



Building a Scalable Query System for Evaluating Complex Game Environments

Christian Werle

Game AI North
October 17-18th, 2017

1

When developing games, we often run into questions such as: How can AI characters get information from the world and how can they use that information to make informed decisions?

Real-World Example

- AI wants to engage an Enemy
 - Find a good location:
 - Must have visibility to enemy
 - Should be close to current location
 - Should be close to covers



2

A real-world example: we have an AI character that needs to find a good location for engaging an enemy. That location should come with certain qualities, such as:

1. Provide visibility of the enemy (otherwise firing a gun might have projectiles just hit obstacles).
2. The new location should be close to where the AI is currently located (it doesn't want to move far away).
3. The new location should be close to covers in case the enemy fights back.

We need a system that can express these wishes in the form of "queries".

Required Features

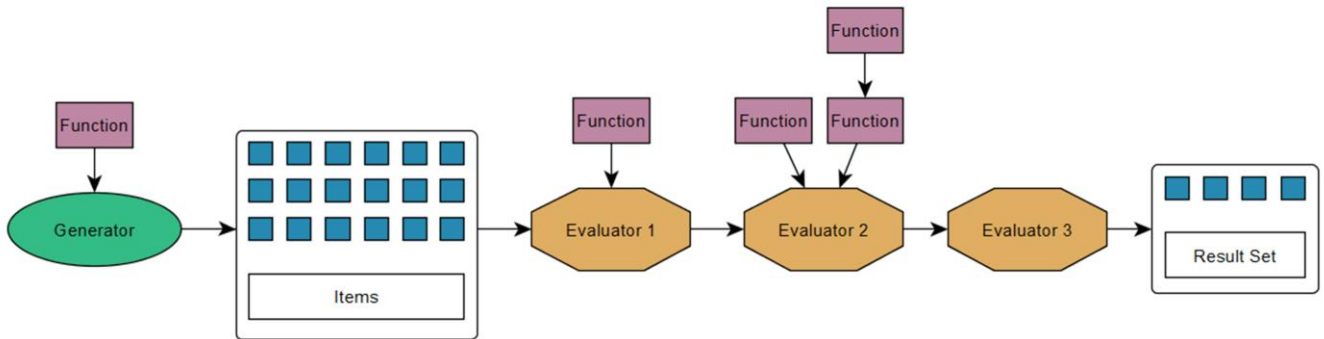
- Operate on any data type
- Customizable
- Debugging Support
- Runtime parameters
- Solve complex situations
- Run multiple queries in parallel
- Data-driven
- Validation



For our new query system, we need the following features:

1. It should be able to operate on any data type (e.g. not just 3D locations in the world, but maybe also 2D locations; areas for finding spawn locations; entities that represent a threat; etc.).
2. It must be fully customizable by the game – the game knows best what it can feed into the system and what it wants to get out of it.
3. Debugging: it must be easy to see not only what is going on while the game is running, but also after a game play session has ended.
4. Runtime parameters: some of the parameters that we would like to feed into the system might only be known at runtime (e.g. the current location of an AI character that starts a query).
5. Complex situations: one query might not be enough to solve a problem, and maybe we need to try multiple ones.
6. Multiple queries should run (conceptually) in parallel, as multiple AI characters or external systems may need to run queries at the same time.
7. Data-driven: queries should not only be authored by coders, but primarily by technical designers.
8. Validation: we need some sort of “post-validation” to ensure all the data have been valid throughout a full query run.

Elements of a Query (1/6)

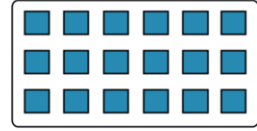


Elements of a query:

1. Generator: produce items
2. Items: the list of objects to reason about
3. Evaluators: process all items
4. Functions: feed parameters into generators and evaluators
5. Resulting items

Elements of a Query (2/6)

- Items
 - Objects of a specific data type
 - “Reasoning Domain”
 - Example:
 - Vec3
 - Area
 - EntityID



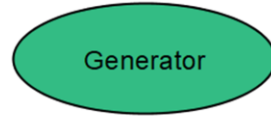
Items can be any kind of objects the game wants to reason about in a query, e.g.:

- Vec3 (for finding the best location to move to during combat)
- Area (for deciding which area to populate next with zombies)
- EntityID (for finding the most dangerous enemy to attack next)

The game can register any data type it wants the system to reason about.
Example from beginning: locations in the 3D world represented as Vec3.

Elements of a Query (3/6)

- Generator
 - Produce or gather Items
 - Create “Reasoning Space”
 - Example:
 - Points on a grid
 - Areas ahead of the player
 - Entities around the player



A Generator is responsible for producing items that the system needs to reason about.

Instead of generating, it can also just gather already existing items from the world (e.g. annotated areas that a level-designer has placed).

