14

Phenomenal AI Level-of-Detail Control with the LOD Trader

Ben Sunshine-Hill

14.1 Introduction

Of all the techniques which make modern video game graphics possible, level-of-detail (LOD) management may very well be the most important, the most groundbreaking, and the most game-changing. While LOD seems like a rather boring thing to think of as "groundbreaking," in order to get the graphical quality we want in the world sizes we want, it's crucial to *not* render everything as though the player was two centimeters away from it. With conservatively chosen LOD transition distances, immense speedups are possible without compromising the realism of the scene in any way. Viewed broadly, even things like visibility culling can be considered part of LOD—after all, the lowest detail possible for an object is to not render it at all. Graphics programmers rely on LOD. It is, in a sense, "how graphics works."

AI programmers use some form of LOD, too, of course, but we don't really *rely* on it. We'll use lower quality locomotion and collision avoidance systems for characters more than ten meters away, or simulate out-of-view characters at a lower update rate, or (similar to visibility culling above) delete characters entirely when they're too far away. But while graphics programmers can use LOD without compromising realism, whenever *we* employ LOD, in the back of our mind, our conscience whispers, "That's just a hack … someone's going to *notice*." We use LOD only when we absolutely must, because we know that it's bringing down the quality of our AI.

There's another sense in which we don't rely on AI LOD. In graphics, LOD acts as a natural limit on scene complexity. The player can only be next to so many objects at once, and everything that's not near the player is cheaper to render, so framerate tends to even out. It's far from a guarantee, of course, but LOD is the first line of defense for maintaining the framerate. For AI, however, the techniques we'd really like to use often aren't feasible to run on more than a small handful of NPCs at once, and a cluster of them can easily blow our CPU time budget. There's no "LOD threshold distance" we could pick which would respect our budget *and* give most visible characters the detail we want.

So we use LOD. But it's not "how AI works."

What if LOD was smarter? What if it didn't even use distances, but instead could determine, with uncanny precision, how "important" each character was? What if it could read the player's mind, and tell the game exactly when to start using high-quality collision avoidance for a character, and when to stop? What if it knew which characters the players remembered, and which characters they had forgotten? And what if its LOD selections *always* respected the CPU time budget when NPCs decided to cluster around the player, but always made good use of the time available when they didn't?

Well, then, we could *trust* LOD. We could use techniques as expensive as we wanted, because we could rely on LOD to keep them from blowing our budget. We could move away from the endless task of tuning LOD thresholds, and hardcoding hack after hack to compensate for the endless special cases where our LOD thresholds weren't enough. LOD could become "how AI works."

That, in a nutshell, is the LOD Trader. It can't read the player's mind, but its simple heuristics are light-years beyond distance thresholds in determining how important a character is to the player. Rather than relying on fixed transition rules, it treats the entire space of detail levels—for all the AI features we'd like to control—as a puzzle to be solved each frame. It attempts to maximize the realism of the game simulation without going over the computation budget, and does it in a remarkably short period of time (tens of microseconds).

It's not magical. Its heuristics are far from infallible, and the optimality of its LOD solutions is approximate. But it is worlds beyond distance-based LOD, and the first time it outsmarts you—its LOD reductions becoming subtle and then invisible—you'll wonder a bit.

14.2 Defining the Problem

The first thing to do when attacking a problem like "optimal LOD selection" is to figure out what we mean by *optimal*. The graphics guys don't need to do this, because their LOD selections can be made *perfectly*—they can transition far enough away that they're not giving up any realism. But every detail reduction we make is going to potentially reduce realism, so we need to define, in a numeric sense, what we're trying to maximize, and what our constraints are. We need to come up with a metric, a system of units for measurement of realism. Yikes.

Well, let's grab the bull by the horns. I claim that what we're trying to do is pick a detail level for each AI feature, for each character, to minimize the *probability* that the player will notice an issue, an unrealistic reduction in detail. This helps nail things down, because we all know all about probabilities and how to compare and do arithmetic with them. In this model, Choice A won't be "a little less realistic" than Choice B, but will rather be "a little less likely to be noticed." We'll refer to the event of the player actually noticing an issue as a

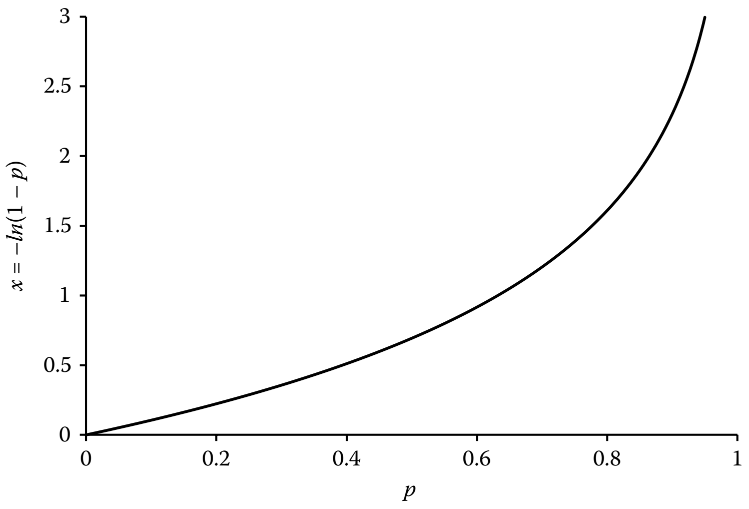

Figure 14.1

Logarithmic probability compared to linear probability.

