Simulation Level-Of-Detail

Stephen Chenney
University of Wisconsin at Madison

INTRODUCTION

As graphics rendering becomes cheaper and cheaper, physics and AI are coming to dominate the cost of generating each frame of a real-time game. Furthermore, technology to control the cost of rendering exists in the form of hardware, visibility culling and level-of-detail, whereas few cost control methods have been described for physics or AI. Simulation level-of-detail is the term given to a range of techniques for reducing the cost of AI and physics without significantly reducing the quality of the behaviors they generate. In this paper I describe a specific technique, proxy simulations, that can reduce the cost of AI and physics without sacrificing the game-play.

AI and physics are the same in the sense that each is intended to generate changing (dynamic) state in an environment. Typically, AI is seen as the high-level controller – *what* to do - while physics takes care of the details – *how* to do it. One useful way to look at AI and physics is as simulators that implement a particular model of the world, producing motion and state that conform to that model. In this view, the Tetris model is that blocks fall from above in a random orientation and build piles on the floor. The Tetris simulator is code that makes sure that happens. Similarly, a car racing game defines an idealized model of car and driver behavior, while the simulator implements that model to produce the actual motion.

The key idea behind simulation level-of-detail is that *it doesn't matter how the model is implemented, provided a player experiences the right thing.* In particular, we are free to substitute a cheaper implementation if it produces the same experiences as the full simulation. In most games, players experience only things they can see and hear. These things are typically restricted to a relatively small region of space. For example, players only see their immediate surroundings in a first person shooter game. In battle strategy games, the fog-of-war generally restricts views. This suggests a good strategy for cheap implementations: avoid doing work for things the player cannot see.

Game developers frequently choose to ignore the motion of some out of view objects. For example, Joe Adzima recently described the AI for Midtown Madness[1], in which cars are simulated only within a sphere around the viewer and are reflected at the boundaries. No work is done for parts of the city beyond the sphere. These existing approaches are acceptable if nothing should ever happen far from the view to influence the game-play. Hence, while they work for games with local action, they are not applicable for games where events beyond the view may have a major impact on the outcome of the game, such as global strategy games. For example, it makes a difference if your out-of-view opponent can accumulate enough resources to launch an attack on your base. Some work must be done to ensure that the attack happens at all – if nothing is done for out-of-view motion then nothing new can ever enter the view by itself.

The best way to think about motion and outcomes is in terms of *events*: the model says that certain things should happen in the world that are important to the game-play, and any implementation must ensure that those things do happen. On the other hand, there is no need to be concerned with other events, which is where savings can be found. For example, as your opponent is gathering resources, it doesn't matter which paths their workers take, only that the workers gather a certain amount by a certain time, triggering the important attack event.

Discrete event simulation is a programming technique for managing events and their effects that can be used to produce cheap simulations. The program maintains an ordered list of future events that must be processed. It steps through this list processing one event at a time. Each event processing step might modify some program state, and might lead to the addition of new events or the removal of existing events. For example, in a strategy game there could be events scheduled for the start of each attack. Games are frequently scripted with events and rules to process them, but in general some computation is performed on a continual basis and the events are detected as they occur. Discrete event simulation *predicts* when events will occur, and avoids any computation between events. This saves work.

In this paper I will show how expensive AI and physics can be replaced with discrete event simulations that generate indistinguishable event sequences at much lower cost. I will begin with a discrete event model for determining which moving objects are visible. I will then describe a traffic simulation in which expensive dynamics are replaced with a cheaper implementation. Significant speedups result. Along the way I will demonstrate techniques for verifying that the outcomes of the cheap simulations are indistinguishable from those of the expensive ones, an important concern for consistent, predictable game-play. All of the work described in this paper was conducted with Okan Arikan and David Forsyth, both from the University of California at Berkeley.

VISIBILITY FOR MOVING OBJECTS

There are several methods that can be employed to determine if a moving object is visible. Simplest among them is placing a bounding volume around the object and testing it against the view volume on every frame. This method is the least efficient, because every object must be touched on every frame. A better variant, that culls more moving objects, works by associating each object with a cell, or multiple cells, in a decomposition of the environment, and only renders objects in visible cells. The decomposition may be tiles, an octree, a BSP tree, or anything else that a visibility system could use. With most implementations of this method, every object is checked on each frame to see if it has left or entered a cell. So this method also requires looking at every object on every frame, but more objects will be detected as invisible.

A very efficient method incorporates a discrete event view of visibility into the cell-association method just described. Assume for the moment that every moving object is a point. We begin by associating every object with the cell that it intersects. Further assume that we can predict exactly when each object will enter a new cell, and put an event in the queue for that time. We know that, until such an event occurs, the associations are correct between moving objects and cells. More importantly, the data structures do not need to be modified until an event occurs, avoiding costly operations.

When an event comes to the head of the queue, we know that an object is leaving its current cell, so we determine which cell it is entering, add the object to that cell and make a new prediction for when the object will leave the cell. We insert a new event into the queue for the new leaving time, and remove the object from its old cell. If we keep doing this, we can be sure that at all times the data structures associating objects to cells are correct, and so we can be sure that we will render the right set of moving objects.

