Stride Scheduling for Time-Critical Collision Detection

Daniel S. Coming†
Desert Research Institute

Oliver G. Staadt∗
University of Rostock, Germany

Abstract

We present an event-based scheduling method for time-critical collision detection that meets real-time constraints by balancing and prioritizing computation spent on intersection tests without starvation. We test each potentially colliding pair of objects at a different frequency, with unbounded temporal resolution. We show that believability is preserved by adaptively prioritizing intersection tests to reduce errors in collision detection, using information about the objects and scene. Through the combination of kinetic sweep and prune with stride scheduling we continuously interleave rendering, broad phase collision pruning, narrow phase intersection testing, and collision response. Our method accrues no per-frame overhead and is interruptible at any point in collision detection, even the broad phase.

1 Introduction

Collision detection is a necessary and often computationally expensive element of virtual environments, interactive 3D graphics, and games. As scenes grow larger and more complex, collision detection becomes a bottleneck for real-time performance. Similar to time-critical rendering [8], delays caused by collision detection are apparent; they result in inconsistent frame rates and/or a significant slowing down of the application. Psychophysical experiments on perception found that latency between the expected and displayed time of collision response impacts belief that the collision caused the resulting change in objects motions [19]. Reducing gaps between colliding objects yields significant improvements in believability of collision handling [19]. Time-critical collision detection must delicately balance the trade of accuracy for performance in such a way that maintains consistent frame rates without slowing down the application while avoiding large interpenetrations and gaps between colliding objects. Scheduling is key to maintaining believability of collision handling in real-time systems [18].

Broad phase collision pruning can significantly reduce the number of pairs to test for intersection, yet many potential collisions (i.e., up to $O(N^2)$) may still require narrow phase testing. Efficient narrow phase intersection testing methods reduce the cost of testing each for a collision [21]. Time-critical collision detection takes active pairs of objects as output from the broad phase, and schedules narrow phase intersection tests for each. If all are treated with equal importance, real-time constraints cannot be guaranteed without choosing a sufficiently large time step, which could introduce temporal aliasing (i.e., missed collisions where objects move through each other due to discrete sampling). Instead, potential collisions should be prioritized to maintain accuracy where it is most important. The placement of importance varies from one application to another, so prioritization methods should be flexible and intuitive to apply, using factors such as the speed, visibility, or importance of objects or the user’s attention. Object pairs compete for computation time and processing order, so priority-scheduling collision detection may allow low-priority collision tests to starve if all tests are scheduled at the beginning of the frame and unfinished tests are discarded at the end (e.g., a one-frame scope for prioritization). When a collision test starves, it can either be ignored or treated as a collision, leading to increased penetrations or gaps, respectively, between colliding objects; either choice suspends belief. Continued starvation amplifies these affects.

Through combination of kinetic sweep and prune [4] with a resource scheduling method known as stride scheduling [23], we substantially improve the performance and quality of real-time collision detection. We (i) prioritize object intersection tests on a per-pair basis resulting in variable testing frequencies; (ii) significantly reduce errors due to latency as well as penetrations or gaps between colliding objects; (iii) develop automatic prioritization functions based on object, scene, and event information; (iv) offer a flexible and intuitive method for combination of priority functions to meet users’ needs in different scenes; (v) include considerations for users to develop their own priority functions, as well as arbitrary priority ranges at the level of object, pair, or type; (vi) extend kinetic sweep and prune for application to discrete collision detection; and (vii) add interruption capabilities to incremental GJK.

∗e-mail: daniel.coming@dri.edu
†e-mail: ostaadt@informatik.uni-rostock.de
2 Related Work

Fares and Hamam [7] provide an overview of collision detection methods. Approaches to time-critical collision detection make use of interruption and level-of-detail as well as traditional optimizations for collision detection. GPUs accelerate collision detection [9, 13], but may not meet time constraints. Discrete-step broad phases [3, 10, 14] can also violate real-time constraints, because they are not interruptible. Interruption would leave them in invalid states where some objects occupy new positions while others remain in prior positions, resulting in both false positives and false negatives. There would be no time left for narrow phase intersection tests or collision response.