Break in Realism (BIR). A BIR only occurs when the player notices the issue; just reducing the detail of an entity isn't a BIR if she doesn't notice it as unrealistic.

14.2.1 Diving Into X-Space

Let's make things a little cleverer, though. Suppose that the probability of the user noticing that some entity is unrealistic is p. Rather than work with that number directly, we'll work with the number $x = -\log(1-p)$. That's the negative logarithm (base whatever, let's use the traditional e) of the *complement* of the probability—the probability of getting away with it, of the user *not* noticing. A plot of p versus x is shown in Figure 14.1. As p goes up, x goes up. If p is zero, x is zero; as p approaches 1, x approaches infinity. (The inverse equation is $p = 1 - e^{-x}$.)

Why this complication? Well, actually, it's a simplification; x turns out to be much better-behaved than p. First, if we have two potential sources of unrealistic events, with (independent) probabilities of being noticed p_1 and p_2, and we want to know the total probability p_{tot} of the user noticing either one, that's $p_{tot} = p_1 + p_2 - p_1 p_2$, not especially nice. For three sources it gets even uglier: $p_{tot} = p_1 + p_2 + p_3 - p_1 p_2 - p_1 p_3 - p_2 p_3 + p_1 p_2 p_3$. In contrast, $x_{tot} = x_1 + x_2 + x_3$. Second, this transformation gives us a better interpretation of phrases like "twice as unrealistic." If some event has probability $p_1 = 0.6$ of being noticed, and some other event is "twice as unrealistic" as p_1, what is its probability? Clearly it can't be 1.2—that's not a valid probability. But in x-space things work out fine: $x_2 = 2x_1$, which leads to $x_1 = 0.91$, $x_2 = 1.82$, $p_2 = 0.84$. What is 0.84? It's the probability of noticing it either time, if the first event happened twice. It's extremely useful to be able to describe probabilities relative to each other in that sort of an intuitive way, because it's a lot easier to do that than to determine absolute probabilities. x-space gives us well-behaved addition and multiplication, which (as we'll see later) are crucial to the LOD Trader. In practice, there's actually very little reason to work with p at all.

14.3 Criticality and Probability

The next thing we have to look at is what we'll call the *criticality* of a character. That represents how "critical" the character's realism is to the realism of the scene as a whole—or, if you like, how critical the player is likely to be about the character's detail level. In a sense, that's what distance stood for when we were using distance-based LOD: All else being equal, a closer character is more critical to the scene than a farther character.

But all else is *not* equal. There are other things to use in sussing out criticality. It would be great if we could hook up eye trackers and EEGs and Ouija boards to the player, but even without that, there's plenty of metrics we can pull from the game to help estimate the criticality of a given entity to the player at a given time.

Before we go further, though, there's a really key thing about BIRs, and about criticality, to realize: not all unrealism is alike. Consider two characters. One is a suspected assassin who the player has been following at a distance for some time. The other is a random villager crossing the road a few feet ahead of the player. Both characters are *important*, in a sense. The first character is clearly the object of the player's conscious attention; if we can only afford high-quality path planning for one character, it had better be that one, because the player is more likely to notice if he's wandering randomly. But the second one occupies more screen space, and possibly more of the player's visual attention—if we can only afford high-quality locomotion IK for one, it should probably be that one.

14.3.1 A Field Guide to BIR's

It's tempting to throw in the towel at this point, concluding that there are as many kinds of unrealism as there are AI features whose detail level we'd like to move up and down. But I think there's a small set of categories that nearly all BIR's fall into. Any given reduced detail level will create the potential for a BIR in at least one of these categories, and sometimes more. The reason to categorize things like this is because each category of BIR can have its own criticality model.

14.3.1.1 Unrealistic State

An *unrealistic state (US)* BIR is the most immediate and obvious type of BIR, where a character's immediately observable simulation is wrong. A character eating from an empty plate, or running in place against a wall, or wearing a bucket on his head creates the potential for an unrealistic state BIR. (Not a *certainty*, mind you—the player might not be looking.) US's don't require any long period of observation, only momentary attention, and the attention need not be voluntary—the eye tends to be drawn to such things.

14.3.1.2 Fundamental Discontinuity

A *fundamental discontinuity (FD)* BIR is a little more subtle, but not by much: it occurs when a character's current state is incompatible with the player's memory of his past state. A character disappearing while momentarily around a corner, or having been frozen in place for hours while the player was away, or regaining the use of a limb that had been broken creates the potential for a fundamental discontinuity BIR. These situations can cause US BIR's too, of course. But even if the character is not observed while they happen, as long as the player remembers the old state and later returns, the potential for an FD BIR remains.

14.3.1.3 Unrealistic Long-Term Behavior

An *unrealistic long-term behavior (ULTB)* BIR is the subtlest: It occurs only when an extended period of observation reveals problems with a character's behavior. A character wandering randomly instead of having goal-driven behaviors is the most common example of an unrealistic long-term behavior BIR, but so is a car that never runs out of gas. At any given time, only a small handful of characters are likely to be prone to ULTB BIR's.