The real case is slightly more complex. Objects generally have extents, so we actually need two types of event: one for an object starting to enter a new cell and one for an object completely leaving an old cell. Also, we possibly cannot exactly predict when these events will occur. If, instead, we always predict that they occur too early, we can test the situation when the event comes to the head of the queue for processing, and re-insert it at a later time if necessary. Alternatively, if we ensure that entry events come up too soon, and exit events come up too late, then all the visible objects will still be found, but some objects that shouldn't be visible will also be sent to the graphics pipeline. While less efficient, it isn't a problem. With these modifications, the discrete event visibility algorithm will always maintain the correct object-cell associations.

Below I give psuedocode for the algorithm, assuming we do re-prediction if the event comes up too early. `PredictNextEntry(O)` returns the time of the time at which the object is predicted to enter its next new cell. `Entered?(O)` returns the cell that object `O` is entering, or `NULL` if it isn't yet entering any cell. `PredictExit(O,C)` returns the time at which an object is expected to leave the cell `C`, while `Exited?(O)` returns the cell that `O` is exiting, or `NULL` if it isn't on the verge of leaving anything. `InsertQueue(Q,E)` inserts the event `E` into a priority queue `Q`. `DeleteMin(Q)` deletes and returns the minimum time entry in the queue. `AddToCell(O,C)` associates object `O` with cell `C`, while `RemoveFromCell(O,C)` removes the association between `O` and `C`.

```
function Initialize() {
     for all moving objects O {
          E.object = O

          E.time = PredictNextEntry(O)
          E.type = enter
          InsertQueue(Q,E)

          E.time = PredictExit(O,O.currentCell)
          E.type = exit
          InsertQueue(Q,E)
     }
}

function ProcessEnter(E) {
     C = Entered?(E.object)
     if ( C == NULL ) {
          E.time = PredictNextEntry(E.object)
          E.type = enter
          InsertQueue(Q,E)
     }
     else {
```

```
            AddToCell(E.object, C)

            E.time = PredictNextEntry(O)
            E.type = enter
            InsertQueue(Q,E)

            E.time = PredictExit(O,O.currentCell)
            E.type = exit
            InsertQueue(Q,E)
      }
}

function ProcessExit(E) {
      C = Exited?(E.object)
      if ( C == NULL ) {
            E.time = PredictNextExit(E.object,E.object.currentCell)
            E.type = exit
            InsertQueue(Q,E)
      }
      else {
            RemoveFromCell(E.object, C)
      }
}

function MainLoop() {
      Initialize()
      while true {
            E = DeleteMin(Q);
            if ( E.type == enter )
                  ProcessEnter(E);
            else
                  ProcessExit(E)
      }
}
```

A big assumption was made in designing the discrete event algorithm: we could predict when every object would enter or leave a cell. Clearly the assumption is reasonable if the motion is of a very simple form, such as ballistic motion. It is also a reasonable assumption if the complete path is known ahead of time, as may be the case with motion capture playback. In the following sections I will argue that adequate predictions are also available for more complex motion, and in the process work can be saved.

A similar algorithm by Sudarsky and Gotsman uses explicit temporal bounding volumes for objects[2], which does not require a decomposition of the environment. The basic ideas are the same, however, and they also assume that predictions of future motion can be made cheaply.

SAVING WORK

In the visibility scheme above, the simulation performs two tasks: it makes predictions about when objects will enter or leave cells, and it generates state for objects that turn out to be visible. This means there is a lot of work we are potentially free to avoid doing: prediction means we don’t have to generate moment to moment motion for visibility computations, and

the fact that the viewer can't see out of view objects means we don't need to produce state for rendering. However, making predictions and getting the things in view correct may require simulating all of the objects all of the time. For example, in a traffic simulation, the motion of one car influences the motion of the following car, which influences the car behind that, and so on through all the cars in the simulation – including those in view. If we are to capture all of these interactions perfectly, we may need to run the full simulation. The motion of the car clearly also affects the time at which it moves from one visibility cell to another, so to get that right may also require the full simulation.

But players almost never care about all the details of the out of view motion. In particular, a player generally only cares about how out of view motion affects their experience, which generally requires whatever's out of view to eventually come into view. While getting the right thing to happen in view might require getting all the right things to happen out of view, it is generally not the case. For instance, in a driving game the player cares that the right number of cars, roughly, are in view, and that they see the right set of cars behaving reasonably when they turn a blind corner. But the player doesn't care about how far apart two cars are on the other side of the city.

We can establish the following broad requirement: we need to make sure things appear in roughly the right place at roughly the right time, but we need not generate moment to moment motion for out of view objects. To do this we will employ two different simulators: an *in-view simulator* for visible motion and a *proxy simulator* for out of view motion. The in-view simulator takes care of visible objects, and we can just use the regular simulator limited to the set of things in view. This will ensure that the viewer always sees the right thing on a moment to moment basis. The proxy simulator must make sure that out of view objects enter the view at the right place and the right time. More specifically, this means that the proxy must be able to make reasonable predictions for when out of view objects will enter and leave visibility cells, because if that information is correct, then the object will be seen when it should be.

