Improving Behavior Efficiency in Virtual Worlds

Jeff White

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-98-10
October 1998

Improving Behavior Efficiency in Virtual Worlds

Jeff White

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the Degree of Master of Science
in the Department of Computer Science at Brown University

October 1998

_____

Professor John Forbes Hughes
Advisor

Abstract

As the complexity of computer games grows, it becomes increasingly necessary to manage the computational cost of both geometry and behavior. Although there have been many advances in software and hardware for efficiently processing geometry, there are relatively few techniques for managing behaviors. In anticipation of the rising cost of behavior execution, we identify techniques for improving behavior efficiency and discuss how these techniques are applicable to specific properties of behavior. We present examples that we have implemented that demonstrate the use and effectiveness of these techniques. We also describe a framework that allows applications to reuse these techniques. Although we focus on game environments, we believe this work is applicable to other virtual environments, such as information visualization, animated films, and virtual walkthroughs.

1 Introduction

Our goal is to create large, scalable games with rich environments and many interesting and compelling characters. We desire content that exceeds the complexity and level of detail provided in current games. To achieve this, we need to use techniques that efficiently process and display a virtual environment to the user.

Two components that influence the runtime performance of virtual environments are geometry and behavior. Geometry defines the visual objects and their properties, such as shape, position, material, and color. An application uses these objects and properties to render and display a scene. Behavior makes the scene dynamic by modifying, creating, or deleting objects. These changes occur in response to events such as the passage of time or user interaction.

As games and other virtual environment applications have become increasingly complex, researchers have developed many techniques for efficiently processing the geometry in a scene. Several algorithms use spatial partitioning to efficiently determine visible sets of geometry [Clark 1976; Fuchs 1980; Teller 1992]. Others use cost/benefit heuristics to perform an "optimal" rendering strategy [Funkhouser 1993]. The advent of consumer 3D graphics hardware has also played a role in accelerating the geometry pipeline. Much of this research has migrated to commercial 3D packages that take advantage of hardware acceleration and support these algorithms [Sowizral 1997; OpenGL 1997; OpenInventor 1994]. These advances have generated rapid improvement in the geometric capabilities of games and as a result, many games are differentiated by their *visual* qualities, such as polygon throughput, texture detail, and other effects.

Similar techniques can be used to manage the complexity of behaviors. For example, consider playing an auto-racing game. The application could stop spinning the wheels on cars that are not visible or are far in the distance. Computational resources are saved by not calculating or updating the rotation of the wheels. A behavior is considered "culled" while it is not updated. The application can do this because the spinning wheel behavior adds a level of detail that is only necessary when you can see it.

As the behavior complexity increases, it becomes both more difficult and more important to simplify the computations. Suppose you were playing a large simulation game like Age of Empires [Microsoft 1998] or SimCity [Maxis 1998], but on a much larger scale – an entire planet. It would be computationally intractable to continually simulate millions or billions of individual people. One solution is to organize behaviors in a hierarchy. For instance, for distant countries the application need only simulate actions that have long range or long term effect, such as political activities. Instead of simulating many individual entities, one entity represents a less detailed, or "degraded," simulation of the entire country. As you move closer to the country, the hierarchy expands to include the simulation of cities, and eventually of individual people as you approach a city. We refer to the levels of the hierarchy as "resolutions" – our imagined game therefore contains multi-resolution simulations. By analogy, a local newspaper presents a multi-resolution view of the world. The newspaper reports many events from the local area, but only the most important ones from the rest of the world. Determining which behaviors in a game require high or low resolution simulation is a complex decision, one that depends on the current game state, the overhead of managing behaviors vs. the savings gained, and other cost/benefit tradeoffs. For instance, suppose in the example above that your friend from far away has agreed to telephone you tonight. You will never hear from him if he is not being simulated. Although he is not visible, his actions directly influence you. The system needs to determine that his simulation is important to you.

We explore these issues and provide several solutions for improving the efficiency of behaviors. While our work focuses on games, he hope our techniques will apply to several types of applications, such as the crowd scenes that are starting to appear in animated films. However, the context of games defines particular constraints. Because games are artistic and entertaining, the constraints are designed to produce enjoyable gameplay. A game should have smooth frame rate, responsive interaction, and convincing animation. It does not, however, always require visual or physical accuracy, which creates opportunity to sacrifice accuracy for efficiency.

1.1 Overview

We have created a set of examples that demonstrate techniques for improving the performance and scalability of an application. We describe these behavior management techniques and draw analogies to the geometry domain.

Along with the examples, we have developed a framework for multi-resolution behaviors. This framework includes reusable components for creating behaviors, specifying behavior properties, and performing level of detail management.

2 Related Work

This section discusses some of the research that is relevant to our work.

2.1 Behavior Analysis and Approximation

Chenney et al. use analysis techniques to control the cost of dynamics in virtual worlds [Chenney 1998]. They present examples of behavior optimization in a virtual fun park with amusement rides. Their system culls the geometry and behavior of a ride when it is out of view. When the user looks back at the ride, the system must determine a plausible state of the ride, which is difficult because the ride's motion is governed by differential equations.

Chenney et al. take advantage of the assumption that viewers cannot accurately predict the state of a system over time. This unpredictability is a result of three factors: a viewer's uncertainty of the last known state of the system, a viewer's uncertainty of the details of the model, and a viewer's lack of knowledge about factors that influence the system. The duration an object is out of view influences the degree of uncertainty, and is classified into three levels: short, medium, and long periods out of view. In a short period of time, users can make accurate predictions, so the behavior must be completely simulated. The dynamics is simulated ahead of the rendering and values are buffered. When the object goes out of view, the geometry and dynamics are culled and the buffer is no longer filled. If the object soon reenters the view, values are read from the buffer. For medium periods out of view, a viewer can accurately predict only a small number of things. Chenney et al. train neural networks to approximate a new state based on the last known state and the elapsed time. For long periods out of view, a viewer cannot predict a new state based on a previous state, though the viewer still has some knowledge about the system's general behavior. Chenney et al. take advantage of this by sampling the new state from a statistical distribution over all states – the *stationary distribution*. This approach is based on the premise that a small amount of uncertainty in a viewer's knowledge of state will grow over time to match the stationary distribution.

These analysis techniques are useful because they automatically generate models for culling. A drawback of this approach is that the system must undergo a pre-processing stage. Additionally, the behaviors must be self-contained systems; they neither react to nor affect other behaviors. Chenney states that their software currently assumes a continuous, low-dimension state space, though he does mention the possibility of adding support for state machines and hierarchies of behaviors.

2.2 Simulation Levels of Detail

Carlson and Hodgins developed a simulation where creatures playing a game interact with each other at different levels of detail [Carlson 1997]. The game consists of a rectangular court in which legged creatures run around and attempt to avoid a puck, the walls, and other creatures. The creatures' dynamics are represented at three levels of detail: a dynamic model, a hybrid kinematic/dynamic model, and a kinematic model. The highest level of detail uses a rigid-body dynamic simulation with three bodies and four controlled degrees of freedom. The middle kinematic/dynamic level of detail uses a point-mass model to determine the location of the body on the plane and a kinematic model to determine the joint angles for the legs given the current running speed. The lowest level of detail uses a point-mass model for the body and holds the leg at a constant angle.