Example from beginning: produce 3D positions on the NavMesh within a certain range.

Elements of a Query (4/6)

- Evaluator



- Fitness of Items: score [0.0 .. 1.0]
 - Soft constraint
 - Example: Distance (p1, p2)
- Filter Items
 - Hard constraint
 - Example: LineOfSightCheck (p1, p2)



Evaluators can have 2 purposes:

- (1) Tell how good or bad an item is by providing a score between [0.0 .. 1.0].
- (2) Decide to discard the item completely if that item violates a certain condition (e.g. if a raycast is supposed to succeed but doesn't then reject the location it's testing for).

Example from the beginning about fitness: the closer the location to the AI character's current location, the better the score.

Example from the beginning about filtering: location must have a line-of-sight to the enemy (otherwise discard that location).

Elements of a Query (5/6)

- Evaluator



- Forms

- Instantaneous
 - Deferred

- Performance cost categories

- Cheap
 - Expensive



Instant Evaluators: one-shot, immediately tells how good the item is (or whether to discard it); can be marked as either cheap or expensive.

Example from beginning: score the distance between 2 locations.

Deferred Evaluators: can run over time (multiple frames); implicitly counts as being expensive.

Example from beginning: do an asynchronous raycast request (serviced by an external system), and wait until we get a response from that external system.

Elements of a Query (6/6)

- Function
 - Access current Item
 - Input Parameters of:
 - Generators
 - Evaluators
 - Functions (“recursive”)
 - Access Global Parameters
 - Literal
 - Converters

Function

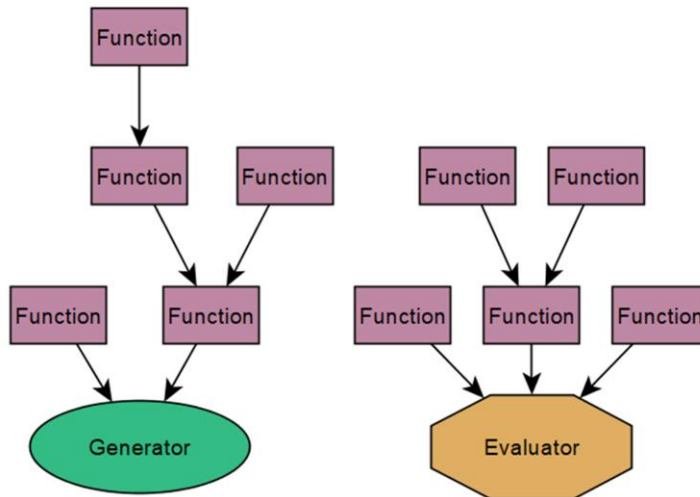


9

Functions have multiple purposes:

- (1) Give access to the current item the system is iterating on in the main loop.
- (2) Provide values of parameters for Generators, Evaluators and Functions themselves (in a nested way).
- (3) Give access to one of the global parameters (that have been passed in from the outside of the query).
- (4) Represent a literal value (e.g. for quick tweaking purposes).
- (5) Do some conversion/computation (e.g. add 2 vectors, return the resulting vector; or: given an entity as an item, return its position in the world).

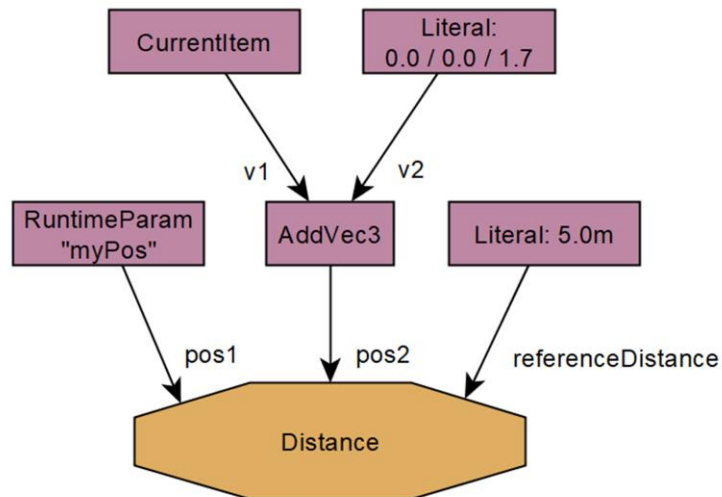
Function Model



10

Here, we have functions that are being called in a graph...
... and a Generator and Evaluator that receive their parameter values from these calls.

Function Model Example



11

An example for how an evaluator can get parameters passed in:

The **Distance** evaluator receives the values of all of its 3 parameters (**pos1**, **pos2**, **referenceDistance**) via function calls – which in turn can have parameters, being serviced by yet further function calls.