A proxy simulation must save work, and the best way to get big savings is to avoid even considering every object on every frame. This suggests a discrete event model for the proxy simulation, because it allows us to ignore objects between events. To decide when events will occur, we do not need to run the full simulation – instead we can use approximations of various forms. As long as those approximations are reasonable, they will not impact badly on what the viewer sees.

With a discrete event simulation, objects don't really have any state between events. This poses a problem if the player sees previously out of view objects, because those objects will have no good state. It is therefore necessary for the proxy to provide a way to quickly fill in what happens to objects between events, so that they can be given state if they become visible. It is generally safe to use approximations, although the proxy might have to maintain additional state to make sure that the reconstruction is reasonable.

The general guide to building a proxy simulation is as follows:

1. Decide which out of view behaviors you wish to maintain.

2. Determine how to predict visibility cell entry/exit events such that those behaviors will be maintained.
3. Determine what state is needed to reconstruct the motion of objects that become visible, and how to keep track of that state.
4. Test the quality of the results.

Next I describe in detail a proxy simulation for a traffic model, which demonstrates each of the above steps. Some existing approaches could be slotted into the framework of proxy simulations. Our work is distinguished by the types of things we accurately model with our cheap system. For example, Midtown Madness avoids work for out of view cars, but their system would fail if a viewer could identify and cared about particular cars leaving the view. Say your virtual partner was on the way to hospital in an ambulance – you wouldn't want them reflected at the boundary to head back to you. Nor could you ignore the traffic along the route, because the ambulance would arrive at the wrong time. Our system handles identifiable cars moving over large areas, getting things like travel times correct at significantly lower cost.

CHEAP TRAFFIC SIMULATION

The traffic model simulates tricycle cars moving through a maze-like network of roads (Figure 1 and Figure 2). Each road is a visibility cell, open only at the ends. A cell and portal algorithm[3] is used to identify at run-time roads that are visible from the current viewpoint. I describe behaviors from the accurate simulation that we might wish to maintain, and a proxy that reproduces those behaviors at a significantly reduced cost, as indicated by various experiments.

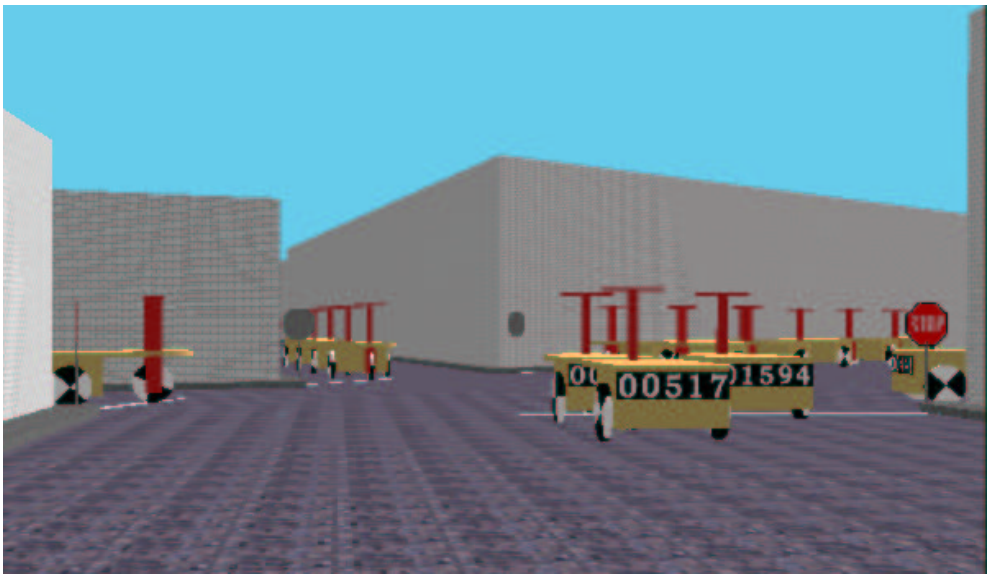

Figure 1: A view of the city traffic simulation. The cars move through a maze-like network of streets, stopping at intersections as shown here. Note that each car is uniquely numbered, allowing a viewer to track its progress if desired.