A creature's level of detail is switched based on the importance of the creature's behavior and the creature's visibility. Creatures that are near other entities (creatures, walls, or puck) are simulated at high resolution. Creatures that are in view but far from the viewer are simulated at the medium level of detail. Creatures that are not visible are simulated at the low resolution. Carlson and Hodgins achieve smooth switching between levels by only switching during a particular phase of the running behavior. They claim that this technique should work well for any running or walking system, but may not generalize to more complicated, non-cyclic behaviors.

Carlson and Hodgins plan to implement more complex switching rules based on tradeoffs between frame rates and simulation accuracy. For example, cost/benefit heuristics might be designed to assign a switching

priority for each creature. In their system, the levels of detail were created by hand, but they point out that automatically generating simulation levels of detail will be an important problem.

2.3 Java™ 3D

Sun Microsystems' Java 3D is an example of a commercial 3D graphics package that attempts to optimize behaviors [Sowizral 1997]. Java 3D performs *execution culling* via the `Behavior` node. Each behavior contains a volumetric scheduling region and a wakeup condition. A behavior is executed when it becomes *active* and its wakeup condition has been met. A behavior is considered active only if the viewer's *activation region* intersects the behavior's scheduling region. Java 3D saves computation by only testing wakeup conditions of active behaviors. This scheme allows Java 3D to automatically activate and deactivate behaviors based on viewer proximity.

In later sections, we describe some examples and a framework that use a similar approach for managing behaviors. In addition to proximity, we use visibility to perform behavior culling.

2.4 Plausible Motion Simulation

Visual aesthetics is more important than physical accuracy for many entertainment applications such as games and animated films. Barzel et al. therefore point out that the goal of computer animation is to create "plausible motion" instead of physically accurate motion [Barzel 1996]. Physically based animation often appears sterile because the model is a practical simplification of real-world conditions. Accurately measuring all variables is impossible, and furthermore, simulating them is computationally intractable. Their solution adds variability to the dynamics, while maintaining visual plausibility. Artists can control this "motion texture" to create realistic and visual effects.

This work provides insight into achieving our goal of simulating large interactive worlds. Instead of attempting to model the world accurately, we strive to create a degraded simulation that provides a plausible and convincing experience.

2.5 Incorporating Update Rate in Graphics Systems

Wloka describes a framework for a time-critical graphics system – one that trades quality for efficiency and guarantees certain response times [Wloka 1993]. A time-critical system is necessary for sustaining a user's feeling of immersion in a virtual environment.

Wloka points out that current graphics systems update all objects every frame. Instead, both the programmer and the system should have explicit control over the update rate of objects and their attributes. An update rate slower than the frame rate is useful, for example, if the object is moving slowly, in the background, or visually unimportant. An update rate faster than the frame rate is useful for error-sensitive numerical simulations or for interaction objects (e.g. cursors).

Wloka states that an ideal system would automatically determine update rates such as by updating an object when its screen representation has "noticeably" changed (e.g. pixel comparison). The system would degrade the update rate based on the rate of change of an object, size of an object, object-viewer distance, and possibly other measures of object importance. Wloka states that such a system is currently infeasible and his research therefore initially focuses on manually specifying the update rate. Wloka describes three models for specifying update rate: simple model, jitter-interval model, and weighted jitter-interval model. The simple model allows an application to specify exactly when updates should occur. The jitter-interval model allows an application to specify a time interval during which an update should occur. The weighted-interval model allows an application to specify preferences along the time interval.

Our work is similar to Wloka's in that we attempt to minimize the computation of an object based on perceivable properties of the object, such as visibility, distance, predictability, and subjective importance. Instead of specifying update rates, though, we create behaviors that are culled while out of view and later initialized to a plausible state.

3 Improving Behavior Efficiency

As the complexity of behavior becomes a dominant aspect of computer games, we want to make behavior execution as efficient as possible. Since many techniques have been devised for managing geometric complexity in virtual worlds, we would like to borrow ideas from this work both to help understand the issues and to develop analogous techniques for behavior management. In this section, we examine some techniques for optimizing the execution of behavior. Because our techniques are based on various properties of behaviors, we begin with a brief description of these properties.

To motivate the discussion, we draw examples from a hypothetical bike messenger game in which the player competes to be the fastest messenger in San Francisco. The player must efficiently deliver packages to locations around the city while navigating around various obstacles, such as vehicles and pedestrians.

3.1 Properties of Behaviors

The properties discussed below are divided into two categories: cost and influence. Cost refers to the authoring and runtime resources expended for a behavior. Influence refers to the degree of influence that an object has on the user and is affected by an object's visibility, predictability, and semantic importance.

3.1.1 Cost

Runtime Cost

The desire to reduce the **runtime cost** of behavior is the motivating factor for behavior optimization. Both algorithmic complexity and the runtime environment affect the runtime cost of evaluating a behavior. The author could statically approximate the cost or the application could obtain it dynamically at runtime.

History Dependence

A behavior's **history dependence** affects the authoring cost for the behavior. Behaviors are either instantaneous or cumulative: instantaneous behaviors depend only on the current state of the world, while cumulative behaviors depend on a history of events. A clock is an example of an object with an instantaneous behavior because its state can be computed as a pure function of time. Cumulative behaviors include state-based behaviors which must be updated in order to maintain correct state and integral-based behaviors which require frequent update to produce accurate values. A traffic light can be modeled with a finite state machine (FSM) that reacts to waiting cars as well as the time. If the traffic light behavior is culled for a period of time, it will not be able to give an accurate value when it returns to view. Its value, or state, depends on a history of events, which is captured by the FSM.

3.1.2 Influence

Visibility

The appearance of objects affects the user's perception of the world. Visible objects have a high degree of influence on the user and therefore there is a high benefit to simulating and displaying them. **Visibility** need not be considered an all-or-nothing quality: instead it can be assigned a scalar value based on distance, object size, screen-size, or all of the above. Coarser geometric models can be substituted when visibility is low with the intent of preserving the user's perception of the object while reducing display time.

Visibility can be further qualified by the user's **focus**. A user can discern objects near the center of the view with higher accuracy. Objects that are visible but on the periphery can therefore be degraded [Gossweiler 1996]. For example, when a bike messenger is cruising down the street, the messenger will be

more focused on the cars than on the pedestrians on the sidewalk. Gossweiler performed studies of human perception to determine relative influence of objects in a virtual environment [Gossweiler 1996].

Motion blur also affects the appearance of objects. Objects that are moving quickly with respect to the user appear blurred or can be seen only for a short time and cannot be seen clearly by the user [Funkhouser 1993]. An example of motion blur is the spinning wheels on a moving car. Because the wheels are moving so fast, we can degrade both the geometry and the behavior.

