Generic Programming 2006, Assignment 3

Kenny Erleben*
Department of Computer Science
University of Copenhagen,
Denmark

May 3, 2007

1 The Geometry Dispatching Problem

Physics-based modeling is used extensively in computer animation. An important part of the animation pipeline is the collision detection. Here it is tested whether geometries of the moving objects collides. If so the physics simulator can respond to the geometric overlaps and alter the motion of the objects.

An important part of the collision detection pipeline is correct handling of different geometry types. Collision detection is done in two phases: a broad-phase and a narrow-phase. During broad-phase collision detection object pairs are selected/generated, these pairs are then processed by the narrow-phase collision detection.

The narrow-phase collision detection algorithm is responsible for determining what geometry pair types it has been invoked with and then invoke some appropriate specialized collision test function for these geometry types. The task of decoding the geometry types and invoking the correct collision handling function is termed "geometry dispatching".

In Section 1.1 and Section 1.2 we will outline a pseudo code example of the classical solution to the geometry dispatching problem. Finally in Section 2 we specify the requirements we have for a new and better solution to the problem.

In this exercise your have to

- Invent and implement a new generic and (hopefully) better solution for the geometry dispatching problem.
- You have to show whether the performance of your solution is better or worse than the classical solution (described in the following).

*e-mail: kenny@diku.dk

1.1 Example Collision Detection Engine

Typically all objects are stored in some collection, and each object know what kind of geometry it has (we will define this later).

```
class Object
{
public:
  geometry_type * m_geometry;
};

std::vector<Object> scene;
```

A simple approach to the broad-phase consists of simply testing all possible pairs of objects for collision. An naïve implementation would look like

```
  void broad_phase(
              std::vector<Object> & scene
            , std::vector<pair<geometry_type*,geometry_type*> > & geometry_pairs
                 )
  {
    geometry_pairs.clear();
    for(unsigned i=0;i< scene.size();++i)
      for(unsigned j=i+1;j< scene.size();++j)
        geometry_pairs.push_back(make_pair(scene[i].m_geometry, scene[j].geometry));
  }
```

Following this the collision detection engine would process each geometry pair and invoke the narrow-phase collision detection on it, that is

```
  void collision_detection(std::vector<Object> & scene)
  {
    typedef std::vector<pair<geometry_type*,geometry_type*> > pair_container;
    pair_container pairs;

    broad_phase(scene,pairs);

    pair_container::iterator p   = pairs.begin();
    pair_container::iterator end = pairs.end();

    for(;p!=end;++p)
      narrow_phase(p->first,p->second);
  }
```

The narrow-phase collision detection is the final step. It is in this final step that the geometry dispatching takes place. We will explain this in the next section.

1.2 Classical Geometry Dispatching

Generally one wants the collision detection engine to support a lot of different geometry types, because it is rather boring to look at cartoons and special effects in movies if the world only consisted of boxes.

This naturally calls for some kind of polymorphism. Traditionally people have implemented dynamic polymorphism. That is a base class is defined with a common interface.

```
class Geometry
{
public:
   virtual std::string get_type() const = 0;
   //--- Bunch of other things here...
};
```

Then all geometry types are inherited from the base class

```
class Sphere : public Geometry
{
public:
  std::string get_type() const {  return ‘‘sphere’’; }
};
```

and

```
class Box : public Geometry
{
public:
  std::string get_type() const {  return ‘‘box’’; }
};
```

and so on for all the geometry types that one pleases. Besides the geometries a collection of collision handling functions is given

```
void test_box_sphere(Box const & A, Sphere const & B)
{
  std::cout << ‘‘box sphere test’’ << std::endl;
}

void test_box_box(Box const & A, Box const & B)
{
  std::cout << ‘‘box box test’’ << std::endl;
}

void test_sphere_sphere(Sphere const & A, Sphere const & B)
{
  std::cout << ‘‘sphere sphere test’’ << std::endl;
}
```

Of course in a real-life engine the collision handling function would be more meaningful, an overlap test would be performed and possible contact points would be computed, we will ignore these things in this context. The important thing to notice is that the functions often have different names, but a somewhat similar argument layout.

We can now write the actual code for the geometry dispatching

```
void narrow_phase(geometry_type * A,geometry_type * B)
{
  if( A->get_type()==''box'' && B->get_type() == ``sphere'')
  {
     Box * box    = static_cast<Box *>(A);
     Sphere * sphere = static_cast<Sphere *>(B);
     test_box_sphere(*box,*sphere);
  }
  else if( A->get_type()==''box'' && B->get_type() == ``box')
  {
     Box * boxA = static_cast<Box *>(A);
     Box * boxB = static_cast<Box *>(B);
     test_box_box(*boxA,*boxB);
  }
  else if( A->get_type()==''sphere'' && B->get_type() == ``sphere')
  {
     Sphere * boxA = static_cast<Sphere *>(A);
     Sphere * boxB = static_cast<Sphere *>(B);
     test_sphere_sphere(*sphereA,*sphereB);
  }
  else
  {
     std::cerr << ``sorry could not decode geometry types'' << std::endl;
  }
}
```

In real-life the decoding of types would not be based on strings, but rather some unique number/constants.

The geometry-dispatching code is executed at very high rates. It is therefore extremely important that this part of the collision detection engine is very efficient.

Although this polymorphic solution work in practice it is not without problems:

- We pay a performance penalty due to the dynamic binding
- It is tedious to extend with new geometry types, because we have to make changes to the type decoding in the geometry dispatching code.
- The geometry dispatching code will very quickly become unreadable and hard to maintain.

2 The Requirements of Geometry Dispatching

One idea of generic programming is

- Once code is written you should not have to make changes to it in the future

In this respect the classical geometry dispatching solution is extremely non-generic. Whenever one extend with new geometry types one have to make changes to the type decoding.

- Your first task is to see if you can come up with a design for a generic geometry dispatching solution.

Geometry dispatching should be extremely efficient otherwise it imparts an unwanted performance penalty. Dynamic polymorphism is therefore unattractive and a solution that is more "static" would be attractive.

- Your second task is to see if you can improve the performance of the dynamic polymorphism

Objects keep track of their attached geometries by a pointer. This is desirable for several reasons: it keeps the implementation simple, and it allow objects to share the same geometric shapes, thereby reducing memory usage. It is also an efficient way to make sure that when people add new geometry types they comply with the geometry interface used by the collision detection engine.

- The third task is to define some design pattern that ensures that geometries can be shared by objects and new geometries comply with a common interface.

You should remember one very important feature of geometry dispatching, say we extend with a new geometry type: `class Tetrahedron`, then it may very well be that there only exist a collision handler for tetrahedron and sphere, but none for tetrahedron and box, or tetrahedron vs. tetrahedron. Therefore you should consider

- Graceful behavior of your design in case no collision handler exist for a geometry pair.

For examples on real-life geometry dispatching see the source code of OpenTissue.