14.4 Modeling Criticality

Let's see about coming up with criticality models for these different categories. Each model calculates a *criticality score* as the product of several factors, some of which are shared between multiple models.

For unrealistic state, the factors are observability and attention. *Observability* comes closest to graphical LOD: it measures how feasible it is for the player to see the character in question. *Attention* is self-evident: it estimates how much attention the player is paying to a particular character. As you might guess, it's the most difficult factor to estimate.

For fundamental discontinuity, the two related factors are memory and return time. *Memory* estimates how effectively the player has memorized facts about a character, and how likely they are to notice changes to the character. *Return time* acts as a modifier to the memory factor: It estimates how attenuated the player's memory for the character will be when she returns to the character, or even if she will ever return at all.

For unrealistic long-term behavior, the three factors are attention, memory, and duration. Attention and memory have already been introduced (note that the return time factor is not acting on memory here); the last one, *duration*, simply refers to how much time and attention the player has devoted to that character.

There's the cast of characters. Now let's come up with actual equations for each one. Note that later factors will often use earlier factors in their input; the order we've listed them in is a good order to calculate them in.

Before we go into these, though, we need to introduce a tool which will be used in a lot of them: the *exponential moving average (EMA)*. The EMA is a method for smoothing and averaging an ongoing sequence of measurements. Given an input function $F(t)$ we produce the output function $G(t)$. We initialize $(0) = F(0)$, and then at each time t we update $G(t)$ as $G(t) = (1-\alpha)F(t) + \alpha G(t - \Delta t)$, where Δt is the timestep since the last measurement. The α in that equation is calculated as $\alpha = e^{-k \cdot \Delta t}$, where k is the *convergence rate* (higher values lead to faster changes in the average). You can tune k to change the smoothness of the EMA, and how closely it tracks the input function. We're going to use the EMA a *lot* in these models, so it's a good idea to familiarize yourself with it (Figure 14.2).

14.4.1 Observability

This is a straightforward one—an out-of-view character will have an observability of 0, a nearby and fully visible character will have an observability of 1, and visible but faraway characters will have an observability somewhere in the middle. For character i, you can calculate this factor as proportional to the amount of screen space (or number of pixels) p_i taken up by the character, divided by some "saturation size" p_{sat} referring to how big a character needs to get before there's no difficulty observing them, and limited to 1: $O_i = min(p_i / p_{sat}, 1)$.

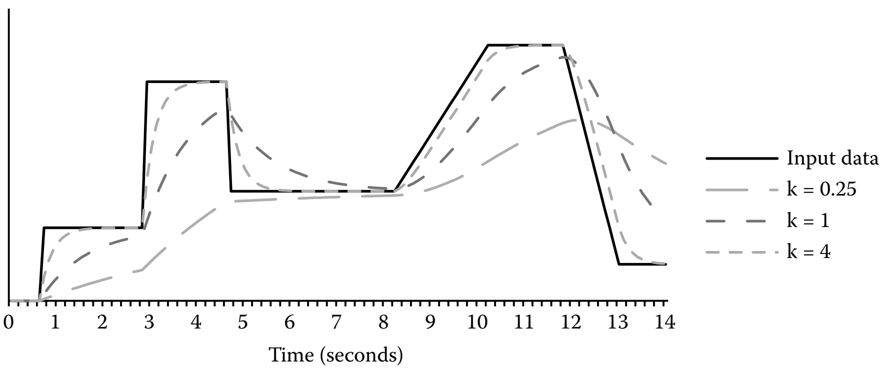

Figure 14.2

The exponential moving average of a data set, with various convergence rates.

We used the amount of screen space taken up by a fully visible character 4 meters away from the camera as p_{sat}. A smaller saturation value may be more appropriate for games intended for high-definition displays.

14.4.2 Attention

As mentioned earlier, attention is the most difficult factor to estimate. There are two steps in determining attention: estimating attempted attention, and applying the effect of interference.

As a first pass, attempted attention $\hat{A}_i$ can be calculated as the EMA of observability: $\hat{A}_i(t) = \alpha \hat{A}_i(t - \Delta t) + (1 - \alpha) O_i(t)$. For observability, you should tune k to have a rapid falloff; we used $k = 2$, which provides a 95% falloff in 1.5 seconds.

You can mix other things into the attempted attention model to improve its accuracy, though. One player behavior strongly associated with attention is *focusing*, where the player attempts to keep the character centered in the camera view. This can be calculated as the EMA of the dot product between the camera's forward vector and the vector to the character, then multiplied by observability. The convergence rate k should be much lower for this term. Other, more game-specific sources can be drawn on for attempted attention as well, such as how much of a threat the character is, or how rapid his motions are. The weighted sum is then used as the attempted attention estimate. For the simple observability-and-focusing model, we found weights of 0.7 and 0.3, respectively, predicted attention well.

The player's attention is not an unlimited resource; the more things they need to concentrate on, the less well they concentrate on each one. Once attempted attention has been calculated for all characters, you should sum the factors up, producing the *total attentional load L*. To compensate for the nonlinearity of this interference effect (as well as interference from sources outside the game), you should also add a constant *ambient attentional load* to L. The actual attention estimate for each character is then the ratio of their attempted attention to the total attentional load: $A_i = \frac{\hat{A}_i}{L}$, where $L = \hat{A}_{amb} + \sum_{j=1}^{n} A_j$. The value $\hat{A}_{amb}$ is a difficult factor to tune; I would suggest setting it such that it represents about 1/3 of L during the most attention-intensive situations in your game. Increasing it will tend to bias resources towards background characters; decreasing it will bias resources towards foreground characters.