Objects have relationships that determine how one object affects another object. Data structures can be used to model these relationships and provide efficient data pruning. Occlusion and containment are relationships that are used to accelerate visibility determination.

Containment can also be applied to *behavior*. In particular, the effect of a behavior or group of behaviors may be **spatially bounded**, i.e. have a finite spatial extent delimited by spatial bounds. For instance, throughout our game, the people and elevators inside an office building will remain inside the building. The behaviors of these people and elevators do not affect any behaviors outside the building. Since no behaviors outside the building are dependent on them, the system can cull the contained behaviors along with the geometry when the building is out of view. If the messenger's dispatcher is inside a building, however, his behavior extends outside the building because he can call the messenger with his cell phone. The dispatcher's behavior, therefore, is not spatially bounded by the building.

Another example is the rotation of car wheels. The rotation of a car wheel is spatially bounded by the geometry of the wheel, allowing us to cull the behavior when the car is distant or out of view. The motion of the car, however, is not spatially bounded because the car could travel almost anywhere within the city. Many behaviors that only add visual realism to an environment are spatially bounded.

Predictability

A user may be able to perceive certain characteristics of a behavior. A behavior's **predictability** refers to the ability of the user, not the computer, to predict state, and is measured on a continuous scale. The motion of a trolley is very predictable to the player since it often travels at constant velocity on a fixed track. On the other hand, the motion of a pedestrian or a taxi is very unpredictable. A computer simulation can accurately predict future state since it has knowledge of all the variables and equations, but a human cannot predict this chaotic behavior. If a behavior has low predictability, the user will presumably not notice if it is simulated with decreased accuracy.

A behavior may also appear **cyclic**, repeating its output over a constant period of time. An animated billboard, for instance, has cyclic behavior. Behaviors with complex output or long periods may technically be cyclic, but may not be recognizable as such by the user. Conversely, mathematically chaotic behaviors may appear cyclic to a human. Cyclic behaviors are often predictable.

Importance

Another property that affects the degree of influence is **importance**. The importance is a subjective and context-dependent quality that an author can specify. Importance can be attributed to either the existence of the object (will it matter if it is culled?) or the accuracy of the object (will it matter if I use fewer polygons or a simpler algorithm?) or both. In the messenger game, the vehicles and pedestrians have a large importance, while other objects such as birds or animated window displays have small importance. Funkhouser demonstrated an adaptive display algorithm that uses a cost/benefit heuristic for maximizing scene quality [Funkhouser 1993]. The heuristic takes into account visibility, motion-blur, focus, and importance.

3.2 Techniques for Improving Behavior Efficiency

We now illustrate techniques for improving behavior efficiency by describing the construction of the bike messenger game. Each of the following sections adds more elements to the game and discusses techniques for optimizing the behavior.

3.2.1 Culling Static Geometry

We begin by creating the scenery – roads, buildings, and the sky. These static elements constitute most of the geometric complexity of the game, and need to be efficiently managed. We can think of the rendering of geometry as a behavior whose spatial extent is simply the geometry that is displayed. Because the geometry is static, we can cull the geometry and then later display it knowing that it has not changed. This permits systems to cull geometry based on visibility. A simple technique for determining visibility is view-frustum culling – removing objects outside the user's field of view. Additionally, the relationships of occlusion and containment can be used to efficiently determine visible sets of geometry. Binary space-partitioning (BSP) trees [Fuchs 1980] and cell/portal graphs [Teller 1992] can be used for occlusion-based visibility determination, and bounding-volume hierarchy (BVH) trees [Clark 1976] can be used to define hierarchical containment relationships.

3.2.2 Culling and Reinitializing Spatially Bounded Behaviors

Suppose that we now want to animate the scene with a simple behavior – we attach a working clock to one of the buildings. This animation requires extra computation, so we would like to minimize its execution cost. Can we cull the behavior when it is not visible? What does it mean for a behavior to be visible?

Because the clock animation exists only to provide visual detail to the player, the behavior has a well-defined spatial extent. The animation affects only the geometry of the clock, and furthermore, no other objects in the scene are dependent on the value of the clock. Therefore, the clock behavior is spatially bounded by the clock geometry. A behavior can be culled when its spatial extent is not visible by the player since the player cannot witness any effects of the behavior. Culling spatially bounded behaviors is analogous to using geometric containment to cull objects or groups of objects whose bounding boxes are not visible. We cannot, however, simply suspend the computation of the behavior. If the clock is stopped while it is out of view, it will show the incorrect time when the user sees it later. Fortunately, this behavior is instantaneous; the clock can be modeled purely as a function of time. We can therefore simply restart the clock animation with the current time when the clock becomes visible.

Suppose we now add some more dynamic elements, such as the trolleys. We model the overall position of the cable car with an FSM that responds to the loading and unloading of passengers as well as the traffic conditions. We model the rotation of the trolley wheels as an integral of the car's velocity. Since these behaviors are cumulative, we cannot trivially initialize them after they have been culled. One option is to evaluate the behavior from the moment it was culled to the present time. If the behavior is reactive to external events, the system must queue these events. The main problem, however, is that this approach entails a reinitialization lag, which results in no overall computational savings. An alternative approach is to approximate an initial state of the behavior.

Chenney et al. demonstrated two techniques for reinitializing cumulative behaviors, both of which generate a plausible state based on a pre-processing analysis [Chenney 1998]. Neural nets were trained to produce an approximate state given the last known state and duration out of view. Over longer periods of time, a stationary distribution over all the states was used to reinitialize behaviors. Note that these techniques apply only if a system is non-reactive and spatially bounded.

After a long time of not seeing a particular trolley, the player will not be able to accurately predict the location of the trolley. Additionally, the locations of the trolleys over time are not that important to the overall game, so it is reasonable to approximate the initial state of the trolley. Suppose that instead of trying to approximate the motion of the wheels, we decide to reinitialize the rotation to an arbitrary state. Since the wheels are spinning rapidly, the user will not be able to predict an accurate state of the wheel. Furthermore, since their importance is low, the user will neither care nor notice if the wheels are initialized to an inaccurate state. The low predictability and importance indicate that the accuracy of the behavior can be degraded.

A behavior's spatial bounds is similar to Java 3D's `Bounds` object [Sowizral 1997]. Lights, sounds, and behaviors in Java 3D can be activated/deactivated as the user enters/exits a bounding volume. In contrast, we use visibility to determine when to cull a behavior.

3.2.3 Level of Detail Behavior

Suppose we now add crowds of pedestrians, which provide additional detail to the city as well as being obstacles for the messenger. The pedestrians need to avoid collisions with each other and with the cars. The addition of the human models and their behavior adds more complexity. Often during the game, many people will be visible, so we cannot cull them. The people in the distance, however, take up little screen space, perhaps only a few pixels. A distant person's visibility is low, which lowers its perceptual influence on the user. The player will likely be focused on the near pedestrians and not the pedestrians further ahead. Thus, we can use geometric level of detail (LOD) to substitute alternative polygonal models of an object [Clark 1976; Krus 1998]. The renderer chooses which model to display based on the object's screen-size.

