Two Brains Are Better Than One
How to thread game AI.

By Orion Granatir

Artificial intelligence (AI) drives gameplay—whether you’re talking about a complex system of AI or a simple scripting engine. To really maximize your AI’s potential, it needs to utilize the entire CPU and this means threading. This article examines how to thread a simple AI and some of the challenges in writing an AI that truly scales with multi-core CPUs.

The concepts described in this article were used in the creation of the multithreaded AI of Intel’s Smoke demo. This demo showcases functional and data decomposition with multiple game technologies, including physics, audio, and AI. The source code is free to download at Whatif.intel.com.
Why Thread?

Imagine you want to update a bunch of AI monsters. If you have only one core, all of those monsters must be processed in order. However, if you have multiple cores, you can process them at the same time.

More cores, and thereby more active threads, means you can have more monsters in your game!

A Simple AI

Let's start by defining a simple AI. Our AI is going to be an enemy that idles, waiting to spot the player. When it spots the player it runs at them and explodes when it gets close.

Many games use a state machine to define AI behavior. So, let's define our states . . .

```c
case AI_SPAWN:
    {
        // Initial setup
        Setup();
        m_State = AI_IDLE;
        break;
    }
```

The first state is AI_SPAWN. This is the initial state and sets up the AI. Once the setup is complete, the AI is put into the AI_IDLE state.

```c
case AI_IDLE:
    {
        // Check if we can see the player
        Collision::Result Result = Collision::RayCast( m_Pos, GetPlayerPos() );
        if( Result.m_Hit == GetPlayerID() )
            m_State = AI_ATTACK;
        break;
    }
```

While in AI_IDLE, the enemy does a ray cast to determine if it can “see” the player. If the ray cast reaches (hits) the player, the state changes to AI_ATTACK.

```c
case AI_ATTACK:
    {
        // Find a path to the player
        Collision::Path Path = Collision::FindPath( m_Pos, GetPlayerPos() );
        // Update our position to move towards the player
        Move( Path );
        // Check if we are close to the player
        Vector Distance = GetPlayerPos() - m_Pos;
        if( Distance.Magnitude() < 1.0f )
            m_State = AI_EXPLODE;
        break;
    }
```

While the AI is in the AI_ATTACK state it finds a path to the player. When it gets close it changes to AI_EXPLODE, hopefully causing damage to the player.

```c
case AI_SPAWN:
    {
        // Initial setup
        Setup();
        m_State = AI_IDLE;
        break;
    }
```
There is another way to reach AI_EXPLODE. If the player damages the AI, it should die and change to the AI_EXPLODE state.

Some interesting challenges arise from threading this code. We want it to run on an arbitrary number of threads, distributing the updates among the available cores of the CPU.

In the AI_IDLE state, the code issues a ray cast with the collision system. When run in serial, only one AI at a time interacts with the collision system. When run in parallel, multiple AIs utilize the collision system. The collision system must be “thread safe,” or the code accessing the collision system must be protected with a lock to allow only one-thread access (for example, use a critical section).

In the AI_ATTACK state, the code is trying to path find to the player. This problem is very similar to the collision system; multiple threads are accessing the pathfinding code at the same time. Furthermore, what if two AIs want to move to the exact same location? This is fine with serial code because the first one there wins. The second AI will “see” that the other AI is already standing there. However, when run in parallel the two AIs might decide to move to the same location at the same time, not realizing that they will end up in the same spot.
Before we tackle the challenges of the collision system and pathfinding, let’s actually thread this code. It may be easier than you think!

**Just Thread It**

Let’s assume we have a global array that contains all our AIs. On each frame we want to loop through the AIs and update them. For our simple AI, this means calling the code that we defined above. Here is the code to update the AI:

```c
void UpdateAI();
{
  for( int i = 0; i < g_Num_AI; i++ )
  {
    g_AI_Array[i].Update();
  }
}
```

Simple enough, now let’s thread it. This is a great opportunity to use data decomposition. We want to break up the AIs into groups of data to be processed in parallel. For example, if we have a four-core machine, let’s update one-quarter of the AIs on each core.

Among the many APIs available for threading, two good ones are OpenMP® and Intel® Threading Building Blocks. In this case, we’ll use OpenMP to divide this loop among multiple threads.

```c
void UpdateAI();
{
  #pragma omp parallel for
  for( int Index = 0; Index < g_Num_AI; Index++ )
  {
    g_AI_Array[Index].Update();
  }
}
```

That was easy! The AIs will all call Update in parallel. Now let’s look at the challenges outlined in the previous section.

**Threaded AI That Plays Well with Others**

For our simple AI, we need the collision and the move system to be thread safe. We could put locks around any access to these systems, but we’ll see lower performance because threads will have to block and wait for each other.

The solution for collision and pathfinding are similar, so let’s focus on collision. For the collision system we will have multiple requests in parallel. For multiple readers, the collision data is just fine. This is no different from the serial case because the data is not changing. When someone starts writing data, then we have work to do. Data access to be protected (with locks) or the data has to be double buffered. If the data is double buffered, the readers can just access the data from the last frame and not worry about the writes until the next frame.