14.4.3 Memory

Our model of memory is a simple one based on cognitive models for an experimental memory task known as *associative recognition*, and on a phenomenon known as *retroactive interference*. In general, a player's memory M_i of a character will tend toward their current attention A_i for that character over time. While memorizing (that is, while memory is increasing), the convergence rate will be a fixed $k = k_m$. While forgetting (that is, while memory is decreasing), we'll use a different convergence rate $k = k_f L$, where L is the total attentional load. So it's an EMA again, but with $k = k_m$ if $M_i < A_i$, and $k = k_f L$ if $M_i > A_i$. We used $k_m = 0.6$ and $k_f = 0.001$, which (for our highest observed attentional load) resulted in a 95% memorization rate in about 5 seconds, and a 50% forgetting rate in 10 seconds under a high attentional load. The latter tended to overestimate long-term retention of memory for characters; this wasn't a problem for us, but you may need to tune that factor upward.

14.4.4 Return Time

Return time is much more objective, because it estimates a player's actions rather than her thoughts. It's somewhat misnamed: the output is not an expected number of seconds until the player's return, but rather an expected attenuation factor to current memory at the moment the player does return, as well as to the probability of the player ever returning at all. It's based on something known as the *Weibull hazard function*. The derivation is rather involved, but the resultant equation is $R_i = k(L)^{-k} e^{Lt_0} \Gamma(k, Lt_0)$. L is the expected future attentional load (you can either use the current attentional load, or smooth it with an EMA), and t_0 is the time since the character was last visible (that is, had an observability greater than 0). k is a tweakable parameter which you can experimentally determine by fitting observed return times to a Weibull distribution; the value we determined was approximately 0.8, and we think that's unlikely to differ much in other games. $\Gamma(s, x)$ is the *upper incomplete gamma function*. Implementations of this function are included in many popular math libraries, including Boost.Math.

14.4.5 Duration

To finish the criticality model factors on a nice, easy note, duration is the total amount of attentive observation: the integral of $O_i A_i$ over time. That is, $D_i(t) = D_i(t - \Delta t) + O_i A_i \Delta t$.

14.4.6 Modeling Costs

Compared to criticality scores, costs are much more straightforward and objective to model. Simply set up your game so that a hundred or so characters are simulating using a particular detail level, then compare the profiler output with a run using a different detail level.

14.5 LOD's and BIR's

Again, the reason to categorize BIR's at all is because each category can have its own criticality score. Having a bit of foot skate is probably less noticeable than running in place against a wall: The latter behavior is more obvious. We will refer to it as having higher *audacity*—that is, a lower LOD will be more *audaciously* unrealistic. But given two characters, if one is twice as likely to be noticed foot skating as the other, it is also twice as likely to be noticed running in place as the other. We don't need to have separate criticality models for the two behaviors, because the same factors affect both. So for a particular category of BIR (in this case, unrealistic state) each character has a criticality score, and each detail level (in this case, a type of local steering which can lead to running into walls) will have an audacity score. To sum up, we have three categories of BIRs, three criticality scores per character, and three audacity scores per detail level. (Note that we're only looking at one AI feature for now—in this example, local steering. We'll move to multiple features later.)

By the way, that "twice as likely" mentioned above should remind you of Section 14.2.1, and not by accident. For a single BIR category, we can think of a detail level's audacity score in that category as a base probability in x-space (for some "standard-ly critical character"), and a character's criticality score in that category as a multiplier to it, the product being the probability of *that* character using *that* detail level causing a BIR in *that* category.

But it gets better! Or, at least, more elegantly mathy! Since we're assuming independence, the probability of that character/detail level combination causing a BIR in *any* category is the sum of the products over all three categories. If we stick the detail level's audacity scores for all categories into a vector, call it the *audacity vector A*, and the character's criticality scores for all categories into another vector, call it the *criticality vector C*, the total BIR probability for the combination is given by the dot product $A \cdot C$. Linear algebra—it's not just for geometry anymore!

14.6 The LOD Trader

Now that we have our model, it's time to introduce the LOD Trader algorithm itself. We'll be adding more capabilities to it through the rest of the article; this initial version controls a single LOD feature (say, how pathfinding is performed) with multiple available detail levels, and limits a single resource (say, CPU time).

As you might guess from the name, the LOD Trader is based on a stock trader metaphor. Its "portfolio" consists of all the *current* detail levels for all characters. The LOD Trader's goal is to pick a portfolio that minimizes the total probability of a BIR. Of course, the best possible portfolio would be one where all characters had the highest detail level; but it must additionally limit the total cost of the portfolio to its available resources, so that's (usually) not an option.

Each time the LOD Trader runs, it evaluates its current portfolio and decides on a set of trades, switching some characters to a higher detail level and other characters to a lower detail level, as their relative criticalities and the available resources change. Remember, the total BIR probability for a character being simulated at a particular detail level is the dot product of the character's criticality vector and the detail level's audacity vector. So the LOD Trader will try to pick detail levels that have low audacity scores corresponding to a character's high criticality scores.