We can apply an analogous technique to the execution of behaviors. Carlson and Hodgins showed that creatures could be simulated with multiple levels of detail [Carlson 1997]. The dynamics of the creatures had three different representations, which were switched based on the viewer location and the game action. We can use a similar approach for the pathing behavior of the pedestrians. Since the distant pedestrians have low visibility, we can use less expensive pathing and collision detection algorithms on them. For instance, we can use polygonal-based collision detection for near people and simple bounding-box tests for distant people and the user will presumably be unable to discern the reduced accuracy of the distant behaviors. Hubbard demonstrated a more sophisticated collision detection system that uses sphere trees to trade accuracy for speed [Hubbard 1995]. We can also allow collisions that are unnoticeable, permitting distant people to walk through each other.

Geometric LOD entails substitution of a coarser geometry model that resembles the fully detailed model. The analogy for behavior is to substitute less expensive algorithms that produce similar but less accurate output.

3.2.4 Aggregate Groups of Behavior

We now add some more obstacles, such as motorcycle gangs that taunt the messenger. Although the techniques used so far have reduced the computational load, the geometry and behavior of these additional entities add a substantial load.

Often, a motorcycle gang drives as a pack, without much interesting individual behavior. When the pack is distant or out of view, we can model the pack as one entity. The "pack" object can perform a degraded simulation in which all bikes are represented by a single object that determines the overall position of the pack and is responsible for generating and receiving any events for the individual bikers. For example, if the pack drives over a patch of broken glass, the pack object needs to approximate the effect of that event, deciding which of the bikes got flat tires and need to pull over and leave the pack. The pack disaggregates those bikes and reinitializes their behaviors. The same disaggregation occurs for all bikes when the pack becomes visible to the player.

3.2.5 Factoring Out Simpler Behaviors From More Complex Behaviors

We add cars which need to make decisions such as what roads to take and when to pass. Is there any way to simplify this behavior? The motion of the cars is neither deterministic nor spatially bounded. The action of the computer cars affect each other and in turn the player.

Let us examine more details of the behavior with the intent of breaking it into simpler parts. Consider a car that is travelling down a street towards an intersection. The car is out of the player's view and there are no other cars in the vicinity. As it nears the intersection, the car will decide which road to take. Suppose the car is going to continue moving towards the intersection at a constant speed. Until the car reaches the intersection, the behavior is both instantaneous and spatially bounded. We can take this phase of the behavior and apply the technique of culling spatially bounded behaviors. The behavior can calculate the time to the next event based on the distance and speed, sleep until the time expires, and then teleport to the destination. If the car comes into view or another vehicle approaches, it can accurately initialize its position and velocity based on the current time.

The technique of breaking a complex behavior into simpler components may make it possible to apply other techniques to the sub-behaviors. Any complex behaviors created from a sequence of primitive behaviors can benefit from the same treatment.

3.3 Discussion

Many of the techniques and properties described above arose from abstracting concepts from the examples that we implemented. We envision a behavior management system that would use the techniques described above in conjunction with all of the properties to determine an "optimal" behavior execution strategy. For example, the system would consider a behavior's predictability over time and degree of influence to determine how accurately to compute the behavior. The system would automatically perform simple culling of instantaneous behaviors whose spatial extent is not visible to the player. The system would use geometric visibility information (e.g. view-frustum, occlusion, or containment) to determine if a behavior's spatial extent is visible to the player. The system would use visibility and proximity information to determine an appropriate level of detail for a behavior. There are drawbacks to such a design, though; the time spent determining what work to do may exceed the effort saved. For example, determining visibility of a flashing neon sign over time may require more computation than simply executing the animation all the time. A large-scale system must be able to make predictions about its own performance, factoring this into the overall runtime cost of behaviors.

We have included several of the techniques and properties discussed above in our framework so that authors may construct behaviors that are automatically managed by the system. The next section describes how we used these techniques and properties in our examples.

4 Examples

This section describes several examples of behavior optimization that we have implemented. The examples are based on behaviors that a real game might include. We believe that a typical 3D game environment requires smooth frame rate, responsive interaction, convincing animation, and physical plausibility (though not necessarily physical accuracy).

The main purpose of games is to entertain and entice. With this in mind, our first goal was to design examples of *active* behaviors. Many current games contain only *reactive* entities that are effectively asleep until the player's presence wakens them. Our more independent behaviors are intended to evoke a sense of continual action of all the entities throughout the game.

A second goal in implementing the examples was to increase the ratio of behavior load to geometry load, imitating what we believe will be the future context of game development. This goal is important for demonstrating our results, for if an application spends 90% of the time rendering, then behavior optimizations will have negligible effect. An ideal environment therefore required many active entities and a small amount of displayable geometry.

Our solution was to use a 3D first-person dungeon environment. The dungeon consists of walls, doors, and monsters. A portal graph [Teller 1992] was used for efficient visibility determination, and also proved useful for optimizing the behaviors. To facilitate path planning, we constructed a "pathing" graph from the portal graph where each node corresponds to a waypoint[1]. By traversing the pathing graph, monsters can explore the entire dungeon while avoiding collisions with walls.

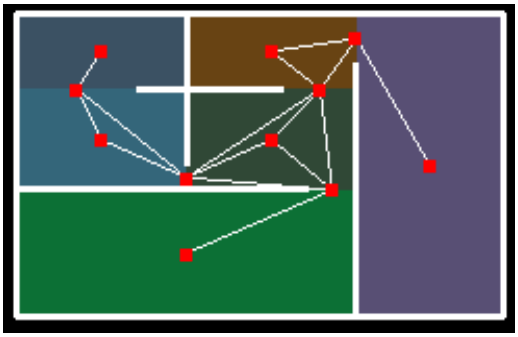

Figure 1: Cells and pathing graph. The thick white lines represent wall segments; each colored rectangle represents a cell. The pathing graph consists of the red waypoints connected by thin white lines.

To visualize the game action we ran a server on a remote machine. The server displays an overhead view of the entire dungeon, showing the state of all the entities, including the player. The game regularly transmits new game state to this overhead display.

4.1 A Searching and Hopping Example

In this example a large number of monsters are actively searching for the player. The searching algorithm is to follow an edge of the pathing graph until the next waypoint is reached. The monster then chooses a random waypoint from among the connected edges and travels towards the new destination. To add visual detail, each monster also hops up and down as it travels forward. The hopping behavior is implemented as an instantaneous function that computes projectile motion based on the modulus of the current time.

[1] A waypoint is a "well-known" location used for path planning. To calculate a path, an entity plans a path to its nearest waypoint, traverses the pathing graph to the waypoint nearest the destination, and plans the remaining path to the destination.

The game starts with the monsters evenly distributed throughout the dungeon. The user can move anywhere about the dungeon and observe the monsters. Because this is just a test scenario, nothing happens when a monster encounters the player.

We implemented three different approaches for degrading the behaviors. The first approach uses distance to determine level of detail. The second approach uses visibility to determine level of detail. The third approach uses both visibility and distance to determine level of detail and update rate.

