Using Metaprogrammed Functors to Implement Double-Dispatch for Collision Handling

Tiago H. C. Nobrega, Diego D. B. Carvalho, Aldo von Wangenheim
LAPIX - Laboratory for Image Processing and Computer Graphics
UFSC - Federal University of Santa Catarina, Brazil
{tigarmo,diegodbc,awangenh}@inf.ufsc.br

Abstract

We provide a strategy to the management of collisions between multiple objects with different types employing multiple-dispatch and both Object-Oriented and Generic Programming concepts. The solution scales well to the number of object types, with a fixed, constant-time cost to arrive at the proper interference-detection routine. Additionally, it helps the application programmer by allowing them to implement only the functionality required by their own program, removing the need for common solutions such as Abstract Base Classes full of pure virtual methods.

1. Introduction

The design of a collision detection engine or framework is a difficult task riddled with decisions and trade-offs on important design goals such as efficiency, modularity, size, dependencies, and others. The C++ language is a mature language with support for both Object-Oriented (which provides runtime polimorphism and aims at easing the development and maintenance of large systems, such as 3D engines) and Generic (with templates, offering run-time speed benefits) programming. In this work we explore the combination of these methodologies to implement the collision detection mechanism between different entities. We aim to expose the benefits this combination provides to both the system designer and the application programmer.

2. Bounding Hierarchies with Double-Dispatch and Meta Programming

Bounding Hierarchies[2] are often employed by collision detection algorithms to speed up interference detection between models with complex geometry, and common choices for bounding objects include Spheres, Oriented Bounding Boxes (OBBs) and Axis-Aligned Bounding Boxes (AABBs). Classes are often designed so that most functionality is placed in a base class, here named BoundingTree, and then object-specific details are delegated to other classes that inherit from it.

Table 1. A matrix of functions.

	AABBTree	SphereTree	OBBTree
AABBTree	*f1*	–	*f2*
SphereTree	–	–	–
OBBTree	*f3*	–	*f4*

The problem arises when a scene is composed of many distinct kinds of entities and a manager object has to decide if two entities are colliding. The manager only has knowledge of BoundingTrees. The problem is such: given two instances of BoundingTree subclasses, how can we efficiently determine if they are intersecting?

Since the choice of algorithm to determine the intersection depends on the types of *both* entities, this problem can be solved with Multimethods, which are "the mechanism that dispatches a function call to different concrete functions depending on the dynamic types of multiple objects involved in the call" (Alexandrescu[1]). To cope with the fact that the C++ language has no built-in support for Multimethods Alexandrescu proposes a few alternatives, one of which is used here - a constant-time matrix of pointer to functions, with some adaptations.

The method consists of assigning a numerical value (an *id*) to each BoundingTree subclass, retrievable at runtime through a virtual function. Alexandrescu uses these ids to populate a matrix of function pointers, at setup time, and to dispatch a function based on the types of two objects.

There are several advantages to this approach: Because the matrices become the sole point of interaction, no class involved in the BoundingTree hierarchy needs to be aware of the others, reducing build times on changes and potentially diminishing coupling — this is a benefit over common implementations of double-dispatch like the Visitor design pattern[3]. Additionally, the cost to arrive at the correct

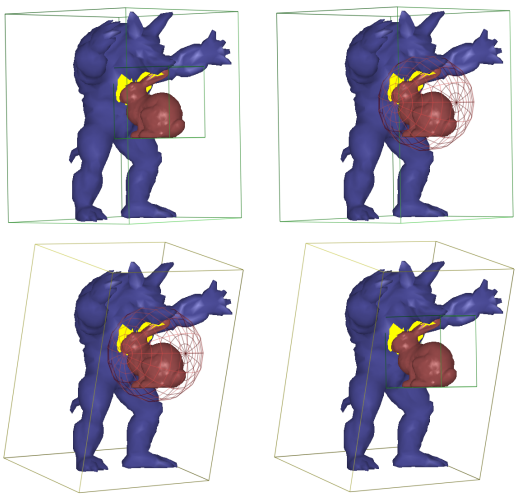

Figure 1. Employing different bounding hierarchies at runtime on the Stanford Armadillo and Bunny.

function is constant regardless of the number of subclasses and the matrix needs to be populated (and thus, functions implemented) only for the desired types, e.g., if the programmer knows their application won't need SphereTrees they can disregard their intersection tests with the other hierarchy types entirely (table 1). Since many collision detection algorithms are commutative, the number of implemented functions can be further reduced.

We extend the matrix to hold Functors (function objects) instead of raw function pointers. The added cost of one virtual call is the downside to the greater flexibility, as functions are poor at affecting the external world and managing state. The functors are small pieces of code written separately from the tree hierarchies and dedicated exclusively to handling the intersection between two types of bounding objects.

2.1. Functor Generation with Metaprogramming

It can be cumbersome to populate the matrices individually, cell-by-cell, because a matrix of n types requires n^2 functions, or $\binom{n}{2} + n$ if the algorithms are commutative. The matrices and the functors are actually class templates and the number of BoundingTree subclasses of interest is usually known at compilation time, so we developed a set of utility functions that work on *Typelists*[1] to create appropriate Functor classes and populate matrices through metaprogramming.

For instance, assume we're interested in managing Sphere, AABB and OBB trees. We have code to perform the intersection tests for all combinations between these three types of geometric objects, and we wrap all of them inside an utility class called OverlapDetection. The signature for all intersection functions is similar such that the general functor to perform the tests could look like:

```
template<class TreeA, class TreeB>
struct OverlapTester{
  bool operator()(const TreeA& a, const TreeB& b)
  {
    return OverlapDetection::overlaps(
      a.boundingObject(), b.boundingObject());
  }
};
```

Next, we implement a function to receive the list of *types* of trees we're interested in and perform all pairwise combinations of types from that list, at compilation time. Each pair fully defines an OverlapTester functor. If a particular pair doesn't follow the OverlapDetection convention, the OverlapTester can be trivially specialized. As long as this specialization is visible to the compiler it'll choose it when forming the specialized pair from the typelist.

3. Conclusions and Future Work

By our initial evaluations the time spent finding the appropriate method to dispatch takes up to 10% of the overall intersection process in the worst case. The scene in figure 1 reaches about 400 full intersection processings per second on an standard Athlon X2 3800+ with both models at around 3 thousand triangles. A next step is to compare this implementation with a more traditional approach and locate the source of the overhead.

We believe the idea of generic functor matrices can be expanded beyond the double-dispatch case. Any time a new kind of operation has to be performed on a BoundingTree a matrix can be employed instead of further extending the basic BoundingTree contract to support new dependencies. For example, employing bounding trees for Raytracing[4] would require line segment-bounding tree intersection tests. A matrix for such an application could have one row for each BoundingTree subclass, and the LineSegment class for its single column.

References

[1] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] C. Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann, December 2004.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[4] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1):6, 2007.