Note that for a given trade, we can find a *relative* cost, but also a *relative audacity* which is the difference in the two detail levels' audacity vectors. Just as the absolute BIR probability for a particular detail level is the dot product of the character's criticality vector and the detail level's audacity vector, the relative BIR probability for a particular trade is the dot product of the criticality vector with the change in audacity. We'll refer to increases in BIR probabilities as *upgrades*, and to decreases in BIR probabilities as *downgrades*.

The heuristic the LOD Trader uses to guide its decisions is a simple *value*-based one: units of realism improvement divided by units of resource cost. If a character is simulated at a low detail level, the value of upgrading it to a high detail level is the relative reduction in the total probability of a BIR divided by the relative increase in resource cost. Valuable upgrades will have a large reduction in BIR probability and a low increase in cost. Likewise, if a character is at a high detail level, the value of downgrading it to a lower detail level is the relative increase in the total probability of a BIR divided by the relative reduction in cost; valuable downgrades will increase BIR probability only slightly and decrease the cost by a lot. To keep the math simple, we'll toss in a negative sign, so that upgrade values are positive, and more valuable upgrades have larger magnitude values. For downgrades, values are positive as well, but the most valuable downgrades will have *smaller* magnitudes. (The exception, which should be handled specially. Under some circumstances, a detail upgrade will result in a reduction in cost, or a detail downgrade may result in an increase in cost. The former should always be chosen; the latter never should.)

During a single run, the LOD Trader runs one or more iterations. In each iteration, it hypothesizes making a set of trades (both upgrades and downgrades) that would respect the resource limits and that might result in an overall reduction in BIR probability. If the hypothetical set of trades does, in fact, reduce BIR probability, the trades are actually performed, and the trader continues to iterate, to try to find additional trades to make. Once it finds a hypothetical set of trades which does *not* reduce BIR probability, it stops (and does not make those trades).

The algorithm for choosing the hypothetical set of trades is simple. First it considers upgrades. It repeatedly picks the most valuable available upgrade to add to its set of trades until it has *over*spent its resource budget. Then, it repeatedly picks the most valuable available downgrade to add to its set of trades until it has not overspent its resource budget. Upgrades and downgrades are stored in priority queues to reduce the search cost. Pseudocode for the LOD Trader is in Listing 14.1; remember that this is the initial version, and we'll add more features and improve performance later.

14.6.1 Multiple Features

One of the most useful effects of the multicategory criticality modeling is the ability to control different *kinds* of LOD at the same time. For instance, we can control pathfinding quality (the quality of which primarily affects the probability of a ULTB BIR) and hand IK (which affects the probability of a US BIR). Put differently, we'd like to control multiple *features* (AI systems whose detail is set on a per-character basis). Of course, we could do that by running multiple LOD Traders, one for each feature. But then we'd have to give each one a separate budget; there'd be no way to automatically shift resources between pathfinding and IK as one or the other became important, or to trade a downgrade in the pathfinding quality of one character for an upgrade in the IK quality of another.

Another problem with the multi-Trader approach is that certain features might depend on each other. For instance, we might control both a character's basic behavior (goal-driven or just standing around) and his pathfinding (high quality, low quality, or disabled). Goal-driven behavior and disabled pathfinding, of course, aren't compatible

Listing 14.1. Initial code for the LOD Trader, supporting only one LOD feature and one resource type.

```
def runLODTrader(characters, lodLevels, availableResource):
    acceptedTrades = []
    while True:
        upgrades, downgrades = calcAvailableTrades(characters,
lodLevels) # returns p-queues, sorted by value
        hypTrades = []
        charactersWithTrades = []
        hypBenefit = 0
        hypAvailableResource = availableResource
        while not upgrades.empty() and availableResource > 0:
            upgrade = upgrades.pop()
            hypTrades.append(upgrade)
            charactersWithTrades.append(upgrade.character)
            hypAvailableResource -= upgrade.costIncrease
            hypBenefit += upgrade.probDecrease
        while not downgrades.empty() and availableResource < 0:
            downgrade = downgrades.pop()
            if downgrade.character in charactersWithTrades: continue
            hypTrades.append(downgrade)
            charactersWithTrades.append(downgrade.character)
            hypAvailableResource += downgrade.costDecrease
            hypBenefit -= downgrade.probIncrease
        if hypAvailableResource >= 0 and hypBenefit > 0:
            acceptedTrades += hypothesizedTrades
            availableResource = hypAvailableResource
        else
            return acceptedTrades
```

as detail levels, but there would be no effective way to coordinate the two traders to avoid that result.

Instead, we let a single LOD Trader balance detail levels of all the features at the same time. A character's current "state" as seen by the LOD Trader will not be a single detail level, but will be the combination of their current detail levels for *all* the features. We refer to a set of levels for all features, which respects all the inter-feature constraints, as a *feature solution*. Rather than picking individual feature transitions, we will pick *feature solution transitions* from one feature solution to another, each of which may result in several feature transitions. For each feature solution, we'll precompute and store a list of possible upgrade transitions and possible downgrade transitions, so that we know which ones to look at for a character currently at any particular feature solution.

