33

Asking the Environment Smart Questions

Mieszko Zielinski

33.1 Introduction

Finding your way in a complex and dynamic environment such as in a shooter game is a challenge, especially if you're just an AI actor with very few CPU ticks to spare every frame. Life is tough. "How do I know where to go? If there are a number of places, how can I tell which one is better? Oh! There's an enemy! Two! Three of them! Who do I shoot first?"

It's a tricky task to create a service that will supply AI with all the data it needs, at low CPU time cost, while being flexible and easy to use. It needs to be able to look for different things, filter them, and score them. For *Bulletstorm*, we created a system that serves all of the spatial awareness needs of the AI while not taking much CPU time and is also intuitive for designers. We concentrated on creating a system which takes input that is easily understandable by humans, rather than making humans produce counterintuitive data that the system will have an easy time consuming.

33.2 Motivation

Early in the development of *Bulletstorm*, two new systems were designed and implemented: one for AIs' logic (our Behavior Tree implementation) and the other for

centralized environment querying. This article will describe the latter system, which we called Environment Tactical Querying (ETQ for short).

Environmental queries in *Bulletstorm* can take two forms: object types and object properties. *Enemies*, *covers*, and *locations* are examples of object types. *On navmesh*, *not visible to enemy*, and *some distance away from leader* are examples of object properties. These object properties could also take the form of preference: *prefer ones not visible*, *prefer ones closer to me*, and so on.

33.3 Goals

The following are goals that we pursued throughout the development of the system.

- *Think of "What to ask" not "How to ask"*: Creating and asking questions was to be made as simple as possible. We wanted nothing to stand in the way of our creativity.
- *Let nonprogrammers do the job*: We aimed from the start to create a dedicated editor for designers, so that whenever they wanted to change or tweak the way an AI picks cover or an enemy, they can do it themselves just by clicking.
- *Code reusability*: That was our main coding commandment, and it resulted in really clean and compact code.
- *Performance*: It was essential to have this system perform its duties without any other system noticing a hit on performance.
- *Asynchronous*: Even though the ETQ system itself was running on the game thread, we made it asynchronous so that questions asked would not block the game. Instead the system was scheduling query processing to be performed during system's regular update. It also allowed us to time slice the main query processing loop.

33.4 Our Philosophy

We wanted our idealized data creator to author queries just by asking him or herself simple questions:

- *What to generate?* Those could be covers, enemy locations, points—anything that has a location in the game world.
- *Who's asking?* What is the context object of the query—which entity is asking the question. Most often that was an AI actor, but questions could also be asked about an enemy being tracked by an AI ("Where could he go?"), a cover or a spawn point for example.
- *Where to look?* What are the spatial restrictions for candidate items? We could generate items in a radius around a context object, for example, or in the object's assigned combat zone, or so on.
- *Which items are good enough?* What are the minimal qualities of an acceptable item? We might require items visible from some reference object or items that are no closer than X to the context object, etc. These formed conditions.
- *Which items are better?* How can we tell one item is better than the other? We could prefer items that are closer or further from something, or prefer items having some property, or have some property's value lower than a set limit, etc.

33.5 Anatomy

The ETQ system is a data-driven solution, and most of its power lies in the design of data representation. Much care went into making it both flexible and efficient. It's time to look a bit more into the details of our design.

33.5.1 Query

A Query has three main components:

- *Context object*: The game entity that is asking the question (or more precisely on behalf of whom the question is being asked). What is the spatial context for this query? This is crucial since this actor's properties will be used to define a subjective world view, like, "Is that spot visible from here?" and, "Is it within my view range?" Note that a context object need not be an AI actor
- *QueryTemplate Id*: Which one of the user-created questions are we asking? All query templates registered with the system have their unique id, and this is the place to indicate which one to use.
- *Items*: All items found are returned as a list. Later, we will remove all items that fail subsequent filtering tests.