What can be customized?

- Item types
- Generators
- Evaluators
- Functions



12

The query system comprises all the elements registered by the game code.

How can it be customized?

- Abstract Interfaces
- Abstract Factories
- C++ Template Helpers



13

Abstract interfaces: used for implementing specific elements by game code.

Abstract factories: for allowing the system to instantiate those elements at runtime.

C++ templates live in between the core interfaces and the game code; they are meant to make it more convenient for programmers to register game-specific elements.

Custom Function Example

```
class CFunction_PosFromEntity : public Client::CFunctionBase<
    CFunction_PosFromEntity,
    Pos3,
    Client::IFunctionFactory::ELeafFunctionKind::None>
{
public:
    struct SParams
    {
        EntityIdWrapper entityId;

        UQS_EXPOSE_PARAMS_BEGIN
        UQS_EXPOSE_PARAM("entityId", entityId, "ENTI", "Entity to get the position from.");
        UQS_EXPOSE_PARAMS_END
    };

public:
    Pos3 DoExecute(const SExecuteContext& executeContext, const SParams& params) const;
};

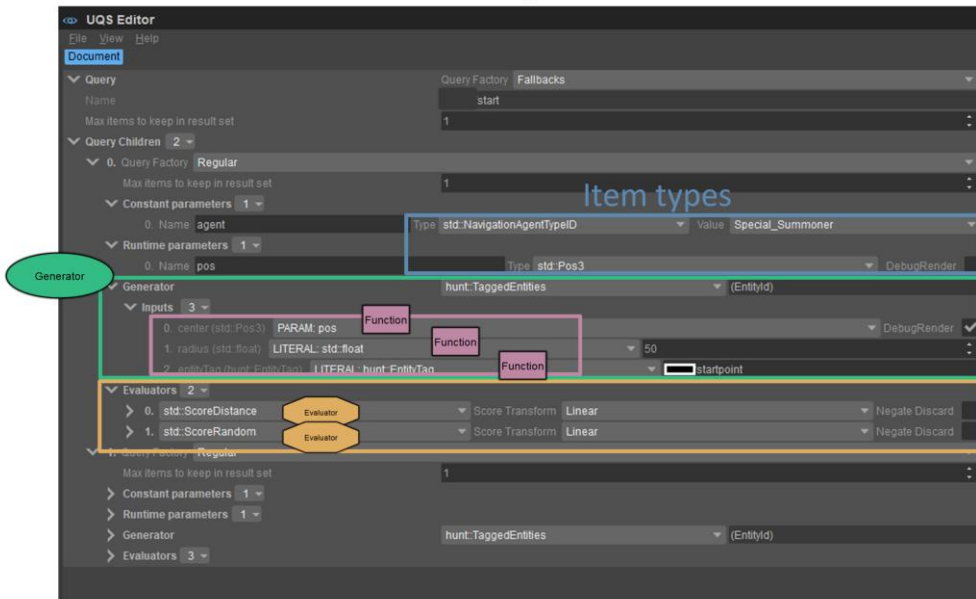
static const Client::CFunctionFactory<CFunction_PosFromEntity> g_functionFactory_PosFromEntity;
```



14

A code example of how a custom Function can be implemented and registered in the query system via a factory.

Query Editor



15

The query editor allows technical designers to use functional building blocks for authoring a query from all available elements that have been registered by the system.

Execution Phases of a Query (1/5)

- Generate
 - Items
- Evaluate
 - Items
 - Monitor Reasoning Space
- Return
 - Result Set



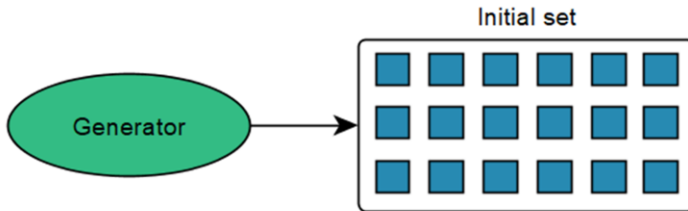
16

The execution of a query basically consists of these steps:

1. Generate some items.
2. Evaluate all of them (while at the same time ensuring that nothing bad happens to them).
3. Return the best 'N' items.

Execution Phases of a Query (2/5)

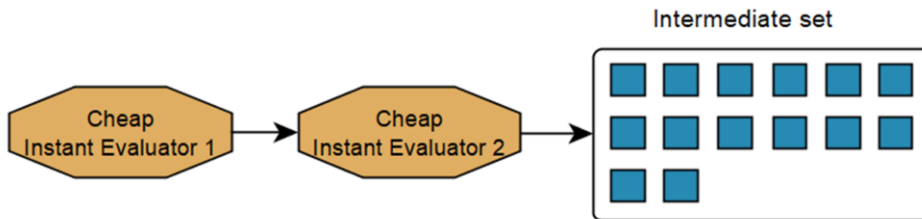
- 1. Generate items



A generator is responsible for producing the initial set of items.