4.1.1 First Iteration: Distance-based LOD

We first created two levels of detail for the searching behavior. The low detail behavior is used for monsters whose distance to the player is greater than a predefined amount, the *distance threshold* (30 meters in our case). Each monster has an LOD manager that periodically computes the distance to the player. The manager switches behaviors when the distance crosses the threshold. Because both player speed and monster speed are bounded, we can ensure that the switch occurs when the player and monster are still fairly far apart.

At high resolution the monsters are updated each frame by the searching and hopping behaviors. At low resolution the searching behavior calculates the arrival time to the destination based on the distance to the waypoint and the monster's velocity. Instead of updating the monster each frame, the behavior schedules itself to be notified when the arrival time occurs. The behavior then sleeps until it receives the time event and then "teleports" the monster to the destination. The hopping behavior was optimized by stopping it entirely at low resolution.

The overhead view shows which monsters are searching at high resolution and which monsters are searching at low resolution. Most of the monsters are far from the player, and are therefore being simulated at low resolution. They are updated only about every 20 seconds – corresponding to the average distance between waypoints and the velocities of the monsters.

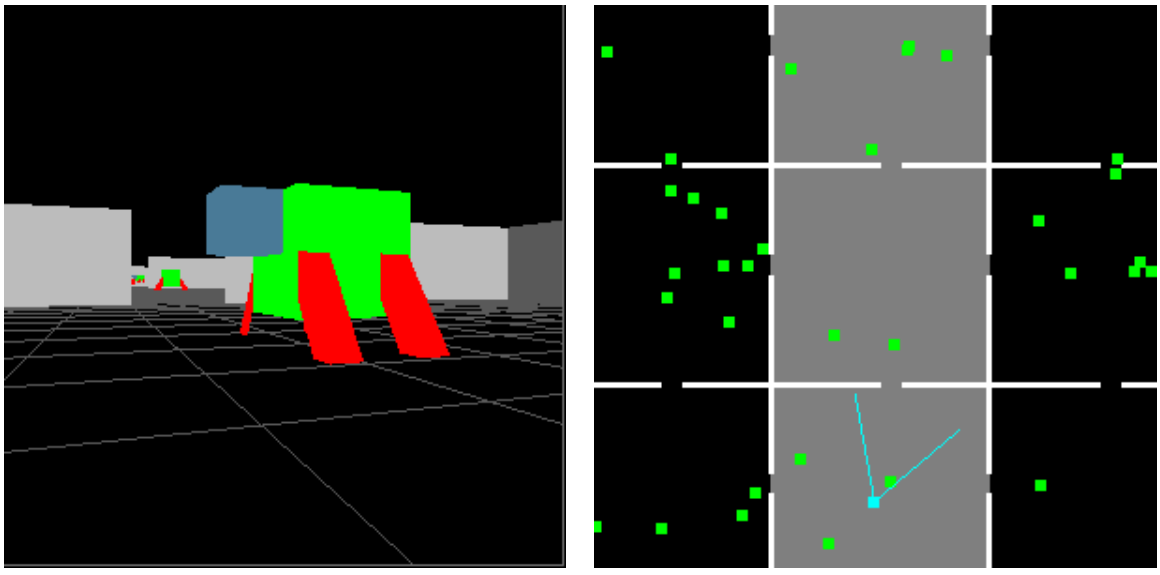

Figure 2: Player view and overhead view. The left image shows the player's view of the dungeon and monsters. The right image shows the overhead view. The cyan dot and lines indicate the player and field of view. The gray areas indicate the visible cells.

This example demonstrates good performance improvement, but there is a problem: Because the player can, by looking through a large sequence of doors, see further than the distance threshold, the player can see monsters that appear frozen. Distance may be a good visibility metric for some environments such as

densely occluded dungeons or foggy areas; however in this example, distance is not a good measure. Increasing the distance threshold, on the other hand, would cause the majority of the monsters to be simulated at high resolution, counteracting the benefits of the behavior detail control.

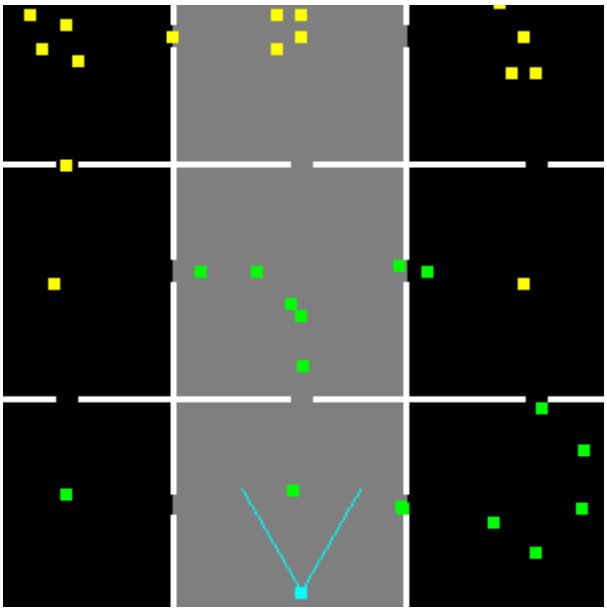

Figure 3: Distance problem. The green monsters are at high resolution and the yellow monsters are at low resolution. The player is able to see "frozen" low resolution monsters.

4.1.2 Second Iteration: Visibility-based LOD

The second iteration addresses the distance problem by using visibility to switch levels of detail. The portal graph conveniently allows us to reuse the visibility information computed by the renderer. The renderer stores a list of the cells visible by the player, and each entity tracks which cell it occupies. Instead of computing distance, each monster's LOD manager compares the monster's cell to the list of visible cells. This method achieves both more accuracy and efficiency than the distance-based approach. It guarantees that the player will not see any idle monsters, and often yields increased culling, such as when many monsters are near but not visible. This method also has a drawback – visibility can change instantaneously, requiring a visibility check at every frame. The next iteration addresses this problem by using both visibility and distance to manage level of detail.

