One problem that limits the ambitions of modern real-time strategy games is how to tackle the simulation of more than a few hundred units on screen all interacting with collision avoidance. How can we raise the bar on the number of units in a game and still have the game run in real time? Strategy and arcade genre games have small unit counts due to processing overhead. In Colony, we show how that overhead can be minimized with threading on the CPU. We show how to architect multi-threaded collision avoidance. We use profiling tools to visualize the application's behavior, identify problems, and verify the solutions. When optimized, Colony can simulate many thousands of AI-controlled units at 30 frames per second.

Introduction
Colony demonstrates a simulation on the scale of thousands of units within the context of a simple, game-like scenario. This demo showcases real-time local collision avoidance in a task-based, optimized code sample. Colony is implemented as an arcade game to simulate the performance budget game developers have for AI pathfinding and collision avoidance technologies. With Colony, we show how to manage a large number of units without busting the performance budget.

**Colony: A Game That Plays Itself**

This is not technically a game in that it has only minimal user interaction and no goal, but it was designed using a similar programming approach. The logic flow and data layout are comparable to what you would find in a modern game. Colony starts with a forest scene on a sunny day. Zooming in closer, you see robots moving about. The robots’ primary goal is to pave over the entire forest. They first build some concrete factories around the map, after which they systematically deforest the region by making concrete, paving a section, and returning for more concrete. They go back and forth like this until the entire forest is gone, then they start over.

**Architecture of the Colony Simulation**

Colony is implemented in two parts, simulation and everything else such as gameplay and anything not related to the units. The map tiles, the trees, the coverage calculation, are all part of the second part. It’s fairly lightweight because there is no real game functionality, but this is where any additional features would be implemented.

Simulation data is stored in Game object along with the characteristics of the level, such as the dimensions of the map and associated data. The key function of the Game object is the simulation update routine. Collision avoidance is done in the update step using a spatial structure for sorting units. Colony uses a modified kd-Tree to store units, allowing fast access to the nearest neighbors of each unit. We also considered using a broadphase sweep and prune (SAP) approach. The problem with a SAP method is that insertions and deletions are expensive, and the unit count fluctuates constantly. Units were generally uniformly distributed on the map. To improve insertion and deletion performance we implemented a simple binning approach to exclude far units from consideration.

The terrain was partitioned into a fixed-size rectangular grid that corresponds to the physical position of the map. At the beginning of each frame, we insert the units into bins based on their X and Y positions in the simulation. The bin size was chosen by experimenting with different sizes.

The search distance in the simulation is based on the dimensions of the bins. Imagine just a few units all jostling around inside a square, each trying to avoid each other while attempting to move along a path to their goal.
The initial implementation of the update step looped through all units and computed the velocities each unit needs to reach its goal. To improve performance, the sight distance of each unit is limited to 2 bins allowing each unit to consider significantly fewer neighbors. The size of each bin is bounded by the maximum number of units that can exist on that part of the terrain.

Units carry with them all the properties necessary to move about the map, such as velocity, their goal position, their position, and their radius. These are all stored in structure-of-array (SOA) data structures to ensure maximum cache efficiency and SIMD usability. All the information is stored in these arrays and then units are accessed solely by index.

Stepping the Simulation: The Update Call

Now that we have a better understanding of the underlying storage and binning, it’s easier to see what goes on during a simulation update. The main loop in Colony calculates direction for units and reconciles physical forces and constraints from their local collision avoidance. A closer look at the simulation reveals a specific design strategy and the threading and performance optimizations that make such a large number of units possible.

Binning

We chose a static binning approach for two reasons. The first is better performance for the nearest-neighbor search needed for unit-unit collision avoidance. The second reason is the ease of hashing a position from the UI to world space. During each frame we rehash all units to their proper bins for the collision avoidance calculation. Current bin size provides the best balance between overhead of processing bins and simulation performance.
We had originally considered using a dynamic binning approach, in which the bins would automatically size themselves to contain all the units. However, this method ended up adding too much overhead in our specific scenario. A dynamic binning approach would make more sense with less uniformly distributed units or a dynamic map.

**Updating Game Objects**

This algorithm calculates each unit’s new direction by casting two rays out from the current unit. These rays go in the direction the unit is currently moving, and are spaced the width of the unit apart.