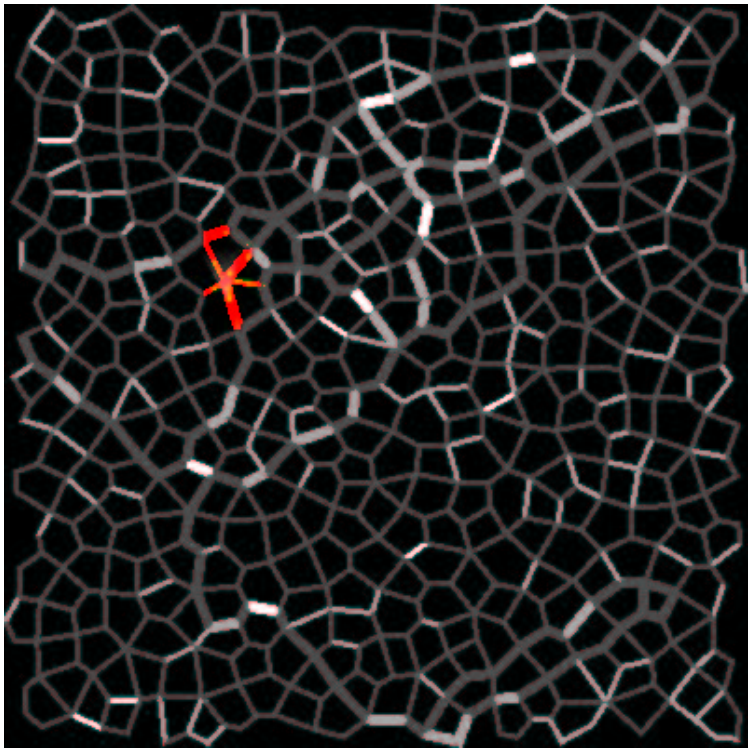

Figure 2: A map view of the city while a proxy is in use. The red cells are the roads potentially visible from the view shown in Figure 1. The other cells have an intensity that corresponds to the number of events processed by the proxy for that road over the course of the previous second of simulation time. Most of the roads are gray, indicating that no events were processed and hence that no work was done for those roads.

The accurate model takes time-steps of fixed length 0.01s. On each step it sets the acceleration for each car to be used throughout the step, and then numerically integrates the motion of each car. Numerical integration is necessary because the orientation and wheel rotations for each car are specified via a differential equation with no closed form solution. The time-step must be kept short because it is the only opportunity for control. Longer time-steps cause cars to overshoot intersections and potentially pass through each other, so manipulating the step length does not produce efficiency gains without introducing major visible errors. In any case, the longest useful time-step for 30fps rendering is 0.033s.

Cars moving under the accurate model exhibit the following behaviors that we seek to maintain:

- **Random path:** Each car takes a random path through the city, with the exception that a car will not turn onto a road that is packed with other cars, unless there is no alternative. Out of view cars managed by the proxy should also make random choices.
- **Stop signs:** The cars obey stop-sign rules at intersections: the come to a stop, wait their turn to move into the intersection, and then pass through, allowing the next car in.

- **Acceleration profile:** Between intersections, each car follows a pre-defined acceleration profile, except that cars slow to avoid collisions with the car in front. If the car in front stops, so will the follower, causing queues to form at intersections as cars line up waiting to enter. When a car enters the view, its velocity and other state should be consistent with the acceleration profile and the cars around it. For instance, a car in an intersection queue should not be moving, and a car that has just moved into an intersection should be moving slowly, because it hasn't had time to gain speed.
- **Traffic density:** Traffic density in the city follows a particular pattern. Some intersections with many incoming lanes typically have long queues, because only one car at a time can pass through the intersection, regardless of how many can reach it. Other intersections rarely have queues, because they may have few incoming lanes or the roads in may be blocked by busy intersections that restrict flow.

Together, all of these properties lead to one meta-property that is visible to a viewer and sensitive to all of the others: car travel times. A viewer can measure travel times by noting the location of a car at two distinct points in time. If cars don't stop, or jump the queues at intersections, they will get places too fast. If traffic density is too high or low, the car will spend the wrong amount of time stuck in queues. Cars following the wrong acceleration profile will move too fast or slow.

We choose travel times as the primary thing to get right with the proxy. Individually, the other properties should be satisfied within the visible regions, but in hidden regions the viewer cannot directly determine the density, or detect whether cars are stopping at intersections, so the proxy need not be concerned with such things unless they influence travel times.

There is, however, one potential difficulty. If a viewer is expecting a car to appear, and it doesn't, they can reasonably infer that there must be a busy intersection between them and the car. If they go looking for that intersection, they should be able to find it. In other words, a viewer can infer things about the out of view world based on what they see in view, and any things a viewer might later see, such as the density of traffic at an intersection, must be consistent with those inferences.

A Discrete Event Traffic Proxy

Recall the task of a proxy: to produce predictions for when an object enters and leaves visibility cells, and to maintain state so that cars that become visible can be correctly updated. The visibility cells in the city correspond to roads. So the proxy must predict when a car will enter and leave a road, which is intimately related to when cars enter and leave intersections. Roads keep pointers to the cars that are on them, and cars keep pointers to the roads they intersect. Cars will become visible either when they drive from an invisible road onto a visible one, or when the viewer moves and sees a new road.

The a high level, the proxy simulates the motion of out of view cars by jumping them from intersection to intersection without determining precisely what happens along the way. The time it takes to move from one location to the next is estimated based on the expected acceleration profile of the cars, and the known locations of other nearby cars. Work is saved by avoiding acceleration setting routines and numerical integration, but the time it takes to get

from one place to another is captured well by the discrete model, along with all the other important aspects of the model. The following sections describe in detail how this is done.

