4.3

Sphere Ttees for Fast Visibility Culling, Ray Tracing, and Range Searching

John W. Ratcliff, Sony Online Entertainment

jratcliff@verant.com

*JL*Jfcile there are many data structures for storing static 3D objects, including Wlf quadtrees, octrees, and BSP trees, they are not always ideal for large numbers of dynamic objects. This gem presents an algorithm and demonstration application that manages thousands of objects in motion that are continuously maintained as a collection of hierarchical bounding spheres in a *SphereTree.*

The design goal for this algorithm has been to make the 99-percentile case spend almost no CPU time updating an object in motion within the tree structure. Queries against the SphereTree perform more tests than other data structures, but this is mitigated by the fact that the tree can be maintained using very little CPU time. This data structure is ideally suited for gross culling of massive numbers of moving objects in a large world space. It doesn't matter if the objects are moving at widely disparate speeds, or even if many of them are not in motion at all. It also has a very low cost when objects are inserted and removed from the tree with great frequency.

Bounding Spheres

There are certain limitations when using a bounding sphere as a culling primitive. A bounding sphere does not necessarily fit very tightly around an object, especially if it is tall and thin. On the other hand, this over-described bounding sphere can be seen as a feature, not necessarily a limitation. A bounding sphere must encompass the complete extent of an object in all orientations. This includes taking into consideration all possible animation poses that might be applied. Additionally, this bounding sphere is presumed to encompass all child objects that are attached to the parent. This allows for the assumption that whatever the visibility state of the parent is also considered true for its children. Another advantage of an over-described bounding sphere is that it can be used to cull animations, shadows, and other associated effects. This extra slop around an object can be an effective tool to determine when to treat an

object, its children, and its associated effects as active or inactive. Culling shadows, animations, and special effects are just as critical as culling geometry alone.

Using Sphere Trees

Every object in the simulation, whether it is in motion or not, uses the class SpherePack to maintain itself inside a valid SphereTree. When an object changes position using the method NewPos(), SpherePack simply computes the squared distance between the new position and the center of the parent node. If it is still contained within the radius of the parent sphere, which is designed to be true almost all of the time, the routine immediately returns. This method is implemented inline for maximum performance. This is the only calculation performed for the vast majority of all objects in motion, even if there are thousands of them. For static objects, nothing is done beyond their initial insertion into the tree.

When a new position would cause an object to pierce the skin of its parent sphere, then that child is removed from the parent and inserted into the root node of the tree. This involves only a couple of pointer swaps to instantly maintain a completely valid tree. When a child node is detached from its parent, it is placed into the reintegration FIFO queue. The parent is added to the recomputation FIFO queue to maintain an optimally balanced tree. At each frame, the simulation performs the process method on the SpherePackFactory so that the reintegration and recomputation FIFO queues can be flushed.

One problem with a quadtree or an octree is that it is possible for a single leaf node to be contained in multiple nodes of the tree. If an object crosses a quadtree boundary, it needs to be represented in both nodes. The SphereTree does not have this property. No leaf node can ever be contained in more than one SuperSphere. What is true for the SuperSphere is automatically true for all children. If a SuperSphere is outside the view frustum, then all of its children are outside the view frustum as well. The same is true for range tests and ray trace tests.

This makes the sphere tree an ideal data structure for these kinds of queries. When performing visibility culling with a sphere tree, each node keeps track of the state it was in on the previous frame. Callbacks occur only when a node undergoes a state change, which allows the simulation to efficiently maintain a list of only those objects in the view frustum.

Demonstration Application

This algorithm is demonstrated in the Windows application SphereTest.exe. SphereTest.exe will create a SphereTree containing 1000 spheres in motion. Even though this demonstration application is in 2D, the SphereTree is a completely 3D data structure. The rather large SphereTree displayed runs at a low frame rate since rendering all of this data is fairly slow under Windows. The SphereTest application

demonstrates building a SphereTree and performing a variety of high-speed queries against it while running a simulation that models the type of situations seen in games.

The number of spheres created by the simulation can be passed as a command-line argument. With fewer spheres, it will be much easier to visualize the SphereTree that is built. If a SphereTree of 5000 to 10,000 items is created, queries will still be seen taking place very quickly, with most of the CPU time burned just calling Windows graphics routines to render the results. If the application is begun with a very large number of items, there will be a pause while the initial tree is built and balanced, after which it will run fairly quickly.

The example simulation models the type of situations one would see in an actual game. In this simulation, *25* percent of the objects are in stasis, and the other 75 percent are always attempting to clump toward one of 16 different attraction points. In an actual game, objects are not usually evenly distributed across the address space. The example simulation demonstrating the use of the SpherePackFactory class is contained in the files Circle.cpp and Circle.h on the companion CD-ROM.

Figure 4.3.1 shows the class diagram for the SpherePack system.

To use the SpherePack system in a simulation, we simply instantiate a SpherePackFactory class with the maximum number of spheres, the size of the root node, the size of the leaf node, and the amount of gravy around each SuperSphere. The gravy factor acts as a bit of slop in our coordinate space to prevent objects from detaching from their parent SuperSphere too frequently. For each object in the simulation, we call the AddSphere() method to create a SpherePack instance. Whenever an object changes position, we invoke the NewPos() method. If the object changes both position and radius, we invoke the NewPosRadiusO method. When an object in the

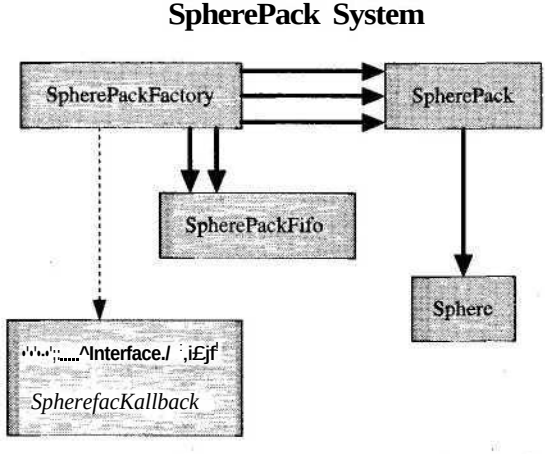

FIGURE 4.3.1 *The* SpherePack *system class diagram.*

simulation is destroyed, we invoke the RemoveQ method on the SpherePackFactory class.

The SpherePackFactory class maintains the complete SphereTree hierarchy and contains the two integration FIFO queues. When performing queries, an interface called SpherePackCallback is used to extract information from the SphereTree. The leaf node SpherePack inherits the properties of a Sphere class.