The Universal Query System (UQS) is a system for undertaking spatial queries in a 3D world.

In a nutshell, the UQS allows to query items, rank and filter them and then return the best 'N' resulting items.

Over time it is intended that the UQS will supersede the Tactical Point System (TPS) of the CryAISystem.

Features

In the TPS, queries are geared around so-called 'tactical points' which are basically positions in the world that are specific to the CryAISystem's object model.

UQS continues to support this concept, but at the same time expands on the idea of 'tactical points' and their context by breaking free from some of their restrictions and to also provide new features. Ultimately, the UQS aims to be a generic system that lends itself well to any new game project with very specific requirements for spatial reasoning.

Some of the main features that the UQS has been specifically designed for:

- Support for more than just 3D points, for example:
  - Spheres: find a landing spot where there is enough room.
  - Entities: select which player to attack.
  - Combat slots: combat manager picks a slot around the player from where to attack.
  - NavMesh triangles: area to investigate.
  - Areas: AI Director selects which area to populate next.
- Support for a list of 'N' best resulting candidates, for example:
  - Useful for spell-casters that can hit multiple targets at the same time.
  - Create a 'heat map' with the most dangerous spots that an AI can see. This can then be fed into the path finder as 'hints', for example to find the safest possible path through an area.
  - A UQS query can also be used as a preliminary 'filter' and then pass the results to some highly specific system that does something 'custom' with it.
- Support for feeding in parameter values into queries at run-time, rather than being restricted to hard-coded values at design-time only.
- Allows for custom names for query parameters so that you can better describe the context that the query is intend to resolve, for example:
  - Currently 'puppet' is hardcoded to mean 'the thing that requested the query'.
  - Better/more suggestive names might be: 'querier', 'attacker', 'hostiles', 'observers', etc.
- Improved debugging (and logging), for example:
  - Which conditions were true/false.
  - How total scoring values were constructed.
  - Which sorting rules were applied.
- Visual history of queries:
  - Items can be visualized using built-in debug rendering facilities.
  - Individual items can be inspected to better understand how their score was constructed.
  - The whole history can be persisted and restored for inspection outside of the running game.
- Custom 'post-validation' for when a query has been completed, for example:
  - The NavMesh might have changed while the query took a couple of frames to complete, so now you want to make sure that the query result is still 'valid'.
- Fallback queries:
  - Allows to create alternative queries with perhaps less restrictions, should the 'primary' query not find any items.
- Query chaining:
  - Use the outcome of one query as parameters for another query.
- Fully extendable on the game side:
  - Custom item types
  - Custom generators
  - Custom functions
  - Custom evaluators

Core Parts of a Query

There are 4 main parts that comprise a query:

- Items
- Functions
- Generators
- Evaluators

Items: Are used to represent the reasoning space that a query is working on. Simply speaking, a query will evaluate a set of items to work out the fitness of each item. Also, they are used as input parameters and return values in functions.

Functions: When called, act as input parameters for generators, evaluators and functions themselves. Also, functions can be used to convert between item types or to mix in some custom logic before propagating their return value to the caller.

Generators: Are responsible for generating items. Their purpose is to create the reasoning space via a set of items that will be evaluated by a query. For example, a generator could create a set of points in a grid-like layout.

Evaluators: Come in 2 flavors: InstantEvaluators and DeferredEvaluators. Both kinds are responsible to "evaluate" an item they have been passed. More precisely, their purpose is to either give the item a score between 0 and 1 (to express the fitness of that item) or to simply discard the item in case it doesn't meet a specific condition.

The difference between both kinds of evaluators is:

InstantEvaluators must return their evaluation outcome immediately. They lend themselves well for undertaking quick calculations. A typical use case would be to qualify the distance between 2 points given a reference distance.

DeferredEvaluators are free to run over time and return their outcome once they have finished processing the item. Whether they do time-sliced work or offload the actual work to a different system is up to them. From the query's point of view, it's only relevant that they are periodically updated until they come up with a result. A typical use case would be to start an asynchronous physics raycast between 2 points and wait until the raycast request has been fulfilled.