Two events are used to predict when cars will enter and leave roads, and each car may have zero or one events in the queue. The proxy only does work when one of these events comes up, and only considers at most three cars on each event. This results in significant cost savings.* The events are:

- **`Enter`** events occur when a car *might* reach the end of a road and enter an intersection. We schedule these events optimistically, and test whether the car is really entering the intersection only when the event is processed.
- **`Exit`** events occur when a car moves out of an intersection and onto a road proper.

I describe the processing of these events below, as well as how they are scheduled. Two additional data structures are required:

- An **intersection waiting list**: each intersection maintains a list of cars either in the intersection or stopped at stop signs waiting to enter (but not cars in queues behind cars stopped at the line). The list is ordered on the time the car got to its stop sign. If the list contains any cars, the first one is always in the process of passing through the intersection.
- A **lane list**: each lane on a road (at least one in each direction) maintains an ordered list of cars on the lane, with the car that is farthest along the lane at the head of the list. Cars in the intersection that are heading into the lane are at the end of the list. In addition to helping schedule events, the lane lists record the location guarantees --- a car is only on a lane list if it is on the road corresponding to the lane.

Cars store pointers back into any intersection and lane lists that they are currently on.

There are three possible situations when an `Enter` event is processed:

1. If the car is not at the head of its lane, we do nothing. The car must be in a queue behind another car, and it can't leave the road (and enter a new road) until that car moves. The car will be considered again when the car in front of it actually enters the intersection.
2. If the car is at the head of its lane, but there is already a car in the waiting list, then we do nothing. The car cannot move until the car ahead of it on the waiting list gets through the intersection, at which point this car will be considered again.
3. If the car is free to enter the intersection, add it to the randomly selected road that it is moving onto, and schedule an `Exit` event for the car. The time for this event is determined by computing how long it will take the car to get through the intersection if it follows the pre-defined acceleration profile, *assuming that there is no car close in front to slow it down*

---

* The discrete event models maintains a priority queue, and the cost of the queue is O(n log n) for n cars with one event each, while the cost of the simulation is O(n). But the constant involved with the queue is so small compared to the constant involved with accurate simulation that it takes more objects than the size of the universe for the queue to be more expensive.

through the intersection. In the accurate simulation such close proximity through an intersection is extremely rare, because the soonest one car can enter the intersection is when the car in front gets out the other side, typically moving too fast and too far ahead to catch.

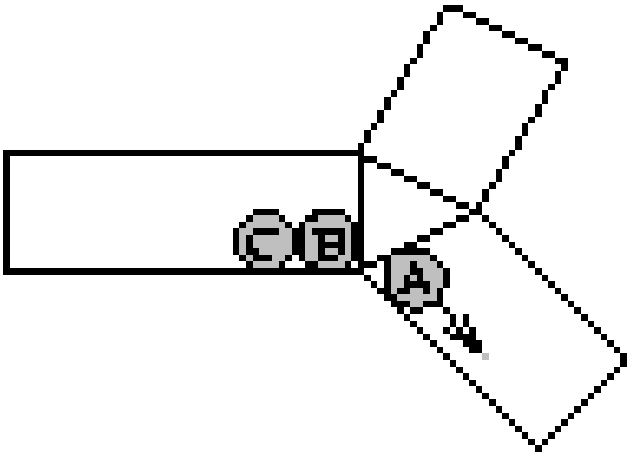

Figure 3: When a car, A, exits an intersection, it may also be necessary to consider the next car to enter the intersection, B, and the car stopped behind it, C.

Exit events are processed as follows (see Figure 3):

- An `Enter` event is scheduled for the car, A, for the soonest time that it could reach the end of the road. That time is computed using the pre-defined acceleration profile *assuming that no other car slows this one*. The only time another car is likely to slow this one is if the leader is forced to wait at the stop sign, and the follower catches it there. But in most such cases the leader will still be on the lane waiting to enter the intersection when the event we are scheduling comes up for processing, and according to the rules for `Enter` events above, we will delay moving this car into the intersection until the one in front makes space. There are still cases, however, when we will fail to detect a delay because the leading car has already entered the intersection when this optimistic `Exit` event is processed. In such cases the proxy will slightly reduce a car's travel time.
- An `Exit` event signifies that car A has left an intersection, so any car that is waiting to enter that same intersection is now free to do so. Hence, if there is a car, B, on the waiting list we move that car into the intersection, change its guarantee and schedule an `Exit` event for it as described in case 3 of the `Enter` event processing list above.
- If we moved car B into the intersection in the previous step, that car must have been waiting at the stop line, and there may be a queue of cars behind it. The next car, C, on the queue will have no event pending if it was stopped behind B, but it is now free to reach the intersection. Hence, we schedule an `Enter` event for car C, for the time it takes for a stopped car to go from first in line to the end of road, which can be computed based on the acceleration profile.