To trigger query processing, one calls the ETQ system supplying information on what query template to run and what is the spatial and gameplay context of that query. One might also request a query to be processed instantly rather than in the background.

33.5.2 Query Template

Now that we know how a question is asked at runtime, how do we define one? We need to express what we're looking for and what properties we'd like our "good items" to have. We might want to require some properties while using others just for item scoring.

A question is defined in the editor, with a special tool we created, and saved as a regular asset. This asset is referenced in code as a Query Template. A Query Template consists of one or more *Options* which in turn are made up of several Tests.

First, the Query Template defines an Option, which contains information on how to generate the item population that will be processed in later steps. This creates a collection of items representing entities in the game world. We implemented a number of generators, for example:

- *Context Object's Enemies*: Gathers all enemies that a given AI is aware of.
- *Covers*: Results in a collection of cover points within a parameterized radius of a context object.
- *Points on grid*: Generates points on a configurable grid around a context object.

ETQ makes the authoring of new generators extremely easy, at times requiring only two or three lines of code.

Once the generator for a Query Template's Option is set, the ways of filtering out and scoring those items is specified. Items are conditioned out and scored by *tests*. Take a look at Listing 33.1 for a pseudocode look at a Test structure. A Test structure designates a

Listing 33.1. Pseudocode of a Test structure.

```
struct Test
{
    TestType;          //Distance, Reachability,...
    ConditionModifier; //None, Min, Max
    Reference;         //Self, Enemy, Leader, Item,...
    TestedValue;       //float, int, bool,...
    SymbolicValue;     //Melee distance, Weapon range,...
    Weight;            //float in [-1,1]
    /** flags */
    bCondition;        //boolean flag indicating this test is
                       //used as a condition
    bValidityTest;     //and/or as a validity test (see 33.5.3)
    bWeight;           //or as a weight
}
```

property to test for (`TestType`), a list of references for the test (`Reference`), a comparison type (`ConditionModifier`), and a value to compare against (`TestedValue`). We can also assign tests a weight if they are to be used for scoring.

The order in which tests are set up is irrelevant. Tests will be reordered by the system according to their computational cost. The cost is estimated by a programmer with a relative list of which tests are more expensive than others (Section 33.7), and whether it's a condition or just a scoring test. Even the most expensive tests are performed first before any scoring takes place. This results in fewer items while scoring, which may or may not save time in the long run. We address this issue in Section 33.10 with *final test*.

In the case where tests are too restrictive and no item gets past the conditions, we might relax the constraints rather than simply fail on the whole query. The system supports this scenario. If a query template has more than one option, then all of them are processed in a sequence until one produces some items which become the result of a query.

33.5.3 Validity Test

A test declared in the query template's option can be a condition, a weight, or both. However, ETQ also allows you to designate certain tests as *validity tests*. These tests will be used not at cover generation time, but later, when an AI is moving to that cover, or sitting in it, to check if the cover is still valid (not exposed to enemy fire for example). Of course, a test can be both a regular test and a validity test at the same time. This way all of the configuration information on how to pick a cover point and, later, how to tell if it is still good (oftentimes not the same thing) are nicely gathered in one place.

We could achieve the same functionality by rerunning a query on that one item in question, but that would result in a number of tests we don't need while performing an ongoing validation. We could also create a separate lightweight query just for validity testing, but that on the other hand would require keeping both queries in sync whenever one of them is changed. Marking some tests as a validity test is the best of both worlds. During the ongoing testing, we only run the tests we need, while at the same time having all of the logic for picking items and for ongoing validation in one query asset.