Execution Phases of a Query (3/5)

- 2. Run cheap Instant Evaluators

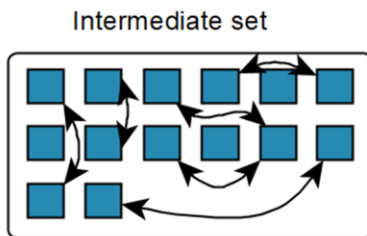


18

All cheap Instant Evaluators will run on the set of initial items. Some may filter items out, thus ending up with a reduced intermediate set of items.

Execution Phases of a Query (4/5)

- 3. Sort items by score-so-far

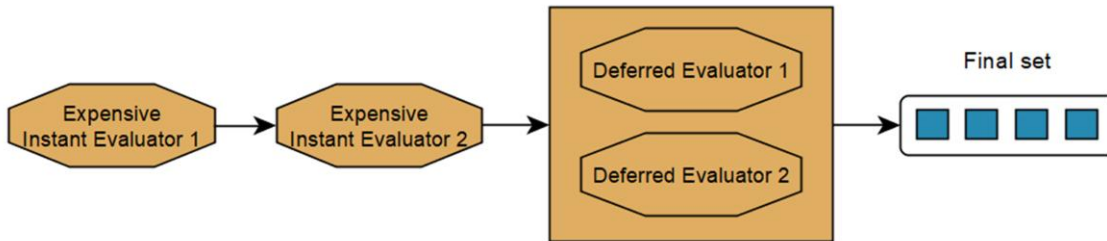


19

We then sort this intermediate set by the scores individual items have so far. The most promising items (highest scores) will then appear at the beginning, the least promising ones (lowest scores) at the end of the list. This is so that in the next step we only need to consider up to as many items as desired, until we are confident that none of the remaining items can get better than what we have fully evaluated already.

Execution Phases of a Query (5/5)

- 4.1 Run expensive Instant Evaluators
- 4.2 Run Deferred Evaluators



20

This is the expensive phase.

First, the expensive Instant Evaluators are run on a single item.

If the item survives, then all Deferred Evaluators will be scheduled. They will run over multiple frames.

This is repeated for as many items as we want in the final result set and until none of the remaining items can improve the result set anymore.

Scoring of Results

- Per Evaluator
 - Normalized [0.0 .. 1.0]
 - Score transform (optional)
 - Invert
 - Sine
 - ...
 - Multiplied by weight
- Final score = sum of all scores



21

Each evaluator must return a score ranging from [0.0 .. 1.0] telling how good the item is...

... then runs through an optional transformation (e.g. to say that we prefer points in the mid-range rather than in the close- and far-range)...

... and finally gets multiplied by a weight to put more emphasis on specific evaluators.

The final score is then: $\text{finalScore} = \text{transform1}(\text{score1}) * \text{weight1} + \text{transform2}(\text{score2}) * \text{weight2} + \dots \text{transformN}(\text{scoreN}) * \text{weightN}$

Error Handling

- Errors can happen in any element
- Can affect:
 - Whole Query (generation phase)
 - Query “fails”
 - Single Item (evaluation phase)
 - Item gets rejected



22

Errors can be handled in any element (e.g. to detect a division-by-zero in a Function). The error will then have different effects depending on where it occurred:

- (1) Generator => whole query quits with an exception and the caller needs to deal with it
- (2) Evaluator => only that particular item will be discarded from further evaluation and will get marked as having encountered an exception

Item Monitoring

- Detect Corruption of Reasoning Space
- Installation
 - By Generator (optional)
- Lifetime
 - Whole query
- Example
 - NavMesh changes => could affect generated locations



23

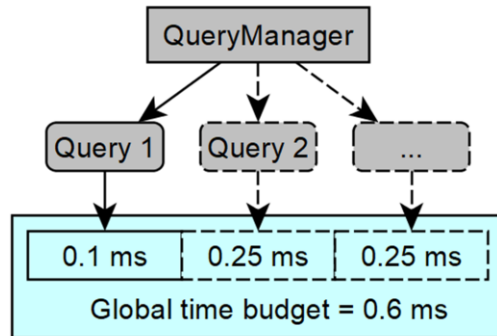
Item Monitors allow for detecting when something bad happens to the items that the query is reasoning about.

They can get installed by the Generator and will run in “parallel” (conceptually) alongside the query.

If an Item Monitor fails, then the whole query will fail.

Query Manager

- Centralized
- Global time budget
- Updates queries
- Queries get:
 - Time-sliced
 - Interrupted
- Donation of unused time



24

Works on a global time budget.