Hubbard [10, 11] used round-robin scheduling and interruptible hierarchical sphere-tree intersection tests to meet time constraints. Collision detection results were refined fairly, but without prioritization. Continuing with that model, O’Sullivan and Dingliana [5, 17–19] explored perception-based prioritization and several priority scheduling methods for collision detection. Further, they developed collision response methods for cases where hierarchical intersection tests are interrupted, specifically for sphere-trees. They enhanced believability through prioritization, however they had a one-frame scope and test frequency. They did not prevent starvation, so low-priority collisions may never be refined. Klein and Zachmann [12] extended interruptible hierarchical intersection tests with branch selection guided by the probability of finding a primitive intersection further down the tree. Further, in parallel close proximity, although the underlying geometry does not intersect, their sphere-trees may. The number of required collision responses can grow exponentially for each level of refinement, when the method is interrupted, due to false-positives introduced by using spheres to approximate the geometry. Sphere-trees have difficulty approximating objects with holes (i.e., genus greater than one) for the first several layers of the hierarchy [11]. Interruption could prevent objects from passing through these holes. We refine completely to the underlying geometry and avoid these problems. Early experiments have been performed with interruptible OBB-trees, which might better approximate objects with holes, but no results have been published [20]. None of these methods discuss interrupting the broad phase. They also do not interleave collision response, so they must predict and reserve time for this to follow interruption of collision detection.

Some methods use variable time steps to calculate the time of impact (TOI) for collisions and advance objects to that precise time before applying collision response [1, 2, 6, 15, 16]. However, these were not designed to meet real-time constraints. They varied time steps to improve accuracy but not to reduce workloads. We do both.

Figure 1: Activity diagram of our method: Scheduling allows collision detection events (e.g., object pair intersection tests, BV pairs begin/cease to overlap) to interleave with simulation tasks. Simultaneous events occur atomically. Collision detection does not surpass the next rendering or other key point \( t_{\text{max}} \).

3 Interruptible Collision Detection

We separate the broad and narrow phases into highly granular, atomic tasks that are triggered by scheduled events. The broad phase and our priority scheduler generate interleaved events as depicted in Figure 1. We prioritize and schedule active pairs of objects for intersection testing. Tests occur simultaneously when they activate and first enter the priority schedule. If a test finds an intersection, we apply collision response to the intersecting objects and re-prioritize any scheduled tests involving either colliding object. Otherwise, the test is rescheduled with a new priority. We allow arbitrary changes to the motion (position, velocity, etc.) at any point in time. Objects advance conservatively, undergoing collision response before they can interpenetrate [18]. Collision response and arbitrary motion changes result in re-prioritization of scheduled tests involving modified objects. When the broad phase deactivates a pair of objects, their intersection test is removed from the schedule. The broad phase activates or deactivates object pairs in order of the exact time their bounding volumes (BV) begin or cease to overlap, so we can interleave broad phase collision pruning, priority calculations, scheduling, and narrow phase tests in time-sequential order. Fine granularity and interleaving of these events helps interruptibility to meet real-time constraints and lets priority scheduling maximize use of excess time to improve accuracy.

Broad Phase: Kinetic Sweep and Prune Our framework could support typical broad phase methods (e.g., [3, 10, 14]) by taking any changes to the active list of object pairs once
per frame and adjusting the schedule as necessary. We use kinetic sweep and prune (KSP) [4], an interruptible event-based broad phase method that provides and removes active object pairs in time-sequential order. Neither the culling quality nor costs of KSP depend on intersection test frequency or length of simulation intervals. KSP is an event-based extension of sweep and prune with excellent performance for continuous collision detection. Like continuous collision detection, we want responses to occur immediately and affect the remainder of the simulation interval, improving accuracy. Interrupting KSP does not preclude the narrow phase or collision response. Instead, by processing events in time-sequential order, simulated time can move forward in increments following each event’s completion. Then, if collision detection is interrupted, objects are in a valid state and all work is complete up to the current point in simulated time, so the objects may be correctly displayed at their positions for that time without large interpenetrations from inconsistent state. When matched with discrete narrow phase intersection tests, we must ensure that updated BV motions do not break current BV overlaps in KSP.