By processing events as above in the correct order, the proxy maintains the correct set of roads for every out of view car. Work is only performed when an event is processed, resulting

in efficient simulation. The various queues are correctly maintained, so we know which intersections are busy and which cars are where. This gives us enough information to reconstruct anything the viewer might infer from their knowledge of the model. The assumptions made in scheduling future events introduce some errors, but we claim that they are small, and our experiments described below support that. It remains to discuss how state is set when an object re-enters the view.

Generating Complete State

The proxy must supply complete state for cars that re-enter the view. This can happen in two ways:

- **Arrival:** the car may drive onto a visible road when it enters an intersection. But cars always enter an intersection from a standing start and aligned along the road, so we know its correct position (the end of the road), speed (zero) and orientation. We do not know the correct rotation angle for each wheel, which is determined by the length of the path traveled by the wheel. But we can safely assume that a viewer has no idea exactly which path a wheel has taken, so we set the rotations randomly.
- **Exposure:** the viewer may turn a corner and reveal an entire road and all the cars on it. We isolate the road, shifting it back in time to the entry time for the first car on the road, and simulating just the road forward from there to the current time, adding cars at the time they entered according to the proxy, and stopping all the cars in a queue at the end of the road (where they must be stopped, or they would not be on the road). This process generates highly accurate state for cars re-entering the view, but significant lag may result when a busy road comes into view. This lag could be reduced with a more aggressive approximation strategy, such as simply placing all the cars on the road stationary in a queue, at the expense of greater errors. This would certainly be a requirement for gaming.

The Simulation Loop

The primary simulation loop has to manage the transitions of cars from non-visible to visible and vice versa. It also has to run the event processing loop for the proxy simulation and call the in-view simulator to generate state for in view objects. One complication is that the in-view simulator must not simulate across events from the proxy, because one of those events might be a car entering the view, which would then need to be simulated. We assume that visible roads are marked for each frame before the simulation code is called. The outline of the simulation loop for one frame is:

1. Test all the cars that were visible on the previous frame to see if they are still visible now. If not, the car must be taken over by the proxy, so an event may have to be scheduled: if the car is in an intersection, schedule an `Exit` event as in case 3 of `Enter` event processing above; if the car is waiting at an intersection, we do nothing, as the car will be considered next when the car ahead of it clears the intersection; otherwise, schedule a `Enter` event based on the time the car entered the road.
2. Test all the roads that are currently visible to see if they were visible on the previous frame. Make all the cars on newly visible roads visible according to the **Exposure** rules above, and bring them under the control of the in-view simulator.

3. Repeatedly, run the in-view simulator up to the next event time and process the next event. Do this until the frame time is up.

Speedup Results

The city traffic environment using the proxy described above generates visually reasonable behavior when compared to a full model. It also achieves significant speedups, allowing the simulation of cities larger than could be handled with an accurate simulation alone. We performed a set of experiments to determine the proxy's efficiency, and another to measure its quality.

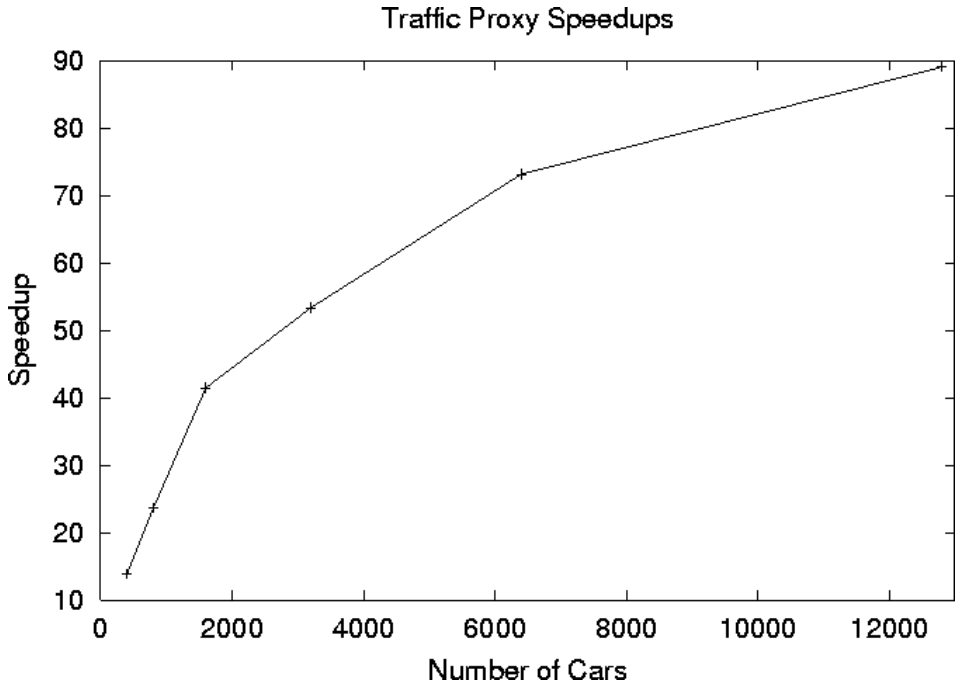

Figure 4: The speedups achieved by the proxy simulation. The proxy achieves almost two orders of magnitude speedup for 12800 cars in the simulation.

