown in Listing 6. It starts by updating the current time, and then it looks at the root of the heap and checks to see if it's time to update that callback. If it is, it calls the callback and removes it. It will continue looping until the heap is empty or the root item is not ready to update.

Listing 6 DelayedCallbackManager update

```
void DelayedCallbackManager::Update(double deltaMs)
{
    m_currTime += deltaMs;

    while (!m_callbacks.empty() &&
           m_currTime >= m_callbacks.front().m_timeToCall)
    {
        m_callbacks.front().m_callback();
        std::pop_heap(m_callbacks.begin(),
            m_callbacks.end(), &HeapCompare);
        m_callbacks.pop_back();
    }
}
```

All that's left is to put it all together. When the warlock is initialized, it needs to calculate how long it will take to reach any of the milestones along its track. This is shown in Figure 1. Each of those milestones is then submitted as an event to be called once it reaches that milestone. If a spell is cast by the warlock, it will go through each milestone and change the time or remove it as necessary. This recalculation can be seen in Listing 7.

Listing 7 Warlock DelayedCallback recaluclation

```
float Warlock::GetDelayedCallbackTime(const Spell& spell)
{
    double currTime = GetCurrentGameTime();
    float deltaMp = spell.GetCost() - m_magicPoints;

    if (deltaMp > 0)
        return deltaMp * kMagicPointRegenTime;
    else
        return 0;  // spell can be cast now
}
```

5 Conclusion

Simulation updates are costly. Sometimes you have long updates, but more often what happens is that you get nickel and dimed by many useless updates. This system allows you

defer those updates so you're not wasting CPU time on adding a delta to a time variable and testing if it's time to do the interesting work. That having been said, there are several improvements that can be made to the existing system.

Currently, callbacks are removed after they've been called. However, repeating callbacks are also possible. It would be trivial to add a bool or enum that flags a callback as repeatable so that when it's called, it gets reinserted into the heap.

Another modification would be to link the owner of the callback to the `DelayedCallback` object itself, allowing the manager to access the owner through an observer. This would allow the manager to notify the owner when the callback has been removed. This isn't necessary for the basic implementation, but if you want multiple systems to be able to control a single callback, it might be necessary. For example, you might want to take the id returned by `AddCallback()` and pass it off to any number of systems, any of which might want to change the callback time or remove the callback. This would allow you to notify all of the other systems that the callback was removed.

This chapter has really just scratched the surface of what you can do with event-based updates. There is no one-size-fits-all solution to every problem, and you will need to tailor the system presented here to fit your specific game. However, by embracing the core concepts of event-driven simulation, you will be able to significantly cut down the amount of wasted CPU time and spend it on more interesting behaviors.

7 References

[Cormen 2009] Thomas H. Corment, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition*. Massachusetts Institute of Technology.