If the set of features is small, this has little impact on the algorithm; the only major change is the need to check that we don't pick multiple upgrade or downgrade transitions for a single character. However, the number of feature solutions increases exponentially with the number of features. Since we would have to evaluate every character/feature solution combination and insert it into our priority queues, this could result in a lot of computation, and a *lot* of memory usage. Most of this work would be wasted, because it would be spent on evaluating lots of expensive upgrades for lots of faraway, unimportant characters—ones we should know we won't upgrade in *any* way, given their teeny criticality vectors.

Listing 14.2. Expanding a character.

```
def expandCharacter(char, transType):
    bestRatio = None; bestTrans = None
    for trans in char.featureSolution.availableTransitions[transType]:
        ratio = dotProduct(char.C, trans.A)/trans.cost
        if isBetterRatio(ratio, bestRatio):
            bestRatio = ratio; bestTrans = trans
    return bestRatio, bestTrans
```

14.6.2 The Expansion Queue

Instead, we'll use a new, lazier strategy. Instead of a priority queue of upgrade transitions, we'll start with a priority queue of *characters*; we'll refer to this as the *expansion queue*. The sort key we'll use for the expansion queue will be the *expansion heuristic*, which estimates the *best possible* value that could *possibly* be attainable by *any* transition for that character. This value represents an upper limit on transition value for each character, and may be over-optimistic, but it will never be pessimistic; in this sense it is similar to an admissible heuristic for A* search. We'll select upgrade transitions by "expanding" the character at the front of the queue (the one with the highest expansion heuristic) into all of its possible upgrade transitions, and selecting the most valuable one. The pseudocode for expanding a character is shown in Listing 14.2.

Because the heuristic may be over-optimistic, we can't guarantee that the character at the front of the expansion queue actually has the most valuable upgrade transition among all characters. To compensate for this, we will continue expanding characters from the expansion queue, even after we've overspent our resource budget. Once we're overspent, each time we expand a character and choose a new upgrade for our set of hypothetical trades, we'll then repeatedly remove and discard the lowest-valued trade from the upgrades, until we're only overspending by one trade (that is, removing another lowest-valued trade would make us underspend). To make this efficient, we'll store the set of chosen hypothesized upgrades itself as a priority queue, ordered so that the front element has the *lowest* value. Often, the just-expanded, just-inserted upgrade will itself be the lowest-value trade, and will be removed immediately after being added.

When can we stop doing this? When the worst-value transition already picked—the one at the front of the hypothesized upgrades queue—has a higher value than the heuristically predicted value at the front of the expansion queue. Because of the admissibility of the heuristic, we know at this point that we'll never find a better upgrade than we've already picked, so we can stop without expanding any more characters, and the chosen set of upgrades is the ones remaining in the hypothesized upgrades queue. In practice, this happens quite quickly.

The downgrade phase works analogously: we keep an expansion queue sorted by smallest possible value, and pick the lowest value downgrade for each expanded character, inserting it into our hypothesized downgrade queue. Once our resource constraint is no longer violated, after each pick, we continue to pop largest-value downgrades off the hypothesized downgrades queue until popping the next one would violate the resource constraint. Once the character at the front of the expansion queue has a larger heuristic value than the downgrade at the front of the hypothesized downgrades queue, we stop expanding characters.

14.6.3 Pruning Transitions

Before we get to the best-possible-value heuristic, let's look at a certain class of feature solution transitions. These transitions are what one might call "penny-wise and pound-careless." Or perhaps one might technically refer to them as *stupid*. For instance, a feature transition that upgraded animation IK to the highest possible quality, but kept collision avoidance turned off, would be stupid. It's *allowed*, yes, but it should never be chosen; well before you decide to spend CPU time on high-quality IK, you should first decide to keep the character from walking through walls. The possibility of stupid transitions isn't a problem for the LOD Trader, because it won't ever choose them, but it does spend time evaluating them. As it turns out, a lot of solution transitions—well over half of them, in our experience—are stupid.

What typifies a stupid transition? In a mathematical sense, it's being "strictly dominated" by other transitions; that is, regardless of circumstances, there's always a more valuable transition. Let's examine how we go about identifying those, so we can ignore them.

Remember, the value (for upgrades) is probability benefit—the dot product of criticality and audacity—divided by relative cost increase. To put this in equation form, for switching character i from feature solution α to feature solution β, we'll refer to the change in resource cost as $r_{\alpha,\beta} = r_\beta - r_\alpha$, the change in audacity as $A_{\alpha,\beta} = A_\beta - A_\alpha$, and the resultant value as $V_{i,\alpha,\beta} = -\left(A_{\alpha,\beta} \cdot C_i\right) / r_{\alpha,\beta}$. (Remember the negative sign—we want positive values for upgrades, even though higher quality is lower audacity.) That depends on the criticality vector C_i. The transition from α to β is "stupid" if, for *any possible* criticality vector, there's some *other*, better feature solution χ such that $V_{i,\alpha,\beta} < V_{i,\alpha,\chi}$.

For a given feature transition, figuring out whether it is strictly dominated can be formulated as a linear programming problem. Alternatively, you can just generate a large number of random criticality vectors, and find the best transition for each. Any transition which isn't chosen at least once is assumed to be strictly dominated, and removed from the list of upgrades to evaluate at that starting feature solution. For stupid downgrades the same thing is done but the "best" transitions are the ones with the smallest-magnitude value.