We determined the simulation computation speedup achieved as the number of cars in the simulation is increased, by comparing the time taken to simulate one frame of dynamics at 20fps (0.05s of simulated time per frame) with the full model and the proxy. The city size was increased along with the number of cars, keeping the overall density constant. For each data point we averaged timing data from four runs each of ten minutes duration with different sets of initial conditions. A different animated viewpoint was used for each run to simulate a moving viewer. All computations were performed on a Pentium III 800. The speedup is plotted in Figure 4. The data indicates that it is possible to fully simulate only 2000 cars in real-time on this machine, while the proxy could simulate 12800 cars using only 0.07 seconds of real time for each second of virtual time.

Measuring Quality

It is a hopeless task to measure the quality of a proxy simulation by comparing its precise behavior to that of the accurate simulation. Small differences at one point will lead to wild changes in the details of other motion. But the predictions made by a proxy for the motion of

cars do not need to exactly match what the full simulation would do – viewers expect variations in behavior and the details are not what matters. A good real world example is driving home from work. It is completely reasonable to call home and tell someone waiting that you are leaving work. The details of your drive will vary from day to day, and you won't always arrive at exactly the same time, but your estimate will still be reasonable. The person waiting for you does not care which other cars you came across on your drive home, only that you arrived in reasonable time.

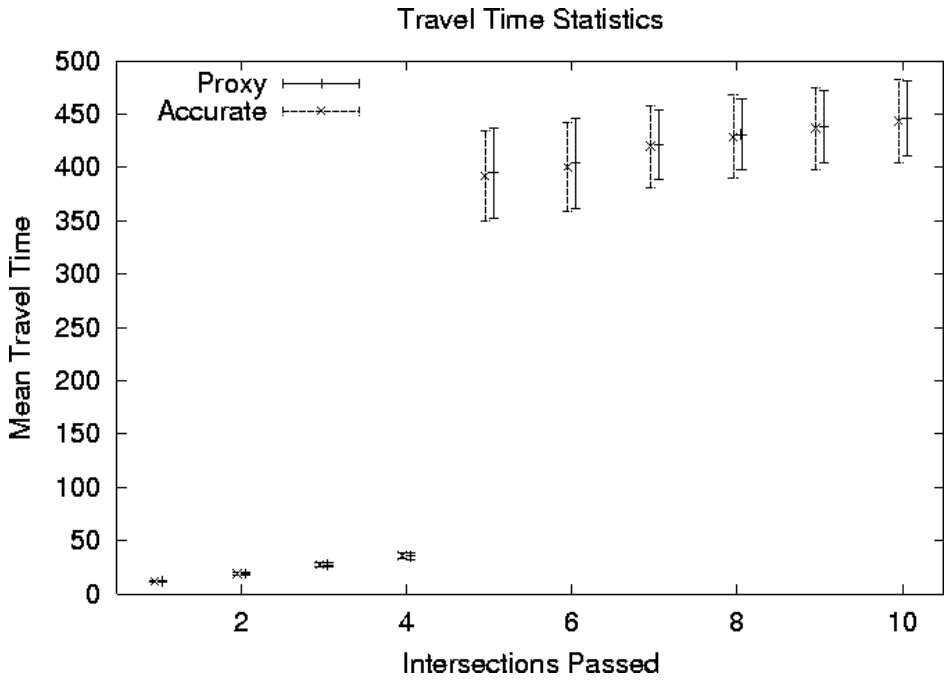

Figure 5: The average time taken by a car to reach various way-points on a fixed route, as estimated by the full model and the proxy model. The way-points are numbered from 1 to 10 across the bottom. The error bars represent one standard deviation. The averages were computed by fixing the path of one car, then running multiple simulations with the other cars starting out in different configurations. Ten simulation runs were averaged for each data point using the 1600 car city. The times estimated by the proxy are reasonably close to those estimated by the full model, with a roughly 1% difference in the estimated means, and almost the same variance. The proxy produces the smaller estimate, which is as expected. All the assumptions made in predicting event times in the proxy ignore potential delays. Without those delays cars under the proxy go faster. This could be remedied by explicitly increasing the predicted event times by an appropriate amount.

We can turn to statistics to formalize this idea and produce quantitative assessments. A viewer effectively takes measurements of certain things, like how long it takes to get somewhere. Those measurements will show variation depending on the details of what is happening. We can talk about the variation in terms of the mean behavior and the standard deviation. The accurate model produces one set of average behaviors, and the proxy will

produce another. *If the statistical behavior of the proxy simulation is indistinguishable from that of the accurate one, then the viewer cannot tell the difference.*

A tough test for the quality of the proxy is to examine the statistics it produces for travel times. We do this by fixing the path of one car, and running multiple simulations with the other cars starting in different configurations. We then compare the average time it takes the car to follow its path when the proxy is employed and when the accurate model is employed. This experiment is the virtual equivalent to determining how long it takes to drive home from work, and checking how both the accurate and proxy models estimate how that time varies over different days. The results are plotted in Figure 5. Note the similarity in the statistics, and the absence of significant bias over long periods of time (the mean travels times are almost the same).