**Narrow Phase: Interruptible Incremental GJK** We designed our method to work with standard narrow phase tests, including interruptible tests (e.g., [5, 10, 11, 17–19]). For this, we extended the Incremental Separating Axis GJK method [22], adding support for interruptions. We chose this method because its incremental approach saves much re-computation, given the high testing frequencies stride scheduling can adopt. This savings leaves a greater proportion of the narrow phase time available for improving accuracy. Unlike with other interruptible narrow phases, ours simply pauses until allowed to continue or to cancel the test. This is performed with purely sequential threading mechanisms. The intersection test runs in its own thread in order to keep its own stack. If it runs out of time, it wakes the main thread and then sleeps until called to continue. This way, we maintain fine grain geometric accuracy in the narrow phase, since our stride scheduling manages the performance-accuracy trade off. We believe this approach could be applied to a continuous test, but our narrow phase performs a discrete intersection test, like other interruptible methods. Similarly, we believe our test could be extended from rigid bodies to apply to deformable objects.

### 4 Stride Scheduling

To take advantage of priority scheduling and also account for starvation, we devise a method similar to stride scheduling [23]. Stride scheduling features priority aging: the longer an element waits in the queue, the higher its priority becomes. Priority aging alleviates starvation in process scheduling, and we apply it to collision-test starvation. We prioritize collision tests continuously, intentionally performing intersection tests between a pair of objects at any desired frequency, based on the priority of that object pair and independent of the frequency of simulation intervals or rendering frames. This alleviates the fixed time step restriction, allowing us to make trade-offs between performance and accuracy per object pair instead of a global trade-off.

We perform stride scheduling for intersection tests as follows. We compute the stride function \( s(A, B) \) as the simulated time interval that an active pair of objects \((A, B)\) waits between successive intersection tests:

\[
s(A, B) = \frac{1}{f(A, B)}.
\]

where \( f(A, B) \) is the frequency at which \(A\) and \(B\) are tested for intersection. Pairs of objects with lower stride will be tested more frequently. Each time we test an active pair for intersection, we recalculate its stride \( s(A, B) \) and reschedule it for its next test at time \( t_{\text{test}}(A, B) \):

\[
t_{\text{test}}'(A, B) = t_{\text{test}}(A, B) + s(A, B).
\]

We can weight the test frequency \( f(A, B) \) relative to a base frequency \( f_{\text{base}} \) to give more or less priority \( p(A, B) \) to the potential collision between \(A\) and \(B\):

\[
f(A, B) = p(A, B)f_{\text{base}}
\]

Our priority function is similar to tickets [23], or tasks’ resource allocations relative to other tasks. A priority of one (i.e., \( p(A, B) = 1 \)) means that tests for intersection between \(A\) and \(B\) occur with frequency \( f_{\text{base}} \). Note that priority is directly proportional to the final test frequency and higher test frequencies for a pair of objects promotes greater collision detection accuracy for that pair. Stride scheduling allows infinite variability in frequency of collision testing that is optimal in this framework, providing different frequencies per pair of objects, by factoring object-specific information into stride. Because \( s(A, B) \) can change (e.g., when either \( f_{\text{base}} \) or \( p(A, B) \) change), we use dynamic ticket modification [23]:

\[
s'(A, B)_{\text{remain}} = s(A, B)_{\text{remain}} \frac{s'(A, B)}{s(A, B)}.
\]

A change in stride only scales the remaining portion \( s(A, B)_{\text{remain}} \) of the stride, so this method gives credit for waiting over the fraction of stride that has already passed. When we modify \( s(A, B) \), we reschedule \( t_{\text{test}}(A, B) \) for \( s'(A, B)_{\text{remain}} \) later than the modification time. If \( f_{\text{base}} \) changes, it affects all existing and future strides. This could
Figure 2: Example of stride scheduling intersection tests, showing the queue’s state (column) at each step, with changes. The first four intersection tests are labeled. Activates or deactivates add or remove active pairs from the queue, respectively. At each step without a deactivate, we test the active pair with the lowest event time for intersection and reschedule it based on its stride (dynamic shown).

happen often if we use it to modulate the overall workload, so we delay modification of existing strides to occur just once per simulation step. While we scale all existing strides at once, Equation 4 provides that we uniformly scale a set of remaining times, which are all in the future and relative to the same modification time. Thus, the relative ordering of the queue remains unchanged. Meanwhile, any stride we recalculate for other reasons, such as following an intersection test, uses the latest $f_{\text{base}}$.