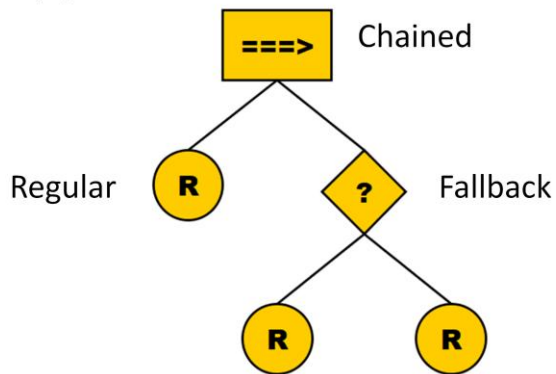
Starts and runs all queries, and notifies the caller (e.g. game code) of finished queries.

Time-slices queries and interrupts them whenever a query runs out of its granted time and resumes them on the next update cycle.

Unused time of a single query will get distributed to all the remaining ones.

Hierarchical Queries (1/4)


- BehaviorTree-like flow control
- 3 Query types



25

Hierarchical queries are similar to a Behavior Tree with composite nodes (queries) running their children in a nested way.

Hierarchical Queries (2/4)

- Regular Query 
 - “Action”
 - Generator
 - Evaluators



26

Regular queries can be compared to the “Action” nodes in a Behavior Tree. They do the actual work of generating items, evaluating them and then coming up with a result set.

The example from the beginning pretty much describes what a Regular query does: First, it generated potential locations on the NavMesh, then determined what the best location is by running evaluators on each of them and then returned the result.

Hierarchical Queries (3/4)

- Fallback Query



- “Selector”
- Tries children one after another

- Example

- 1st try: find closest visible enemy
- 2nd try: find closest enemy




27

Fallback queries can be compared to the “Selector” nodes in a Behavior Tree. They run one child after another. When one comes up with at least 1 item in its result set, this will be the overall result of the query and no further child queries will get executed.

Fallback queries can be used to relax some constraints in further child queries that would otherwise not be able to find items.

Hierarchical Queries (4/4)

- Chained Query 
 - “Sequence”
 - Runs all children
- Example:
 - 1st: generate points on NavMesh
 - 2nd: re-use generated points in remaining queries



28

Chained queries can be compared to the “Sequence” node in a Behavior Tree. They will run one child after another, not matter what the outcome of each child is. Chained queries can be used, e.g., to do some expensive computation in the first child, and then re-use the outcome in all remaining child queries.

Debugging (1/8)

- Motivation:
 - What was going on?
 - What went wrong?



29

In our example from the beginning, we might be interested to find out where locations in the world have been generated and which of them was considered the best one.

Also, we might be interested to know why certain locations have been discarded (or even encountered an error).

Debugging (2/8)

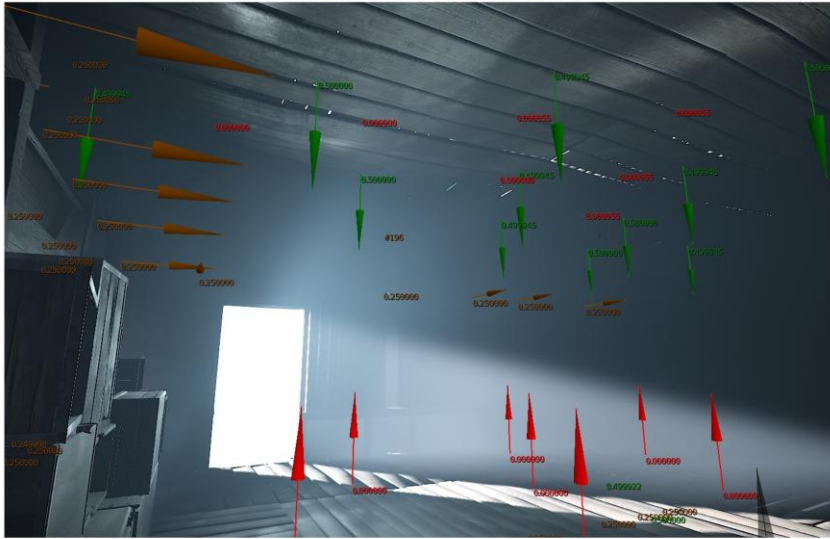
- Per Item type
 - Custom Visual Representation
 - Spheres
 - Lines
 - Arrows
 - ...



30

Upon registration of Item types, it's not only possible to specify their data type, but also what such objects should look like when being rendered for debugging purposes.