These results indicate that the proxy achieves significant speedups while producing very accurate estimates of system behavior. Note, however, that the cost of the system still grows as more and more out of view objects are introduced. This is to be expected: we still do some work for every out of view object, so as more are added more time is spent working on them, reducing efficiency. The only way to avoid this drop is to ignore some objects completely, by adding and deleting them at a boundary.

RELATED REFERENCES

There are now a few papers out there on various aspects of simulation level-of-detail, or simulation culling. My colleagues and I have published previously on the topic of dynamics culling, which is an extreme case of level-of-detail in which no work at all is done for out of view objects[4]. However, the techniques in that paper are limited to objects that do not move over large distances, and so can be statically bound for visibility purposes.

Jessica Hodgins and Deborah Carlson first coined the term simulation level of detail[5]. They describe a hopping robot avoid-the-puck game in which robots that are further from the viewer are replaced with simpler dynamic models. They discuss the problem of ensuring that their approximations don't change the outcome of the game. Also on the theme of simulation level-of-detail, Setas, Gomes and Rebordão[6] describe tree models with several levels of detail for populating virtual terrain. The trees blow in the wind, with more distant trees using simpler (and cheaper) dynamic models.

The NueroAnimator technique described by Grzeszczuk and Terzopoulos[7], and the Synthetic Motion Capture approach by Yu and Terzopoulos[8] may also be viewed as simulation level of detail techniques, because both produce implementations that are relatively cheap to evaluate when compared to the accurate simulation. However, no attention is paid to the errors that these techniques introduce. In particular, the synthetic motion capture fish exhibit significantly, and noticeably, different behavior when compared to their fully simulated counterparts, which may be unacceptable in a gaming environment.

CONCLUSION

This paper has presented simulation level-of-detail, and proxy simulations in particular. It leaves open a number of questions. The specific example of traffic simulation demonstrates that many aspects of the proxy design are tightly tied to the particular system that is to be

approximated. Regardless, the broad outline in the Saving Work section provides a guide to employing similar techniques in other areas. Of particular interest is extending to proxy models for bots and other autonomous agents.

My colleagues and I have done some work on proxies for large numbers of path planning bots in a tile-based world. The bots must get from one point to another according to a user or AI's orders. The primary cost in such a system is path planning and collision avoidance. Our work uses random delays in travel times to simulate the effects of inter-object interactions, such as waiting for another bot to clear the path. High speedups result, and it is possible to simulate many thousands of bots in real time.

The problem of avoiding computation for out of view objects is similar to that of avoiding network traffic for out of view objects in networked multi-player games. In particular, to avoid sending data, you must know that which objects are visible for each user. Normally, this is done by a central server, which may lead to poor scaling in future systems. The problem in avoiding the server is that you need to know where an object is in order to know that you don't need its data, but of course you have to communicate to determine the location. By predicting future locations, as proxy simulations do, the position information would need to be sent less often, thus reducing communication overhead.

The primary take home point from this paper is that it is possible to save very large amounts of simulation time by avoiding unnecessary work, particularly work for out-of-view systems. Moreover, such work can be avoiding while maintaining a detailed, consistent world model.

---

[1] Joe Adzima, "Using AI to Bring Open City Racing to Life", Game Developer Magazine, Dec 2000, pp 26-31.

[2] Oded Sudarsky and Craig Gotsman, "Dynamic Scene Occlusion Culling", IEEE Transactions on Visualization and Computer Graphics, vol 5, no 1, 1999, pp 13-29.

[3] David Luebke and Chris Georges, "Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets", Proceedings of the 1995 Symposium on Interactive 3D Graphics, April 1995, pp 105-106.

[4] Stephen Chenney, Jeffrey Ichnowski and David Forsyth, "Dynamics Modeling and Culling", IEEE Computer Graphics and Applications, March/April 1999, pp 79-87.

[5] Deborah Carlson and Jessica Hodgins, "Simulation Levels of Detail for Real-time Animation", Proceedings of Graphics Interface 97, 1997, pp 1-9.

[6] M. N. Setas, M. R. Gomes and J. M. Rebordão, "Dynamic Simulation of Natural Environments in Virtual Reailty", Proceedings of SIVE95: The First Workshop on Simulation and Interaction in Virtual Environments, July 1995, pp 72-81.

[7] Radek Grzeszczuk and Demetri Terzopoulos, "NueroAnimator: Fast Neural Network Emulation and Control of Physics-based Models", Proceedings of SIGGRAPH 98, 1998, pp 9-20.

[8] Qinxin Yu and Demetri Terzopoulos, "Synthetic Motion Capture: Implementing a Synthetic Virtual Marine Environment", The Visual Computer, 1999, pp 377-394.