We allow active pairs to join or leave the schedule as they activate or deactivate, respectively, as provided by the broad phase. Figures 2 and 3 depict an example demonstrating successive steps of our method. Figure 2 shows how stride scheduling prioritizes intersection tests in the queue. It illustrates a stride’s effect on testing frequency as well as variable-length strides and object pairs joining or leaving the schedule due to broad phase events in Figure 3.

4.1 Prioritized Stride Functions

The choice of stride function significantly affects the performance and accuracy of stride-scheduled collision detection. We aim to both minimize latencies and gaps between colliding objects, within time constraints. Information used in our stride and priority functions must be efficient to gather. Stride functions compose a frequency base $f_{\text{base}}$ with optional object-specific priority functions $p(A, B)$. Substituting Equation 3 into Equation 1, we get:

$$s_{\text{base}}(A, B) = \frac{1}{p(A, B)f_{\text{base}}}.$$  (5)

We present results from scheduling with the following stride functions in Section 5. The priority functions which contribute to stride are designed for combination with each other and future extension.

**Constant Stride** A constant stride represents a fixed time step $\Delta t$ so that intersections between possibly colliding objects are all tested at the same constant frequency $f_{\text{const}}$. To obtain this, we give all $(A, B)$ the same priority $p(A, B)=1$ and replace $f_{\text{base}}$ with $f_{\text{const}}$:

$$s_{\text{const}}(f_{\text{const}}) = \frac{1}{f_{\text{const}}} = \Delta t.$$  (6)

With constant stride, real-time performance depends largely on the choice of $\Delta t$; even though the method is interruptible, there is no load balancing, so collision detection may accumulate too much simulation latency to offer reasonable quality. Besides affecting causality, latency can alter the apparent movement rates of displayed objects. When latency is low, objects move smoothly and behave as expected, but with increasing latency, objects appear to slow...
down and move in noticeably discrete steps. Then, as latency subsides, objects appear to accelerate unnaturally and move faster than they should.

**Dynamic Stride** Real-time constraints require a stride function that can adapt to changing work load without the dependence on choosing a good $\Delta t$.

$$s_{\text{dynamic}}(A, B) = \frac{1}{p(A, B)f_{\text{dynamic}}},$$  \hspace{1cm} (7)

where $f_{\text{dynamic}}$ replaces $f_{\text{base}}$ as an adaptive and infinitely variable testing frequency. The scheduler may decrease $f_{\text{dynamic}}$ to meet time constraints when necessary, and increase $f_{\text{dynamic}}$, and thus improve accuracy, when there is extra time for collision detection. We want to fill the CPU time $t_{\text{avail}}$, which is known to be available for intersection tests, with as many tests as possible, while advancing the current simulated time $t_{\text{simulated}}$ to the goal of matching the real time on a wall-clock $t_{\text{real}}$. We want:

$$(t_{\text{real}} - t_{\text{simulated}})f_{\text{dynamic}} \sum_{i \in \text{queue}} p_{ci} c_i \leq t_{\text{avail}},$$  \hspace{1cm} (8)

Simulated time is not required to be synchronized to real time, but this calculation includes simulation latency between the simulated time and real time in an attempt to resolve it within about one frame, so that it will be unnoticeable. It also factors an unscaled estimate of the future cost, the priority (therefore relative-frequency) weighted sum of last known costs $c_i$ (in seconds) for each test $t_i$. This estimate will be scaled by $f_{\text{dynamic}}$. We rely on coherence for the actual test costs to be close to this recent $c_i$, given similar positions and bounding volume tree traversals. We do not keep $c_i$ for inactive pairs of objects; recall that each time object pairs activate, we perform an immediate intersection test. This provides a starting $c_i$ as a side effect.

**Velocity Priority** Equation 7 reduces gaps between colliding objects by maximizing intersection testing frequency. However, the size of these gaps is not a factor of only time but also velocity. Hence, we prioritize with the relative velocity between objects, testing for intersections more frequently between rapidly approaching objects.

$$p_{\text{vel}}(A, B) = \frac{|v_A - v_B|}{v_{\text{max}}},$$  \hspace{1cm} (9)

$$s_{\text{dyn}, \text{vel}}(A, B) = \frac{1}{p_{\text{vel}}(A, B)f_{\text{dynamic}}},$$  \hspace{1cm} (10)

where $v_A, v_B$ are the velocities of objects $A, B$, and $v_{\text{max}}$ is the magnitude of the maximum relative velocity measured.