Table 33.1 Shows how different tests can be used as both conditions and weights

Test	Condition	Weight
Visibility	Is (not) visible	Prefer (not) visible
Distance	More/less/equal to X	Prefer closer/further away
Configurable dot	More/less/equal to X	Prefer more/less
Within action area	Is (not) in action area	Prefer (not) in action area
Reachable	Is (not) reachable with navigation	Prefer (not) reachable
Distance to wall	More/less/equal to X	Prefer more/less
Current item	Is (not) current item	Prefer (not) current item

33.5.4 A Test in Any Role

In order to have a nice unified interface to all tests, we decided we'd have every test make sense in both roles, as a *condition* as well as a *weight*. For example if we use "distance to enemy" as a condition, we can require it to be less or more than 2000 units. But if we use it as a weight, then we interpret it as preferring a smaller or greater distance to the enemy. We express how much we prefer a tested property by setting the test's weight value. The higher the value, the more a given property is desired. Table 33.1 gives some more examples.

33.6 The Heart

The core ETQ algorithm is captured in the pseudocode in Listing 33.2.

Listing 33.2. The core ETQ query algorithm.

```
foreach Option in QueryTemplate.Options:
    Query.Items = (generate items with QueryTemplate.Generator
            using contextual data from Query);
    if Query.Items is empty:
        continue to next option;
    foreach Test in Option.Tests:
        Reference = (find world object Test refers to);
        if Reference not empty or not required by Test:
            //explained in Section 33.7 under "Fail Quickly"
            if Test has a fixed result:
                apply result to all Query.Item elements;
            else:
                perform Test on all Query.Item elements;
            if Test.bCondition is true:
                filter out Query.Item elements that failed Test;
            if Test.bWeight is true:
                foreach Item in Query.Item:
                    calculate weights from every test result
    if Query.Items not empty:
        foreach Item in Query.Items:
            sum up all weights calculated by weighting tests;
        sort Query.Items descending with computed weight;
        return success;
return failure;
```

33.7 Implementation Details

There are always some little tricks you can perform while implementing even the simplest algorithm. The following are some of the optimizations we implemented.

- *Start with cheaper tests:* We manually presorted test-types according to expected performance. For example, a Distance test is more expensive than checking if an actor has a tag, but is less expensive than finding out if a point is on the navmesh.
- *Fail quickly*: For some tests it's possible to fail early, thus saving computation. For example, checking whether a point is on the navmesh will always fail if there's no navmesh. Tests can fail even sooner if they require a reference that doesn't exist in a given context, like the squad leader or an enemy for example.
- *Normalize test results and weights*: We quickly discovered that trying to weight a distance test against a "has a property" kind of test was an impossible undertaking. Even if it could be done with a known maximum distance, it would fall apart as soon as we change the maximum value and would require retweaking. So we decided to normalize all test results. While performing a test we store the maximum result value and once the test is finished, we normalize all the results with the stored maximum. In some cases we would also use the item generation range of a processed option. We also made sure that weights are always within the range [-1, 1] (via the editor), which together with normalizing results gave us some good mathematical properties and allowed reliable query tweaking.
- *Debug-draw whatever you can*: It's impossible to overvalue debug drawing. While developing a system like this it's crucial to be able to trigger any query on any target of your choice, during runtime, and you need to see the results. Countless times we found bugs in a query just by debug-drawing its results, which proved a significant time saver.

33.8 Editor

Taking advantage of the ease of creating tools with *Unreal Engine 3*, we created a tool for ETQ. Using the tool made working with query assets a lot more pleasant. As a direct consequence, we were more willing to work with queries, tweak them, and instantly see if something was set up wrong. Figure 33.1 shows a Query Template in our editor with examples of option and test node properties.