Debugging (3/8)

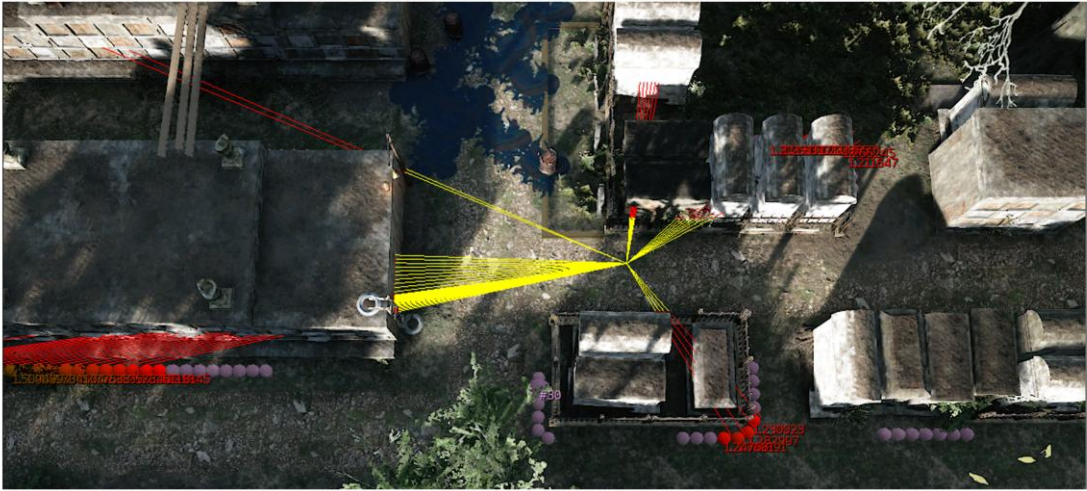


Visualization of points with direction



Here, we visualize points with directions as opposed to just points (for which spheres might be the representation of choice).

Debugging (4/8)



Potential spawn locations for enemies



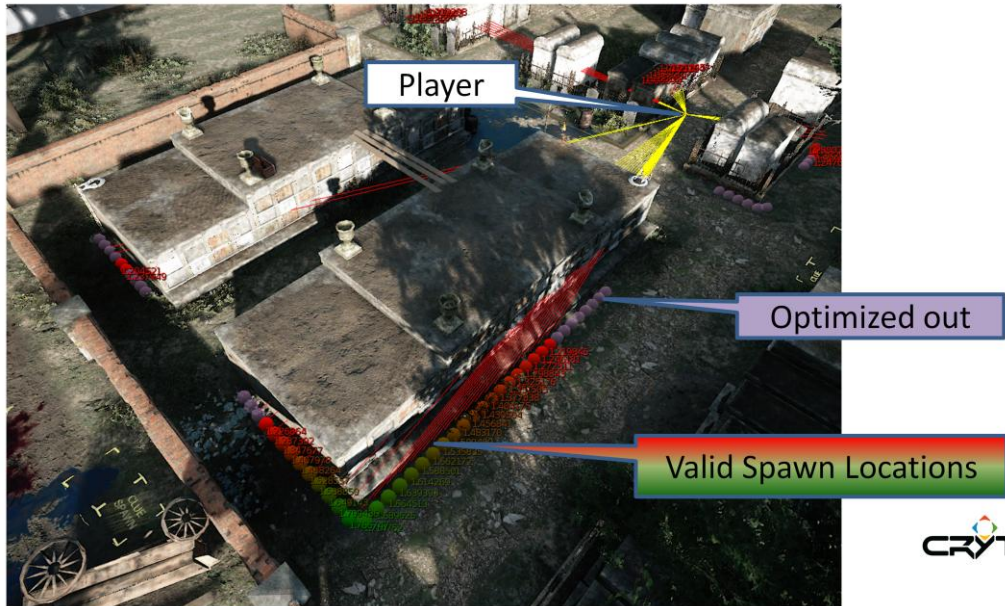
32

A scene from a system that figures out potential spawn locations for enemies dynamically.

Given the player's position, several qualities are being considered:

- (1) Spawn locations must have a minimum distance.
- (2) Player must not have a Line-of-Sight to them.
- (3) Prefer spawn locations furthest away from the player.