Stride $s_{\text{dyn}, \text{vel}}$ incorporates velocity prioritization $p_{\text{vel}}$ with dynamic load balancing to reduce temporal aliasing, even when work load is high.

Prioritization with acceleration and higher order motion is possible, however, it is likely to have diminishing returns unless there are significant differences between the accelerations of objects. The improvement would be more noticeable if increased workload causes significantly reduced testing frequency, e.g., below 10 Hz.

Naturally, a lower bound on the distance between objects, along with upper bounds on velocities, gives a TOI estimate, and intersection test between them would not be necessary until that TOI, as described by Cameron [1,2] or Mirtich and Canny [16]. Such conservative advancement could easily be scheduled with our algorithm. However, such distance calculations are expensive, take many iterations to converge to the point of collision, and may not meet time constraints.

**Mass Priority** We can also consider the effects that properties such as mass have on collision response. When objects with different masses collide, the ratio of their masses has a significant effect on their resulting motions, so it is important to detect these collisions accurately.

$$p_{\text{mass}}(A, B) = \frac{\max(m_A, m_B)}{\min(m_A, m_B)},$$  \hspace{1cm} (11)

where $m_A$ and $m_B$ are the masses of objects $A$ and $B$.

**Velocity Mass Stride** Combining multiple priority factors that complement each other further improves accuracy for collision detection and therefore response. For example, the following stride function $s_{\text{vel}, \text{mass}}$, which works well in our heavy projectile test scenario, considers the relative velocity of objects in addition to the ratio of their masses.

$$s_{\text{vel}, \text{mass}}(A, B) = \frac{1}{p_{\text{vel}}(A, B)p_{\text{mass}}(A, B)f_{\text{base}}},$$  \hspace{1cm} (12)

$$f_{\text{base}} \in \{f_{\text{const}}, f_{\text{dynamic}}\}.$$

### 4.2 Total Stride

Combining appropriate priority factors that describe the needs of a particular application should be flexible and intuitive. The following is the general form for calculating the
total stride as a combination of priority factors:

\[
s_{\text{total}}(A, B) = \frac{1}{p_{\text{total}}(A, B) f_{\text{base}}},
\]

\[
p_{\text{total}}(A, B) = \prod_{F \in P} \left( \sum_{\text{factor} \in F} p_{\text{factor}} \right), \tag{13}
\]

where \( F \) is a category of factors and \( P \) is the set of all factor categories. This form allows the proportional combination of factors within categories in \( P \), relative to factors across categories. For example, it is intuitive to combine linear and angular velocities relative to the size of an object. Both motion factors affect the speed of individual points on an object, but size affects the possible penetration depth and likelihood of temporal aliasing. Examples of geometric factors include relative size, complexity, or topology of an object. Interaction includes both user interaction and inter-object interactions (collisions), and is straightforward to automate with a decaying function of time since the last time an object’s motion changed. Region factors account for logical scene partitions (e.g., walls) as well as proximity to more likely collision areas (e.g., ground versus sky). This is not intended as an exhaustive list of priority categories or even factors within categories, but as a basis to be built upon. Application areas likely have specific additional factors to consider, and arbitrary priority factors are also possible.

**Priority Function Considerations** Priority functions are intuitive to apply, and do not require special numerical tuning except for the proportions of scale of priority factors within the same category. We scale these relative to the maximum or minimum values seen at run time. It is also feasible to limit the range of priority values globally or selectively if specific prioritization is desired. The range of numbers chosen are not important if the dynamic stride base is chosen, as its adaptation will quickly conform to the scale of these numbers. Relative proportion of priorities within a category matters most. The dynamic stride base will also adaptively factor out range differences between different categories. Thus, we combine them if the scale between two categories should be preserved. Additionally, consider the case where either an individual priority or the accumulated priority for an entire category is equal to zero. When the combined priority due to all motion factors is zero, (rigid) objects hold still relative to each other, and their intersection status will not change. Since we always perform an intersection test before calculating the stride to the next test, we know this status. Stride would be infinite, and no test would be scheduled. The next time to perform an intersection test is when either of these objects changes its motion.