14.6.4 The Expansion Heuristic

Returning to the heuristic, we'll use it for the expansion queue—that is, estimating the best possible value for any transition for each character. Let's look at that value formula again: $V_{i,\alpha,\beta} = \left(A_{\alpha,\beta} \cdot C_i\right) / r_{\alpha,\beta}$. Rearranged, it's $V_{i,\alpha,\beta} = C_i \cdot W_{\alpha,\beta}$, where $W_{\alpha,\beta} = A_{\alpha,\beta} / r_{\alpha,\beta}$. For a particular starting feature solution α, we can gather the W-vectors for all upgrade transitions into a matrix $\mathbf{W}_\alpha = \left[W_{\alpha,\beta}, W_{\alpha,\gamma}, \cdots\right]$. Then we can prune it, removing any column that does not have at least one entry greater than the corresponding entry in a different column. Once we have the W-matrix stored for each starting feature solution, we can calculate the heuristic value quite quickly, as the maximum entry in the vector $C_i\mathbf{W}_\alpha$. We do the same thing for downgrade transitions, using a separate matrix for those. (Remember, for downgrades we want smaller values, so we prune columns that do not have at least one *lower* entry.) This value heuristic calculation is shown in the pseudocode of Listing 14.3.

Listing 14.3. Calculating the value heuristic for a character.

```
def calcValueHeuristic(char, transType):
    elems = matrixMul(char.C, char.featureSolution.W)
    if transType == 'upgrade': return max(elems)
    else: return min(elems)
```

14.6.5 Multiple Resources

CPU time may not be our only resource constraint. For instance, suppose one of the features we'd like to control is whether a character remembers other characters who have been friendly or hostile towards him. That could easily become a large RAM sink, so we'd like to keep our memory usage under a particular budget as well. This is a situation where we might be able to use multiple LOD Traders, one for each resource type, but it's possible that a single feature might have ramifications for more than one resource. As before, we'd like a single trader to do all the work of balancing things. The cost of a detail level will now be vector-valued, as will the total cost of a feature solution and the relative cost of a feature solution transition.

The first thing we have to do is adapt our value heuristic. "Dividing by cost" doesn't work anymore because cost is a vector. We'll use a *resource multiplier vector M* to generate a scalar metric for cost. During the upgrade phase, the resource multiplier for each resource type is the reciprocal of the amount of that resource, which is currently unused. If CPU time is at a premium but there's plenty of free memory, the resource multiplier vector will have a larger entry for CPU than RAM. In case a resource is neither over- nor underspent, it should have a large but not infinite resource multiplier. During the downgrade phase, the resource multiplier is directly proportional to the amount of overspending; resources that are not overspent have a resource multiplier of 0. The resource multiplier vector is recalculated before each upgrade and each downgrade phase, but not during an upgrade or downgrade phase.

Next, we need to adapt our stopping criteria. Rather than picking upgrades such that the only resource is overspent by a single upgrade, we will pick upgrades until *any* resource is overspent. We will then pick downgrades until *no* resources are overspent.

We also need to adapt our definition of stupid feature solutions, and our expansion heuristic. When determining whether a feature solution will ever be chosen, we need to check it against not only a large number of random criticality vectors, but also resource multiplier vectors. And when generating $W_{\alpha,\beta}$, we need to maximize it over all possible resource multiplier vectors: $W_{\alpha,\beta} = A_{\alpha,\beta} / \left(\min_M M \cdot r_{\alpha,\beta} \right)$. (For both of these, you should consider only *normalized* resource multipliers.) In order to get the best performance results out of both feature solution pruning and the expansion heuristic, you should come up with expected limits on the ratios between resource multipliers, and clip actual resource multipliers to these limits.

Finally, note that some feature solution transitions will have both positive and negative resource costs: these should be allowed as upgrades, but not allowed as downgrades.

Listing 14.4. Making the expansion queue over all characters.

```
def makeExpansionQueue(characters, M, transType):
    if transType == 'upgrade': expansionQueue = maxQueue()
    else: expansionQueue = minQueue()
    for char in characters:
        valueHeuristic = calcValueHeuristic(char, M, transType)
        expansionQueue.insert(char, valueHeuristic)
```

Listing 14.5. Final pseudocode for the LOD trader, supporting multiple features and resource types.

```
def runLODTrader(characters, availableResources):
    acceptedTrades = []
    while True:
        M = calcResourceMultiplier(availableResources)
        hypUpgrades, hypAvailableResources =
selectTransitions(characters, M, 'upgrade', availableResources)
        M = calcResourceMultiplier(availableResources)
        hypDowngrades, hypAvailableResources =
selectTransitions(characters, M, 'downgrade', hypAvailableResources)
        hypTrades = hypUpgrades + hypDowngrades
        if calcTotalBenefit(hypTrades) > 0:
            acceptedTrades += hypTrades
            availableResources = hypAvailableResources
        else:
            return acceptedTrades
def selectTransitions(characters, M, transType, availableResources):
    expansionQueue = makeExpansionQueue(characters, M, transType)
    if transType == 'upgrade': transitionHeap = minQueue()
    else: transitionHeap = maxQueue()
    while availableResources.allGreaterEqual(0) or
isBetterRatio(expansionQueue.peekKey(), transitionHeap.peekKey()):
        char = expansionQueue.popValue()
        bestRatio, bestTrans = expandCharacter(char, M, transType)
        transitionHeap.insert(bestTrans, bestRatio)
        availableResources -= bestTrans.costs
        while (availableResources + transitionHeap.peekValue().costs)).
anyLess(0):
            discardedTrans = transitionHeap.popValue()
            availableResources += discardedTrans.costs
    return transitionHeap.values(), availableResources
```

14.6.6 Putting it All Together

Listing 14.5. shows the updated pseudocode for multiple features and resources.