Debugging (5/8)



33

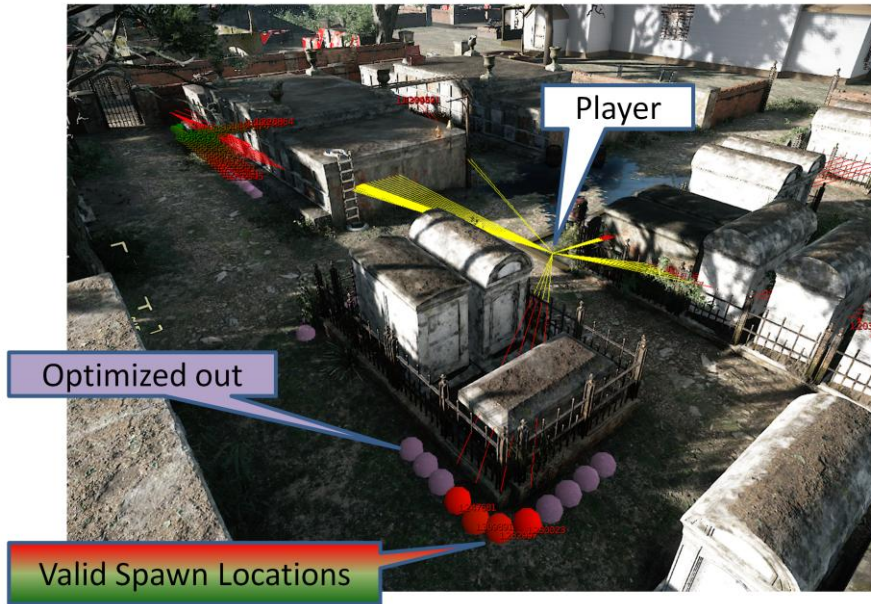
Same scene from a different angle:

At the top, we see the location of the player.

At the bottom, we have potential spawn locations of enemies, with color gradients indicating how good these locations are:

Green = good, red = bad, purple = these items couldn't improve anymore and got discarded prematurely before remaining evaluators had to run on them.

Debugging (6/8)



34

Same scene from a different angle.

The purple locations indicate that they couldn't improve anymore after the limit of 50 locations had been reached.

This means that not all of the remaining evaluators had to run on them – in particular doing raycasts for visibility checks was no longer needed as these items were already too bad.

Debugging (7/8)

- History
 - Record live data
 - Persist Debug Rendering
 - DebugRenderWorld
 - XML file dump
 - Debugging on a Server



35

Whenever a query starts running, it will also start recording a history with all relevant data and debug render primitives.

Histories can be saved to an XML file at any point in time.

This makes debugging on a server quite feasible – data can be recorded while the game is running and inspected at a later time on a different machine.

Debugging (8/8)

- History Inspector
 - Offline Tool
 - Read History XML file
 - Reconstruct Debug Rendering
 - Interactive analysis
 - Scores
 - Filtering
 - Exceptions
 - Quality Assurance 😊



36

The History Inspector is an offline tool that can read the recorded history and reconstruct all the debug rendering on a per-item basis.

It allows the user to get detailed information about individual items (e.g. how good an item scored, why it was discarded, what errors occurred, etc.)

This is also great for QA, since they can play the game, record all queries and later on hand the history to a programmer for tweaking and fixing individual queries.

History Inspector (1/2)

The screenshot shows the History Inspector application. On the left, a table lists various queries with columns for Query ID, Query Blueprint, Items (accepted), Elapsed time, and Timestamp. On the right, detailed information for a selected query (Query #939) is displayed, including elapsed seconds, consumed seconds, timestamp, and a breakdown of phases and frames.

Query ID	Query Blueprint	Items (accepted)	Elapsed time	Timestamp
#928: SLA...	specials/meat...	1 / 146	161.80 ms	895:02:46.726
#929: SLA...	specials/meat...	1 / 148	164.52 ms	895:02:47.014
#930: SLA...	specials/meat...	1 / 160	139.58 ms	895:02:47.700
#931: Default...	sandy_test	50 / 140	1869.72 ms	895:02:49.422
#932: SLA...	specials/meat...	1 / 161	158.71 ms	895:02:49.717
#933: SLA...	specials/meat...	1 / 161	136.34 ms	895:02:50.432
#934: SLA...	grunts/wander	2 / 59	13.66 ms	895:02:50.995
#935: SLA...	grunts/wander	4 / 61	129.88 ms	895:02:51.008
#936: SLA...	grunts/wander	0 / 61	296.37 ms	895:02:51.138
#937: SLA...	grunts/wander	0 / 58	282.82 ms	895:02:51.434
#938: SLA...	grunts/move	1 / 116	156.49 ms	895:02:51.845
#939: Default...	sandy_test	50 / 129	1487.86 ms	895:02:52.409
#940: SLA...	specials/meat...	1 / 161	142.24 ms	895:02:52.413
#941: SLA...	grunts/move	1 / 121	156.00 ms	895:02:53.742
#942: SLA...	specials/meat...	1 / 164	155.22 ms	895:02:54.052
#943: Default...	sandy_test	50 / 121	1472.38 ms	895:02:55.021
#944: Default...	sandy_test	50 / 126	2172.54 ms	895:02:57.712
#945: SLA...	grunts/wander	2 / 59	442.79 ms	895:02:57.717
#946: SLA...	specials/meat...	1 / 163	151.72 ms	895:02:57.854
#947: SLA...	grunts/wander	54 / 76	303.79 ms	895:02:58.145
#948: SLA...	grunts/wander	4 / 61	572.95 ms	895:02:58.160
#949: SLA...	grunts/wander	105 / 135	715.99 ms	895:02:58.448
#950: SLA...	specials/meat...	1 / 149	307.80 ms	895:02:58.565