![Figure 4: Our test environment: objects' color visualizes their strides in our scheduling algorithm, using \( s_{\text{velocity}} \); green is lower stride, blue is higher. Notice that the large sphere slows down considerably and can use longer stride.](image)

### 5 Results

Our real-time simulation features a heavy projectile colliding into a dense cluster of smaller objects (see Figure 4). The sphere is 960 triangles with an overall diameter of 20 m and initial velocity of 4 m/s towards the cluster. The cluster consists of 400 moving tori with 240 triangles each, with varying diameters averaging 150 cm, and initialized with random positions, orientations, scales, and velocities. Scale varies enough to allow small tori to pass freely through the centers of larger tori. The average velocity is on the order of twice the average diameter (of objects), per second. The sphere has 40 times the mass of the tori.

Rendering time steps advance at a rate of 30 Hz, with a maximum of 33.33 ms real time allowed per frame. When rendering completes, collision detection and response receive the remaining time. Every 33.33 ms, the simulation stops requesting events from collision detection, so collision detection and response stop and the narrow phase pauses any work in-progress. For collision response, we use a simple impulse-based rigid-body response. Because our narrow phase preserves detail down to the actual geometry, we do not need to perform special corrections to the collision response. Results were gathered over 300 frames of simulation (10 s), using one CPU of a dual 2 GHz AMD Opteron 246 and less than 100 MB of the available 8 GB RAM.

**Meeting Time Constraints** The granularity of interruption affects the ability of real-time collision detection to maintain a target frame rate. The charts in Figure 5 demonstrate that stride scheduling with interruptible GJK meets real-time constraints whether using constant or dynamic strides. These tests demonstrate a granularity of less than 0.5 ms (e.g., to meet a hard deadline) with our scheduling method, regardless of stride function. Although all interrupted stride methods meet the deadline, constant stride functions often accumulate latency or else fail to take advantage of available unused time. Instead, our dynamic stride functions adaptively make use of available time with more frequent intersection testing, improving accuracy. Further, our priority functions involve a number of
Figure 5: (a) 0.03333 s target frame completion time. Combining the original KSP with 30 Hz discrete non-interruptible GJK tests cannot always meet the time constraint (frames 59–109) and often leaves unused resources (frames 1–58, 110–300). (b) Stride scheduling and interrupting GJK meet time constraints while introducing little additional cost but accumulate high latency (frames 57–144) and still leave unused resources. This latency peaks at around 220 ms and only subsides when there would be otherwise unused resources. (c) The dynamic stride function makes good use of resources and comes close to time constraints, but it violates time constraints (frame 27). (d) Dynamic stride with interruptible GJK meets time constraints, uses available resources to improve accuracy, and minimizes latency.

Measured Error and Latency We use two error metrics to evaluate simulation quality. Our narrow phase calculates penetration depth \( d_{\text{depth}} \) as the shortest distance an object must move in order to remove an intersection with another object. Greater penetration depths are more challenging to resolve correctly. Next, we quantify visual impacts of latency as the distance \( d_{\text{disp}} \) between an object’s current location and its correct location once latency resolves. This may represent a delay in the object’s advancement or collision response. We gathered results for both metrics. Results in Table 1 demonstrate that time step selection significantly impacts latency. As testing frequency increases, collision detection takes longer to resolve the latency, if ever. For example, in this scene, testing active pairs at a frequency of once per rendering frame (30 Hz) results in a 6.6-frame latency that takes over 50 frames to resolve. Even decreasing the testing frequency to 20 Hz still results in a 2.6-frame latency that takes 79 frames to resolve. At higher frequencies we have observed latency that never resolved, depending on scene complexity. By contrast, scheduling with any of our dynamic stride functions (e.g., Figure 5(d)) allows our method to resolve latencies, usually within one or two frames, without allowing accumulated latency to exceed even one third of a frame. This level of latency (3–6 ms) is imperceptible. Table 1 shows that latency improves significantly, for any stride function that uses the dynamic stride base, compared to constant strides.