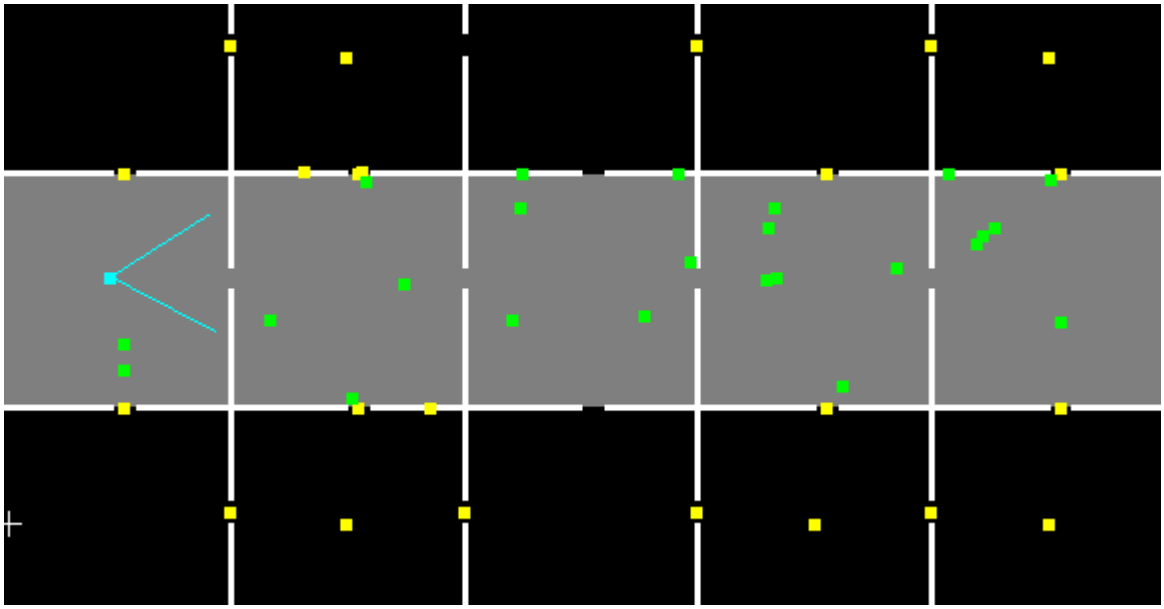

Figure 4: Visibility switching. All visible monsters are simulated at high resolution. All non-visible monsters are simulated at low resolution.

4.1.3 Third Iteration: Three Levels of Detail

The next step was to increase the scale of the simulation to a much larger world that contains 1000 monsters. To manage this, we added a third level of detail for monsters that are both not visible and further away than a second distance threshold. Instead of teleporting from waypoint to waypoint along the pathing graph, the distant monsters sleep for a period of time proportional to the distance to the player. The monster then computes how far it would travel and teleports to a random location within that distance.

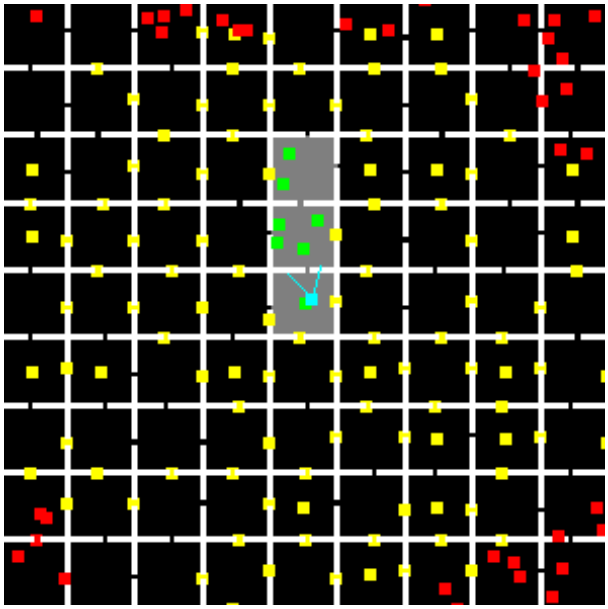

Figure 5: Three levels of detail. The visible monsters in green are simulated at high resolution. The non-visible but near monsters in yellow are simulated at medium resolution. The non-visible and far monsters in red are simulated at low resolution.

4.1.4 Results and Discussion

For the first two iterations, simulating all 100 monsters at high resolution achieves about 40 frames per second (fps).[2] The distance-based LOD increases the frame rate to approximately 60 fps, a 50% increase. The visibility-based LOD further increases the frame rate to about 80 fps, two times faster than with no behavior management. In the third iteration, simulating all 1000 monsters at high resolution produces a frame rate of only 5 fps. Using three levels of detail increases the frame rate to 60 fps, a 1200% increase.

Decreasing the amount of computation performed as entities get further away from the user allows the designer to enlarge the application. For example, if each doubling of entities only adds a constant computational load, then we can greatly expand the scale of the application. Consider a game containing 10 monsters in which each monster uses 10 ms of computation per second. The total work done for all monsters is 100 ms per second. By using a lower level of detail, we can add 10 more monsters, each of which only uses 5 ms of computation per second, bringing the total work to 150 ms instead of 200 ms. We could add another 20 monsters, each of which uses 2.5 ms per second, bringing the total work to 200 ms. Continuing in this manner, we see that the computational load increases linearly as the number of entities increases exponentially. Our examples show that this is a plausible goal, at least over a limited scale.

4.2 A Looting Example

Our next example expands on the searching example. The monsters now have a goal: search a dungeon for treasure and escape with the loot.

Each monster travels along the pathing graph as before, but with some added capabilities and constraints. Each time a monster arrives at a waypoint it checks the current cell for any treasure. If it detects any items it hops to the nearest item and grabs it. The monster continues to pick up any other items in the cell before returning to the pathing graph. The monsters also remember the last ten waypoints visited, which helps them explore new territory. In addition, a “commander” has instructed the monsters to stay within a particular region of the dungeon. There are four such zones and the commander has evenly divided the monsters among them.

[2] The results are obtained from a Sun 200Mhz Ultra 1 with 128M RAM and Creator3D graphics hardware.

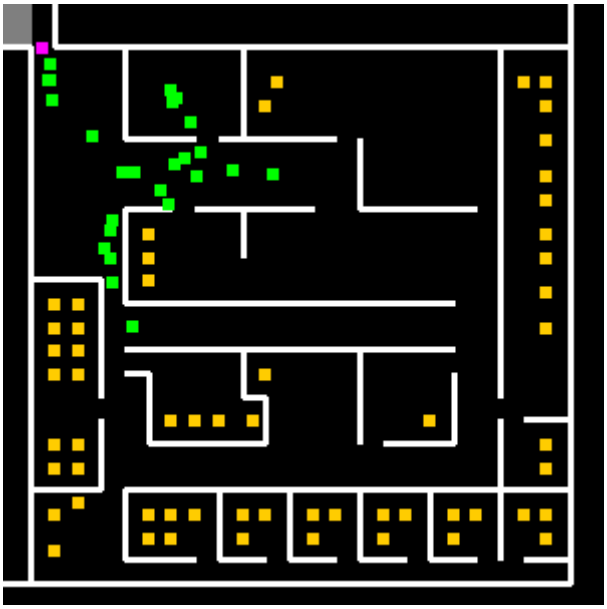

Figure 6: A zone. The monsters have just started looting the zone for treasure (orange dots). The entry door to the zone is represented by the magenta dot at the top left of the image.

This example serves two purposes: to demonstrate the use of spatially bounded behaviors and aggregate behaviors. The dungeon is divided into four zones, each of which acts as a behavior boundary: no events can enter or exit the bounds of the zone. Each monster is contained in its zone and the player cannot affect any behaviors from outside the zone. Consequently, we can cull all the behaviors inside a zone while the player is outside the zone.