=== Query #939 ===
Default / sandy_test
No exception
elapsed frames until result: 12
elapsed seconds until result: 1.452988 (1487.86 milliseconds)
consumed seconds: 0.002230 (2.23 milliseconds)
timestamp query created: 895:02:52.409
timestamp query destroyed: 895:02:53.896
canceled prematurely: NO
items desired: 50
items generated: 129
items still to inspect: 0
final items: 50
items memory: 2064 (2 kb)
items working data memory: 7224 (7 kb)
Instant-Evaluator #0: full runs = 129
Deferred-Evaluator #0: full runs = 50, aborted runs = 0
Phase '1' = 1 frames, 0.000120 sec (0.12 ms) [longest call = 0.000120 sec (0.12 ms)]
Phase '2' = 5 frames, 0.001440 sec (1.44 ms) [longest call = 0.000830 sec (0.83 ms)]
Phase '3' = 0 frames, 0.000070 sec (0.07 ms) [longest call = 0.000070 sec (0.07 ms)]
Phase '4' = 0 frames, 0.000040 sec (0.04 ms) [longest call = 0.000040 sec (0.04 ms)]
Phase '5' = 0 frames, 0.000020 sec (0.02 ms) [longest call = 0.000020 sec (0.02 ms)]
Phase '6' = 0 frames, 0.000010 sec (0.01 ms) [longest call = 0.000010 sec (0.01 ms)]
Phase '7' = 6 frames, 0.000480 sec (0.48 ms) [longest call = 0.000240 sec (0.24 ms)]

--- Item #99 ---
final score: 1.664513
disqualified due to bad score: no

IE #0: hunt.Director.ScorePointsOn? ☒
DE #0: std.Player.RaycastFromPlay ☒



History Inspector showing a list of past queries on the left side...
... and detailed information of a particular query on the right side.

History Inspector (2/2)

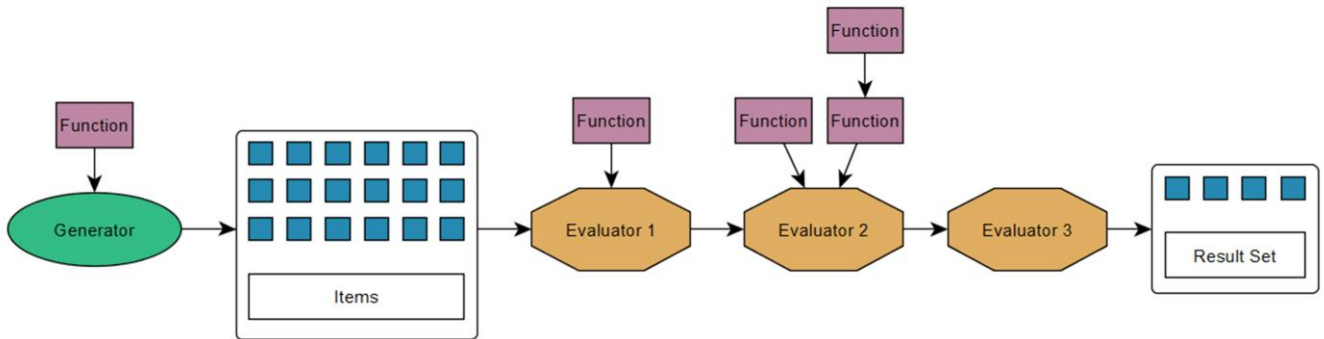


38

Sandbox editing mode:

Having loaded the history and pointing the camera at a specific item in the reconstructed debug rendering world will then reveal in-depth details about the outcome of that particular item.

Summary (1/2)



39

To wrap it up: we've learnt about the processing pipeline of a query, in particular about the 4 important elements: Items, Generator, Evaluators and Functions.

Summary (2/2)

- Customizable
- Complex scenarios
- Scalable
- Reusable by other systems
- Debugging Support



40

Considering the initial features that the query system was supposed to implement:

- (1) The system can be fully customized by the game for all 4 element types.
- (2) Complex problems can be solved by hierarchical queries.
- (3) The centralized Query Manager runs multiple queries at the same time and balances the available time among them.
- (4) The query system is generic enough to be usable by different systems as well rather than only by AI characters.
- (5) With its extensive debugging support, queries can be run and recorded on, e.g., dedicated servers and then debugged on a different machine at a later time.

Questions?