The tool further provided the following features:

- *Weights auto-scaling*: Whenever a weight of a test has been changed to something outside of the range [–1, 1], the editor rescaled all weights in a given query option proportionally, so that they again fit in the range mentioned in the implementation section.
- *Auto-arranging visuals*: A query was represented as a tree-like structure. All elements that were used to visualize tests were put in columns with an option node as a head, to achieve a unified look for every query, regardless of who created it. This made it a lot easier to find your way around in someone else's queries.
- *Descriptive labels*: We made every test node in a query asset produce a non-programmer understandable description string for itself and displayed it on

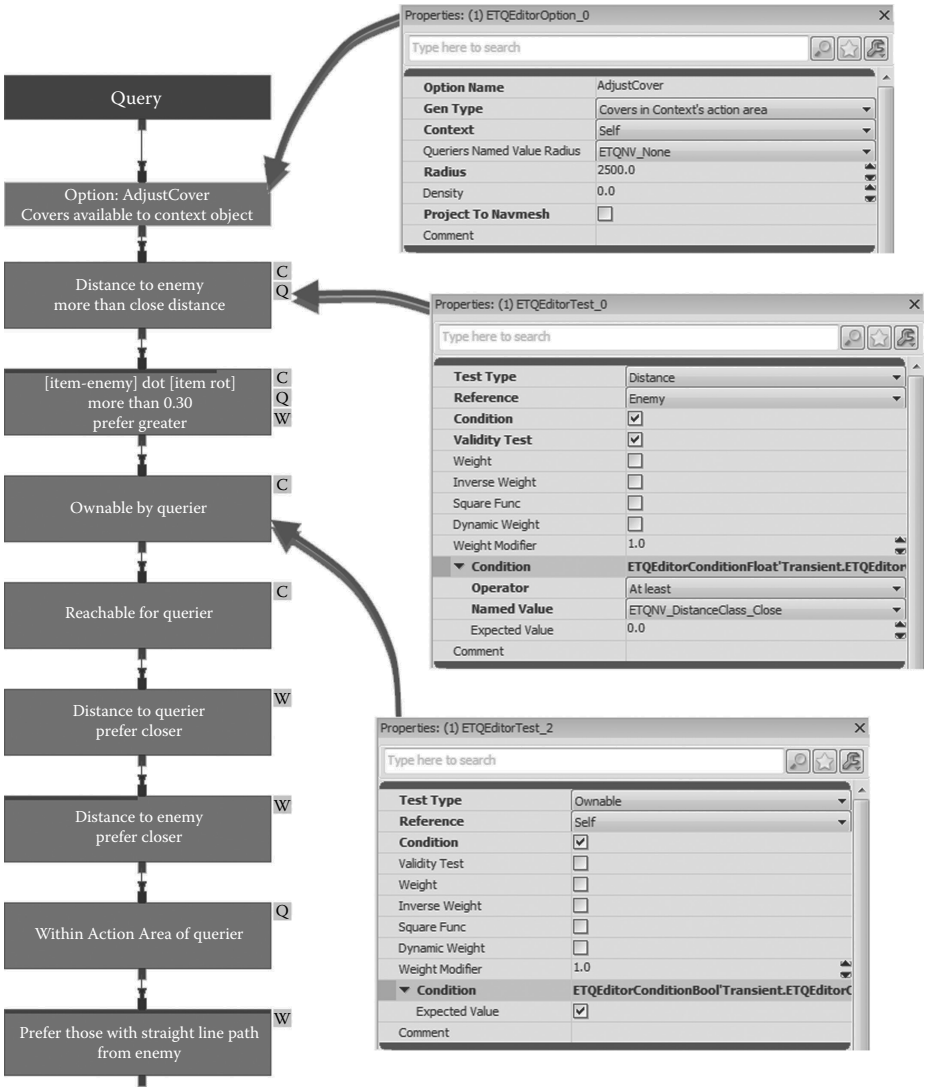

Figure 33.1

A Query Template in our editor with examples of option and test node properties.

its visualization in the editor. For example, we had labels like "Leader has a straight line path to (condition)" or "Distance to context object, prefer less (weight)." The idea was that even an untrained person could more or less tell what a given query will generate just by looking at it.

- *Coloring*: This one's pretty obvious, but we colored with yellow (as opposed to everything else being in dark colors) everything that was incorrectly set up or was missing some values. This way we instantly knew where a given query was broken at the very first glance.