14.6.7 Other Extensions to the LOD Trader

In addition to constraining which levels for different features can be used together, it's possible to constrain which levels of a single feature can transition to which other features. For instance, you might transition a character from prerecorded animation to fully dynamic motion, but not be able to transition back from dynamic motion to prerecorded animation. This can be done simply by discarding feature solution transitions that include such a transition.

It's also possible to attach costs and audacities to a feature transition itself, instead of just to feature levels. Attaching costs to transitions can be useful if the transitioning process itself requires nontrivial computation; attaching audacity can be useful if the transition is liable to produce a visible "pop" or if it involves a loss of character information which could later lead to a FD or ULTB BIR.

In some situations it's useful to introduce the concept of a "null" detail level for a particular LOD feature. For instance, the "standing around" behavior detail level would only be compatible with the "null" locomotion detail level, and the "doing stuff" behavior detail level would be compatible with all locomotion detail levels *except* the null level.

An unusual but useful application of the LOD Trader is as an alternative to the "simulation bubble" often used to delete faraway characters. This is done by means of an "existence" LOD feature, with "yes" and "no" levels, where the "no" level is compatible with all other LOD features being "null" and has zero audacity and zero cost, but where the transition from "yes" to "no" itself has US and FD audacity. When a character transitions to the "no" existence level, it is removed.

Another unusual but useful application is to consider "save space" as an additional resource. This can be constrained only when the game is about to be saved, and ensures that the most useful and memorable bits of game state are kept around when saving to a device with limited space.

The LOD Trader can also be leveraged in multiplayer games, simply by adding up the criticality for a given character based on all players currently observing that character. Because of the additive nature of the x-space probabilities, this will result in correct estimation and minimization of BIR probability. Additionally, the LOD Trader can be used to ration network bandwidth by controlling update rates and update detail for different characters; in this situation, a separate LOD Trader instance is run for each player.

Finally, not all characters controlled by the LOD Trader need to have the same LOD features; you need only maintain different transition sets for each "kind" of character, and then have the LOD Trader control, say, both pedestrians and stationary shopkeepers. In fact, not all "characters" controlled by the LOD Trader need be characters at all: it can heterogeneously manage humans, vehicles, destructible objects, and anything and everything which can benefit from criticality-driven LOD control.

14.7 The LOD Trader in Practice

We've had great results with the LOD Trader. We implemented it in a free-roaming game involving hundreds of characters, having it control eight separate features with hundreds of potential feature solutions. We also implemented conventional distance-based LOD picking, so that we could compare the two. The trickiest part of the LOD Trader implementation process was tuning the criticality metrics and audacity vectors, due to their subjectivity. This is an area where playtester feedback could be extremely helpful.

As expected, distance-based LOD picking was only marginally acceptable. In order to guarantee a reasonable framerate, it was necessary to set the threshold distances so close that things like low-quality locomotion could be clearly seen, particularly in sparsely populated areas of the game world and where there were long, unobstructed lines of sight. The LOD Trader, in contrast, was very effective at maintaining framerate in the most crowded situations, and in sparse areas it simulated most characters at the highest LOD.

A controlled, blinded experimental study verified out this impression: Viewers shown videos of the game running with distance-based LOD picking were consistently more likely to experience BIRs, and to experience them more often, than viewers shown videos of the game running with the LOD Trader [Sunshine-Hill 11].

The LOD Trader itself had very good performance: its average execution time was 57 microseconds per frame, or 0.17% of the target frame time. Its memory usage was 500 kB for the transition data and 48 bytes per entity, both of which could easily be halved by picking narrower datatypes, with no reduction in functionality.

14.8 Conclusions

As mentioned in the introduction, the LOD Trader isn't magical. It can't read the player's mind. Its criticality models are approximate, and often very inaccurate.

But that's okay. The goal of the LOD Trader is not to make wildly audacious detail reductions and get away with them. Rather, the goal is to be clever enough to do detail reduction in the *right places* in those moments when detail reduction has to happen *somewhere*. In those moments, the question is not *whether* to reduce LOD but *how* to reduce LOD without causing glaring problems, and we just can't depend on distance-based LOD picking to do that.

And we *need* to be able to depend on our LOD picker. Because detail reduction is always, *always* going to be needed. We'll never have enough computational resources to do all the things we want to do. And when we're not doing detail reduction at runtime, that just means we're doing it at development time, throwing away good techniques because we don't know if we'll always be able to afford them. That's the real benefit of the LOD Trader: the power to implement AI techniques as lavishly detailed as we can imagine, and others as massively scalable as we can devise, with the confidence that our game will be able to leverage each one when it counts the most.

References

[Sunshine-Hill 11] B. Sunshine-Hill. *Perceptually Driven Simulation* (Doctoral dissertation). Available online (http://repository.upenn.edu/edissertations/435), 2011.