These rays are compared against every neighbor unit in each of the neighbor bins. If either ray hits any of the units, the current unit turns to the same point on the opposite ray (the yellow arrows in Figure 4). This causes the unit to make a sharp or gradual turn, depending on the distance to the neighbor.
Parallelize Serial Hotspots

Amdahl’s law reveals how serial code limits the maximum performance we can expect [M.QUINN04]:

**Amdahl’s Law**

Let $f$ be the fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$. The maximum speedup $\psi$ achievable by a parallel computer with $p$ processors performing the computation is:

$$\psi \leq \frac{1}{f + (1 - f)/p}$$

Ideally the Colony simulation would complete the entire update in parallel. Unfortunately, sequential steps are often necessary, as many computations serve as input to further steps. In Colony, we chose to distribute our work as tasks. For our approach, we utilized a Task Manager wrapper around Intel® Threading Building Blocks (TBB) that provides an easy interface to TBB’s task-stealing architecture. With
this Task Manager, we create a taskset for each step in the simulation, specifying all prerequisites to the taskset as dependencies. Dependencies of a taskset are previously created tasksets that must complete before the new taskset can begin execution. The simulation is partitioned such that each taskset does not require data calculated in other tasks within that operation, and that the range of data being operated on is easily split up. Our update function performs the following parallel steps, operating on arrays where the start and end indices demarcate a group of tasks executed in parallel across the cores of the CPU.

1. Assign units to bins
2. Determine new directions for binned units (depends on number 1)
3. Update units along their new direction and perform unit logic (depends on number 2)

We can see the impact of this parallelization with Intel® VTune Amplifier XE 2011, showing the CalculateDirection method’s concurrency when measured on a 4-core 2nd Generation Intel® Core™ processor with HDG 3000.

![Concurrency](image)

**Figure 5. Concurrency of the CalculateDirectionTask function**

Intel VTune Amplifier XE 2011 also shows us overall concurrency. The majority of the simulation time is well distributed across the available hardware threads with only minor serial sections that can be mostly attributed to rendering:
Figure 6. Overall concurrency histogram

And here is the task distribution across one frame as observed using Intel® Graphics Performance Analyzers (Intel® GPA) Platform View with an exploded view of one hardware thread. As previously shown in Intel VTune Amplifier XE 2011, calculating directions takes up the majority of the time:

Figure 7. Work spread across 8 threads on the test system
Performance

Figure 8. Unit count at 30 frames per second
With Intel® Hyper-Threading Technology on, we see a 1.79x performance gain when moving from 2 cores to 4 cores. This analysis was performed on a 4-core 2\textsuperscript{nd} Generation Intel® Core™ processor with HDG 3000 with each core at 2.30 GHz.

**Future Work**

Potential future work that can be done on Colony includes interactivity, animation, and improved binning. Interactivity and animated units would make the current workload more representative of a real game workload. Improved binning should be able to adapt bin size based on unit density. Smaller bins for high densities and larger bins for low densities would equalize the workload of each bin. In the binning approach currently implemented, some bins may have a max of one hundred units while other bins may be empty. A dynamic binning approach would better distribute the work of calculating directions for high concentration of units across multiple threads.

**Conclusion**

As CPU core counts continue to increase, the ability to harness that available horsepower is crucial to budgeting a larger number of units in modern real-time strategy games. Colony demonstrates one of many possible technologies for simulating a large number of units with collision avoidance. More
advanced technologies could be considered, as well as implementations that focus on more advanced AI. As seen in our simulation update step, the key to easy parallelism lies in the identification of sequential blocks of code with tools such as Intel® Parallel Studio and the storage of data so that it can be easily broken up into chunks for processing, all of which is possible with Intel® Threading Building Blocks.

**Resources**


**About the Authors**

Kyle Weicht is a software engineer in the Intel Software and Services Group, where he supports Intel graphics solutions in the Visual Computing Software Division. He holds a B.S. in Game Development from Full Sail University.

Omar A. Rodriguez is a software engineer in the Intel Software and Services Group, where he supports Intel graphics solutions in the Visual Computing Software Division. He holds a B.S. in Computer Science from Arizona State University. Omar is not the lead guitarist for the Mars Volta.

Jeff Freeman is a software engineer in the Intel Software and Services Group, where he supports Intel graphics solutions in the Visual Computing Software Division. He holds a B.S. in Computer Science from Rensselaer Polytechnic Institute.