33.9 Pros and Cons

There are a number of good properties ETQ achieved. Some examples include:

- *Intuitive query creation*: With the tool we created, designers were able to construct a query with minimal tutoring. The idea itself was so close to the way people express these kinds of queries that even less technical designers had no trouble understanding it.
- *Data driven*: Having data control the way code behaves is the holy grail of game development. For example, programmers no longer need to be involved in every change to the way AIs pick enemies or cover.
- *Efficient*: By having our system time-sliced and our queries set up to not generate insane numbers of items, we were able to squeeze ETQ to under 0.02 ms per frame on average, while still having it look for game entities of specified properties with very sophisticated queries.
- *Flexible*: Adding new tests or generators was so easy that whenever someone needed to do something that the current tests or generators did not provide, adding a new one was simple—very easy to add and still efficient at runtime (due to time-slicing).

The main issue with ETQ is it can take some time or experience to tweak the queries to get the desired behavior. On the other hand, iterations while working with queries were very fast (we also had some runtime tools for it) and inexperienced users developed the required intuition quickly.

The ETQ system was also used in *Gears of War: Judgment*. On that project, it exhibited a number of issues. One major issue was that in some scenarios it generated huge CPU usage spikes of up to 15 ms. This was due to performing expensive tests on a large collection of items, since we treat each test on a single collection of items atomically and don't time-slice it. These spikes resulted from the query generator reading cover gathering range from level-placed entities (in this case "Goal Actors"), a value that was set up by level designers. This is in fact a problem inherent in data-driven systems and care needs to be taken to make sure data supplied by designers doesn't kill the system's performance. The quick fix was to limit the radius to some experience-based maximum value.

33.10 Things to Fix and Improve

There were a number of improvements that we were unable to get to in the final stages of shipping *Bulletstorm*. These include:

- *Merging tests*: Certain tests tend to show up as a group. For example, while querying for cover points, *dot product to enemy more than X*, *distance to enemy more than Y*, and *not my current cover* would routinely come up as group. Intuitively it would make sense to have one special test that does all three things instead of three individual tests executed one after the other.
- *Final test*: Often, even numerous filters were not able to reduce enough the total number of items to test. When the system reached the expensive tests, there were far too many items to process and performance was poor. The idea of the *final test*

is to pick the required first *N* items that pass that test and abort testing the rest. The final tests would be the very last ones to process, and this way we'd get good enough results without calculating expensive tests for all items.

- *Reversed processing*: [Robert 11] presented a scheme to quickly pick a cover better than the one our AI is currently using. The same could be added to ETQ. Thus instead of running regular processing of a query, we could take the AI's current cover point, grade it first, and then accept the very first item that passes all of a query's filters and has a higher score.
- *Multigenerators*: Allow a Query Template to use multiple generators to be able to run unified testing on a collection of different items (like regular points and cover points in one pass).
- *Multithreaded implementation*: There was neither a need nor CPU resources to run ETQ in a separate thread, but with the trend toward more CPU cores, this is a promising future direction.

33.11 Conclusion

Even though the described system was very simple in its design, it proved to be very powerful. We gained a lot of environment querying power without eating up a lot of CPU time. Having ETQ driven by data and queries, created with our dedicated authoring tool, allowed very rapid iteration over how AI picks enemies or covers, further enhanced with runtime debugging tools.

There's also a higher level gain. From the very start we were thinking about ETQ in asynchronous service terms and it helped us make the rest of our AI system components asynchronous as well. Designing and implementing asynchronous AI takes a slightly different mindset, but it results in solutions that scale well on multiple cores, which is a requirement in coming years.

Acknowledgments

I'd like to thank my wife, Agata, for talking me into writing this article. I also would like to thank Łukasz Furman for helping me make all my crazy ideas for *Bulletstorm* AI happen.

References

[Robert 11] G. Robert. "Cover Selection Optimizations in GHOST RECON." Paris AI Conference Shooter Symposium, 2011.