Each zone aggregates the behavior of the monsters. When the player enters the zone, the zone reinitializes the monsters to a plausible state and starts their behaviors. The zone must estimate the group behavior based on the elapsed time and the previous state of the monsters. This requires a model of the cumulative behavior of all the monsters. The locations of the monsters and the specific treasure that is acquired over time are not deterministic. We therefore devised a heuristic to approximate the collective behavior of the monsters. All the monsters in a zone start near each other. Starting from a node near this location, we perform a breadth-first traversal of the pathing graph. The number of nodes traversed is proportional to how long the zone was asleep. At each node, any nearby treasure is grabbed. When the traversal is complete, the zone distributes the monsters among the leaves of the traversal. This approximation spreads the monsters out from their starting locations, finding all treasure along the way. In practice, this appeared to be a good approximation of the full looting simulation.

This technique works well the first time the player enters the zone. A different technique is needed to initialize state when the player enters subsequent times. The monsters are then scattered and a breadth-first traversal from any single node would be a poor approximation. Since the monsters are scattered, we just advance each monster along the pathing graph, grabbing any treasure along the way.

The zone can perform an additional optimization: the author has specified that the monsters will complete their task in an approximate amount of time, the *task duration*. If the player enters the zone after the task duration, the zone simply removes all the monsters and loot.

4.2.1 Results and Discussion

To demonstrate the effectiveness of this example, we ran the example with and without the behavior management. Without the behavior management, all of the monsters are actively looting the dungeon and we achieve about 45 fps. With the behavior management, we are able to maintain about 70 fps.

Each zone aggregated the collective behavior of its monsters by initializing the monsters to a plausible state, and acted as a behavior boundary between the monsters and the player. Although both the aggregation and the spatial bounds were "hand-coded," understanding the concepts was helpful in implementing this example. A future system may be able to automatically generate aggregated behaviors and determine spatial bounds of behaviors.

Our measure of success does not take into account the user's perception of the simulation. Although we are able to maintain an acceptable frame rate, we have not attempted to measure the accuracy of the degraded simulation. Through informal observation, the behaviors appeared to be consistent and plausible, though user studies would provide a better measure of success.

5 Framework

We have developed a framework in Java that supports behavior authoring and behavior management. The framework is not tied to a particular graphics system; an application that controls the main loop can plug-in the framework. We chose Magician (a Java/OpenGL wrapper) [Descartes 1998] as the rendering library because of OpenGL's good immediate mode support. We provide a brief description of some of the components of this framework.

5.1 Creating Behaviors

The mechanism for creating behaviors is flexible. The framework provides a low-level event model and a hierarchical FSM (HFSM) library built on top of it. Authors can implement behaviors using just the event model, or also in combination with the HFSM library. Behaviors are not required to subclass any specific class, though behaviors that wish to support automatic culling and level of detail switching must implement the `BehaviorProperties` interface.

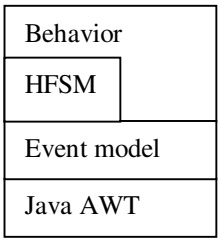

Figure 7: Behavior creation components. The event model abstracts Java AWT events. The HFSM library lies on top of the event model. Behaviors are created using the event model and HFSM library.

5.1.1 Event Model

The event model is the foundation of the behavior framework. The event model is a thin layer that provides an efficient and consistent interface to both system (Java AWT) and application-defined events. Based on discrete event simulation principles, the event model provides a central priority queue for reading and writing events. The event model's `AppQueue` is responsible for dispatching events in order by event timestamp. A subinterface of Java's `EventListener` interface allows an object to register interest in a future event, which will cause the object to be notified when the event fires. The event model wraps all Java AWT events to fit into this model, and includes support for other frequently-used events such as elapsed time, elapsed frames, and application-defined events.

5.1.2 HFSM Library

On top of the event model lies the HFSM library. The HFSM library is a flexible and extensible library that can be used to create complex behaviors. This library is based on the description of state diagrams for dynamic modeling given by Rumbaugh et al. [Rumbaugh 1991], and supports states, transitions, guards, FSMs, composite states, activities, and actions upon transition, state entry, and state exit. State-based behaviors can choose to use this library or implement their own state management.

5.2 Managing Behaviors

The framework for managing behaviors includes components for sensing proximity and distance to the player, an interface for specifying behavior properties, and classes that implement specific behavior optimization techniques.

5.2.1 Sensors

Sensors are used to detect proximity and visibility of entities. The abstract `VisibilitySensor` class contains an abstract method for sensing visibility. Subclasses provide the implementation for detecting visibility. This allows implementations to be based on the environment. Our examples use a `PortalGraphVisibilitySensor` to determine visibility of entities in a portal graph. The implementation simply queries the `PortalGraphRenderer` for the list of visible cells and it then checks if the entity occupies one of the cells. The `ProximitySensor` measures the squared distance between any two entities.

The `VisProxManager` manages efficient retrieval and dissemination of visibility and proximity data. It uses a small state machine that controls when visibility and proximity information is computed. The state machine has two top-level states: *visible* and *not visible*. The *not visible* state has a parameterized list of states, which correspond to increasing distance thresholds. When the entity is not visible, the `VisProxManager` queries distance and visibility at a frequency proportional to how far the entity is from the player. When the entity is not visible but near, the `VisProxManager` queries the `VisibilitySensor` each frame. There is a chance that the entity will become visible any frame. When the entity is visible, the `VisProxManager` only needs to check for visibility, but not every frame. It is acceptable if the behavior continues to run for a short duration after the entity is no longer visible. This also sustains hysteresis so that the behaviors do not oscillate quickly between two levels of detail. The `VisProxManager` is constructed with an instance of `VisibilitySensor` and `ProximitySensor`. The `VisProxManager` fires a `VisProxStateChangeEvent` when the state of the visibility or proximity changes. The LODManager listens to the `VisProxStateChangeEvents` and broadcasts them to its list of BehaviorManagers.

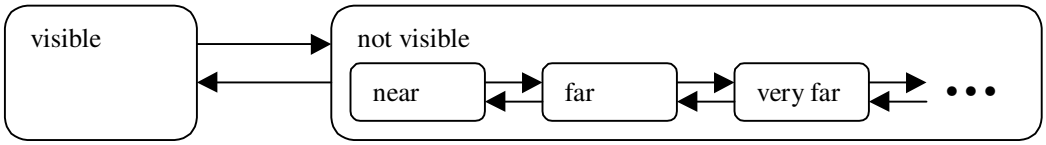

Figure 8: VisProxManager finite state machine. visible *and* not visible *are the two top-level states. The* not visible *state is composed of a list of states representing distance to the viewer.*

5.2.2 BehaviorProperties, BehaviorManager

Behaviors that should be automatically managed must implement the `BehaviorProperties` interface. This interface contains methods for querying properties of the behavior, starting and stopping the behavior, and for setting the level of detail of the behavior. This interface allows the system to determine if a behavior is instantaneous, cumulative, cyclic, etc.