We can run collision independent of AI three ways: issue early and use late, deferred requests, and task stealing. Each is worth a bit of investigation.

**Issue Early, Use Late**

For AI_IDLE, the code must do a ray cast, which can be queued for processing. If we just queue the request, we have very little code that needs to be thread safe; only the queue needs a lock, and that’s a short piece of code (it could even be implemented using lockless techniques). The collision system can then run on another thread and service all the requests.

We have to change our AI a little to make this work. Let’s add a PreUpdate method that is called on all AIs prior to calling Update. In this PreUpdate, we issue our request to the collision system.

Here is the new threaded AI loop:

```c
void UpdateAI();
{
  // Process all PreUpdates (this is new)
  #pragma omp parallel for
  for( int Index = 0; Index < g_Num_AI; Index++ )
  {
    g_AI_Array[Index].PreUpdate();
  }

  // Process all Updates
  #pragma omp parallel for
  for( int Index = 0; Index < g_Num_AI; Index++ )
  {
    g_AI_Array[Index].Update();
  }
}
```
This code now processes all the PreUpdates in parallel and then all the Updates in parallel. Now let’s take a look at how we would write the AI_IDLE case in PreUpdate and Update:

```cpp
void Ai::PreUpdate( void )
{
    switch( m_State )
    {
    case AI_IDLE:
        { // Issue collision check
            m_Handle = Collision::RayCast( m_Pos, GetPlayerPos() );
            break;
        }
        // ...
    }

    void Ai::Update( void )
    {
        switch( m_State )
        {
        case AI_IDLE:
            { // Check if we can see the player
                Collision::Result Result = Collision::GetResult( m_Handle );
                if( Result.m_Hit == GetPlayerID() )
                {
                    m_State = AI_ATTACK;
                }
                break;
            }
            // ...
        }
    }
```

When the AI is in the AI_IDLE state, it issues a ray cast request in PreUpdate and gets a handle back. On another thread, the collision system is taking this request and processing it. When we reach Update, the code requests the result of the collision using the handle. Hopefully the collision system has had enough time to process the request. If the collision is not ready, we have to block and wait until it’s ready.

**Deferred Requests**

Waiting kills performance. If the collision system cannot process requests fast enough, the AIs end up waiting. To solve this, we can make our requests more deferred. In the PreUpdate, we issue a request only if we didn’t have one pending (that is, we don’t have a valid m_Handle). And the code in Update has to gracefully continue if Collision::GetResult returns “not ready.” The code probably just does nothing for this frame.

The collision system has a lot more time to process the request. However, a request could span an entire frame and increase latency. Task stealing can make this better.

**Task Stealing**

When the code calls Collision::GetResult, the AI can block or defer processing until the next frame. The last option is to use task stealing. If the collision request isn’t ready, the thread scheduler can have the thread processing AI “steal” the request. The thread processing AI then starts helping the collision system, allowing for better load balancing.

To learn more about task stealing and a thread scheduler that supports this style of execution, read about “promises” in Brad Werth’s “Optimizing Game Architectures with Intel Threading Building Blocks”.

---

**Deferred Requests**

Waiting kills performance. If the collision system cannot process requests fast enough, the AIs end up waiting. To solve this, we can make our requests more deferred. In the PreUpdate, we issue a request only if we didn’t have one pending (that is, we don’t have a valid m_Handle). And the code in Update has to gracefully continue if Collision::GetResult returns “not ready.” The code probably just does nothing for this frame.

The collision system has a lot more time to process the request. However, a request could span an entire frame and increase latency. Task stealing can make this better.

**Task Stealing**

When the code calls Collision::GetResult, the AI can block or defer processing until the next frame. The last option is to use task stealing. If the collision request isn’t ready, the thread scheduler can have the thread processing AI “steal” the request. The thread processing AI then starts helping the collision system, allowing for better load balancing.

To learn more about task stealing and a thread scheduler that supports this style of execution, read about “promises” in Brad Werth’s “Optimizing Game Architectures with Intel Threading Building Blocks”.

---
Conclusion

Threading AI is required to get the best performance out of modern CPUs. By properly structuring collision and pathfinding, adding threading to AI is simple. Making a collision and pathfinding system use requests has multiple advantages: it’s easy to work with threaded systems, it’s easy to thread the system itself, and the requests can be processed on remote compute devices, such as network machines, SPUs, and more.

Check out Smoke on WhatIf.intel.com to see these concepts in action and give multithreaded AI a try.

Multithread AI FTW!

About the Author

Orion Granatir is a senior engineer in the Visual Computing Software Division. He is the technical lead on the Smoke demo project. Prior to joining Intel in 2007, Orion worked on several Sony PlayStation* 3 titles as a senior programmer with Insomniac Games. His most recent published titles are Resistance: Fall of Man* and Ratchet and Clank* Future: Tools of Destruction. He wants to hear from game developers on what topics they would like to see covered in future articles. Please contact him at Orion.R.Granatir@Intel.com