The accumulated global root mean square (RMS) \( d_{\text{depth}} \) for all collisions is shown in Table 1. With or without interruption, dynamic stride provides lower \( d_{\text{depth}} \) than even testing all possibly colliding objects at every time step, and does so without the high latencies that are characteristic of constant frequency testing. All of our dynamic stride functions yield improvements of almost two orders of magnitude in latency and two to three orders of magnitude in the latency induced displacement \( d_{\text{disp}} \). While not incurring noticeable latency, dynamic stride functions found time for far more intersection tests and detected more collisions than constant strides. Factoring velocity and/or mass into the dynamic stride function reduces \( d_{\text{disp}} \). Visually and numerically, the dynamic mass stride function excelled by giving priority to the most critical object and the reactions it caused. Priority functions had little to no effect on costs or latency.

Acceptable test frequencies which maintain accuracy and prevent latency accumulation vary with scene, objects, and time. Our method quickly and automatically adapts test frequencies to the scene and objects as well as changes that occur in dynamic scenes. Our method provides lower RMS error on penetrations and displacements and better latency performance than even a single globally optimal test frequency, because we apply greater test frequencies to objects pairs that are likely to have more error, without the cost of applying those test frequencies to all object pairs.

<table>
<thead>
<tr>
<th>( s(A,B) )</th>
<th>Tests</th>
<th>Found</th>
<th>RMS ( d_{\text{depth}} )</th>
<th>RMS ( d_{\text{disp}} )</th>
<th>( L^2 )</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_{\text{const}}(40 \text{ Hz}) )</td>
<td>18324</td>
<td>335</td>
<td>0.00921</td>
<td>0.39496</td>
<td>6.78114</td>
<td></td>
</tr>
<tr>
<td>( s_{\text{const}}(30 \text{ Hz}) )</td>
<td>13612</td>
<td>319</td>
<td>0.01067</td>
<td>0.13306</td>
<td>1.57030</td>
<td></td>
</tr>
<tr>
<td>( s_{\text{const}}(20 \text{ Hz})^* )</td>
<td>9048</td>
<td>331</td>
<td>0.01171</td>
<td>0.03935</td>
<td>0.76023</td>
<td></td>
</tr>
<tr>
<td>( s_{\text{dyn, no interrupt}} )</td>
<td>58211</td>
<td>526</td>
<td>0.00824</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>( s_{\text{dyn}} )</td>
<td>58092</td>
<td>557</td>
<td>0.00831</td>
<td>0.00089</td>
<td>0.02311</td>
<td></td>
</tr>
<tr>
<td>( s_{\text{dyn, vel}} )</td>
<td>56231</td>
<td>608</td>
<td>0.00812</td>
<td>0.00084</td>
<td>0.02128</td>
<td></td>
</tr>
<tr>
<td>( s_{\text{dyn, vel, mass}} )</td>
<td>54376</td>
<td>567</td>
<td>0.00926</td>
<td>0.00072</td>
<td>0.02180</td>
<td></td>
</tr>
<tr>
<td>( s_{\text{dyn, mass}} )</td>
<td>54739</td>
<td>499</td>
<td>0.00855</td>
<td>0.00015</td>
<td>0.02322</td>
<td></td>
</tr>
</tbody>
</table>
6 Conclusion

We presented our interleaved, interruptible, collision detection method, in which every component scales independently of temporal resolution. Further, we vary temporal resolution per object pair, with our stride scheduler, so that dynamic stride can maximize intersection test frequency within real-time constraints. This results in improved accuracy as well as reduced latency, compared to testing with constant temporal resolution, like most methods use. Asynchronous prioritization of object intersection tests further improves accuracy of collision detection in the real-time setting and establishes the foundation for our method to adapt to many applications. We provided automatic prioritization functions to account for object and scene information, an intuitive, flexible method for combining priority factors, and detailed suggestions to aid future prioritization development. As future work, we intend to perform further experiments with priority functions accounting for angular velocity or for cases with unknown underlying dynamics.

Acknowledgments: This work was supported by HP-CITRIS and GAANN Fellowships and U.S. Army’s PEO-STRI NAVAIR under Contract No. N61339-04-C-0072. Thanks to Kimberly Sprague and the members of the Institute for Data Analysis and Visualization (IDAV), University of California, Davis, for helpful discussions.

References