`BehaviorManager` is an abstract class that controls LOD switching and culling for a single behavior instance. The `BehaviorManager` uses the visibility and proximity information provided by the LODManager to manage the behavior. There are a few concrete subclasses that implement the techniques discussed in Section 3, based on the behavior's properties. A `BoundedBehaviorManager` is used for behaviors that have known spatial bounds. The manager starts and stops the behavior as it becomes visible and not visible. An `LODBehaviorManager` sets the level of detail on its behavior based on the visibility and proximity information provided by the `LODManager`.

We anticipate creating more BehaviorManagers that use other properties such as predictability and importance.

5.2.3 GroupManager

This part of the framework provides support for aggregation and disaggregation. A `GroupManager` managed a group of entities. The `GroupManager` class defines two abstract methods: `collapse()` and `expand(long dt)`. The `dt` argument specifies how long the group has been culled and is managed by the base class. Subclasses implement these methods to define the aggregation and disaggregation of behaviors.

The `GroupManager` is parameterized with the events that cause it to collapse and expand. For example, a flock of birds may be expanded/collapsed based on proximity to the user. In the looting example, a door's open event causes a zone to expand. The zone collapses when the player exits the bounds of the zone.

6 Conclusions and Future Work

We have described techniques for optimizing behavior execution in virtual worlds. These techniques are based on properties of objects that relate to the user's perception, the application content, or the execution environment. We demonstrated the use and effectiveness of these techniques and properties with examples from a game. Finally, we have implemented a framework for reusing these techniques in other virtual environments. Our work has generated many ideas for further research. This section presents some of the research we are planning.

Our research group intends to write more examples of multi-resolution behavior. In particular, we would like to develop methods for managing large-scale hierarchies of geometry and behavior. We are also exploring techniques for simulating dense behavior, such as ballroom dancers. Occlusion culling is not as effective in this type of scene and there is a higher geometry load. Consequently, we need to use more sophisticated techniques for controlling level of detail based on the viewer's location and orientation. We are also coordinating the dancers' behavior level of detail with the geometry level of detail. Another technique for multi-resolution behavior we are exploring is filtering of motion-capture or key-frame data. This technique will be used to simplify generic character animations, such as walking, running, or dancing.

One area we did not fully discuss is how a behavior system would determine the spatial bounds of a behavior. In our examples, the spatial bounds were implicitly coded into the behaviors: the searching and hopping behaviors were bound by each monster, and the groups of looting monsters were bound by the zones. It may be possible for a system to automatically determine spatial bounds of behaviors. This could be implicit, such as by using an expression graph that describes relationships between behaviors, or explicit, as in Java 3D's scheduling regions.

We plan on expanding the framework and adding automatic support for more of the behavior properties, such as predictability and importance. We would like the system to dynamically scale behaviors based on the runtime performance. The system can add as much detail as the computer can handle. This eliminates the need for content creators to target specific runtime environments. All users will experience the content at the fullest possible level [Lander 1998].

Multi-player environments bring up several interesting issues. For example, the system needs to consider all viewers when managing behaviors. A simple solution may be to manage behaviors based on a "common denominator" of all the viewers; all the clients receive the same data. Alternatively, the server could manage the behaviors independently for each viewer. A third option is to have each client responsible for its own behavior management. These last two options bring up consistency issues. Two players could witness different resolutions of the same behavior, which may cause an "unfair" inconsistency between them.

As new standards for authoring and executing behaviors arise, and as the level of behavior complexity rises, the need for behavior optimization techniques will increase. This technology will enable content creators to create large-scale games with unprecedented gameplay.

Acknowledgements

I thank my advisors John Hughes and Andy van Dam for all their guidance and support. I thank the following people for helping me with various aspects of this project: Jen Stewart, Rosemary Simpson, Steve Dollins, Dom Bhuphaibool, Matthias Wloka, Nancy Pollard, Greg Seidman, Dave Jackson, and Dan Goldstein. I also thank all the other members of the Brown Graphics Group for interesting discussions and inspiration.

References

[Barzel 1996] Ronen Barzel, John Hughes, and Daniel Wood. “Plausible Motion Simulation for Computer Graphics Animation.” In *Computer Animation and Simulation ’96* (Proceedings of the Eurographics Workshop). 1996.

[Carlson 1997] Deborah Carlson and Jessica Hodgins. “Simulation Levels of Detail for Real-time Animation.” In *Graphics Interface ’97, Proceedings.* 1997.

[Chenney 1998] Stephen Chenney, Jeffrey Ichnowski, and David Forsyth. “Efficient Dynamics Modeling for VRML and Java.” In *Proceedings of VRML 98*. 1998.

[Clark 1976] James Clark. “Hierarchical Geometric Models for Visible Surface Algorithms.” In *Communications of the ACM*. Volume 19, Number 10, October 1976.

[Descartes 1998] Alligator Descartes. “The Magician OpenGL Interface.” `<http://www.arcana.co.uk/products/magician/>`. 1998.

[Fuchs 1980] Henry Fuchs, Zvi Kedem, and Bruce Naylor. “On Visible Surface Generation by A Priori Tree Structures.” In *Computer Graphics* (SIGGRAPH ’80 Proceedings). 1980.

[Funkhouser 1993] Thomas Funkhouser and Carlo Séquin. “Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments.” In *Computer Graphics* (SIGGRAPH ‘93 Proceedings). 1993.

[Gossweiler 1996] Richard Gossweiler. “Perception-Based Time Critical Rendering.” Ph.D. thesis, University of Virginia. 1996.

[Hubbard 1995] Philip Hubbard. *Collision Detection for Interactive Graphics Applications*. Ph.D. thesis, Brown University. 1995.

[Krus 1998] Mike Krus. “Levels of Detail.” `<http://www.limisi.fr/Individu/krus/CG/LODS>`. 1998.

[Lander 1998] Jeff Lander. “Looking Forward with a Backward Glance at the CGDC.” In *Game Developer.* Volume 5, Number 8, August 1998.

[Maxis 1998] Maxis. “SimCity.” `<http://www.maxis.com/>`. 1998.

[Microsoft 1998] Microsoft. “Age of Empires.” `<http://www.microsoft.com/games/empires/>`. 1998.

[OpenGL 1997] OpenGL Architecture Review Board. *OpenGL Reference Manual.* Reading: Addison-Wesley. 1997.

[OpenInventor 1994] Open Inventor Architecture Group. *Open Inventor™ C++ Reference Manual.* Reading: Addison-Wesley. 1994.

[Rumbaugh 1991] James Rumbaugh et al. *Object-Oriented Modeling and Design.* Englewood Cliffs: Prentice Hall. 1991.

[Sowizral 1997] Henry Sowizral, Kevin Rushforth, and Michael Deering. *The Java™ 3D API Specification*. Reading: Addison Wesley. 1997.

[Teller 1992] Seth Teller. “Visibility Computations in Densely Occluded Polyhedral Environments.” Ph.D. thesis, University of California at Berkeley. 1992.

[Wloka 1993] Matthias Wloka. “Incorporating Update Rates into Today’s Graphics Systems.” Brown Computer Science Technical Report. 1993.