Scoring of Items

As mentioned above, evaluators are responsible for scoring an item. Generally speaking, each evaluator in a query will be asked to evaluate each generated item. Their scores will then be added together to form the overall score of that item. To allow a little more control over the individual scoring by a single evaluator, each evaluator comes with a weight that the score will be multiplied by before its added to the overall score.

The overall score of an item will be used to work out what the best 'N' items are to return in the final result set of a query.

Technical Information

For more technical information on how to work with these features, see the following pages:

New Features in CRYENGINE 5.4

- Added concept of GUIDs to all UQS elements to prevent relying on their names.
- Added concept of Parameter IDs to Generator's parameters, Functions and Evaluators to no longer rely on their names.
- Added support for providing descriptions to all UQS elements from within the code to reflect in the UI.
- Added helper classes that makes running a query easier from within the code: `UQS::Client::CQueryHost`, `UQS::Client::CQueryHostT<>`.
- Added concept of Evaluation result transforms to manipulate the outcome of a single item. This can be achieved via a score function and discard negation.
- Added a helper class for easier setup of the UQS when using the XML data source: `UQS::DataSource_XML::CXMLDataSource`.
- Added preliminary Schematyc support to allow running queries from within Schematyc.
- Query Editor: added tool tips for most elements that originate from the descriptions provided by the programmer.
- Query History Inspector: double-clicking on a historic query now "teleports" the Editor camera to a close-by position.

Breaking Changes in 5.4

Namespaces and Variable Names in C++ Code Have Been Adjusted to Better Match the Coding Conventions

Old namespaces:

- uqs
  - client
    - internal
  - core
  - datasource
  - datasource_xml
  - editor
  - shared
  - stdlib
  - uqseditor

New namespaces:

- UQS
  - Client
    - Internal
  - Core
  - DataSource

- DataSource_XML
- Editor
- Shared
- StdLib
- UQSEditor

GUIDs and Coder-defined Comments

Due to the introduction of GUIDs and coder-defined comments in all of the UQS elements, the way how factories for generators, functions, evaluators, etc... get registered has changed. Before, it was mainly about creating a global instance of that factory with a list of parameters as constructor arguments. Now all the parameters get bundled in a struct.

Old way (this will no longer compile):

```
static const UQS::Client::CGeneratorFactory<CGenerator_PointsOnPureGrid> generatorFactory_PointsOnPureGrid
("std::PointsOnPureGrid");
```

New way:

```
UQS::Client::CGeneratorFactory<CGenerator_PointsOnPureGrid>::SCtorParams ctorParams;

ctorParams.szName = "std::PointsOnPureGrid";
ctorParams.guid = "498bce51-a2b9-4e77-b0f9-e127e8a5cc72"_uqs_guid;
ctorParams.szDescription =
    "Generates points on a grid.\n"
    "The grid is specified by a center, size (of one edge) and a spacing between the points.\n"
    "Notice: this generator is very limited and doesn't work well if you intend to use it for things like
uneven terrain.";

static const UQS::Client::CGeneratorFactory<CGenerator_PointsOnPureGrid> generatorFactory_PointsOnPureGrid
(ctorParams);
```

It's also possible to use C++11 lambda expressions:

```
static const UQS::Client::CGeneratorFactory<CGenerator_PointsOnPureGrid> generatorFactory_PointsOnPureGrid(
    []()
    {
        UQS::Client::CGeneratorFactory<CGenerator_PointsOnPureGrid>::SCtorParams ctorParams;

        ctorParams.szName = "std::PointsOnPureGrid";
        ctorParams.guid = "498bce51-a2b9-4e77-b0f9-e127e8a5cc72"_uqs_guid;
        ctorParams.szDescription =
            "Generates points on a grid.\n"
            "The grid is specified by a center, size (of one edge) and a spacing between the points.\n"
            "Notice: this generator is very limited and doesn't work well if you intend to use it for things
like uneven terrain.";

        return ctorParams;
    }()
);
```

UQS Standard Library

- The data type `Vec3` has been replaced by `Pos3`, `Ofs3` and `Dir3`. Functions, generators and evaluators using the old data type have been adjusted accordingly.
- The function `std::Vec3 Vec3Add ( std::Vec3 v1, std::Vec3 v2 )` has been superseded by `std::Pos3 Pos3AddOfs3 ( std::Pos3 pos, std::Ofs3 ofs )`.