Computable Logic Through Dynamic Measureless Connections

https://github.com/johnphantom/Dynamic-Stateless-Computer
or https://tinyurl.com/statelesscomputer

ABSTRACT

A computable logic, without measurements, that arises from how connections are made and/or broken over time – offering a unique topological implementation of a finite-state machine.

This paper presents a novel computational framework that implements logical operations through dynamic connection topology in constrained scripting environments, independent of numerical measurement or evaluation. We demonstrate that the Quakeworld engine's command architecture enables the construction of fundamental logical and arithmetic operations using only connection manipulation, requiring external operator intervention rather than automated execution.

INTRODUCTION

Conventional computation assumes that logical operations manipulate measurable symbolic or numerical states. This paper investigates an alternative: a measurement-free logic system in which information propagates solely through the topology of command connections. Using only three primitive operations—alias (command definition), bind (input mapping), and echo (output)—available within the Quakeworld scripting environment, we demonstrate that conditionals, loops, and bounded arithmetic can be implemented without explicit state variables or numerical evaluation. Though non-Turing complete and operator-dependent, this framework reveals that logical behavior can arise from connectivity alone, with implications for understanding computation independent of measurement and representation.

MATERIALS AND METHODS

Logic Implementation:

The computational models developed in this study employ the Quakeworld scripting engine and its derivatives, including Counter-Strike, as the experimental platform. The primary mechanism for logic implementation utilizes the "alias" system command, which enables the definition and redefinition of command sequences. A distinguishing feature of the proposed approach is that command strings execute exclusively through invocation of previously defined alias operations, creating self-referential execution chains that propagate information solely through the topology of established connections, independent of explicit value measurement or numerical state representation.

Input/Output Interface:

Logic state externalization is facilitated through two complementary system commands that serve as the model's interface layer. The "bind" command establishes deterministic mappings between physical input events—specifically, keyboard inputs—and corresponding computational procedures, thereby enabling user-directed manipulation of system state. The "echo" command provides the dual functions of state interrogation and output communication, rendering the current logical configuration as text to the console. Together, these commands constitute the complete input-output apparatus for the measurement-free logical system, maintaining the principle that all information propagation occurs through connection topology rather than explicit numerical evaluation.

Principal Arguments and Methods

- The methodology is based on scripting within game engines (Quakeworld/Counter-Strike), utilizing only three system commands:
    - `alias`: creates or redefines command sequences ("connections").
    - `bind`: links physical inputs to these connection scripts.
    - `echo`: outputs or communicates current logic states or results.
- The logic constructs include basic conditional branches (if-thens), looping structures (do-whiles), randomizers, rudimentary databases, and math emulators—demonstrated below through an abacus-like calculator that operates within system-imposed value limits (-20 to 20), mapping single-digit inputs to connection states without direct measurement.

RESULTS

Worked Example: Computing 5 + 5

Initialization: The script begins with the system linked to the `sumzero` state. This state establishes three links: `sum` → `sum0` (display value), `sumrotatedown` → `sumnegative1` (decrement), and `sumrotateup` → `sumpositive1` (increment).

Mode Selection: The + key is pressed, bound to the `plus` command. This redefines `rotate` to link to `sumrotateup`, establishing addition mode.

First Operand: The 5 key is pressed. This executes `calc5`, which performs five sequential `rotate` commands, each advancing the counter by one state. After five rotations through `sumpositive1` → `sumpositive2` → ... → `sumpositive5`, the system is now at the `sumpositive5` state. An `echo 5` command outputs the current input value.

Second Operand: The 5 key is pressed again. The `calc5` command executes again, performing five more `rotate` commands. Since `rotate` is still linked to `sumrotateup` from addition mode, the counter advances five more states: `sumpositive5` → `sumpositive6` → ... → `sumpositive10`. An `echo 5` command outputs the current input value.

Result: The Enter key is pressed, executing `sum` executing `sump10`, This outputs `echo` from the current state, which is `sumpositive10`, displaying "10" as the result.

The core mechanism: Each numeric operation is performed by traversing the state topology a fixed number of times. The mode determines the direction of traversal (increment or decrement). The final state identity encodes the result.

Text for calculator_simple.cfg

```
//Copyright All Rights Reserved 2025
//this script was built in Counter-Strike GO but will work in any QuakeWorld engine derivative game, including the original Quake.
//this script binds the KeyPad keys 0 through 9 along with - + and Enter
//This is limited to sums of -20 to 20, each number needs to be mapped and limits the system. This is also limited to single digit input.
//Starts in add mode. You can enter numbers in add or subtract mode without repeatedly hitting - or +
//list of system commands used (already built into the QuakeWorld engine):
//bind - input (for linking keys to commands - all of which in this example are user made commands)
```

```
//bind format: bind <key> <command>
//echo - output: for text output to the console (make sure you have the console turned on in settings, and use ~ in game to bring it down)
//echo format: echo <string of text>
//alias - the connection creator: for linking a string of commands, both user and system commands, together into one new callable user created command
//alias format: alias <command name> “<commands>”
//all of the rest of the commands in this particular script are user created, here, with alias

//input binds
//bind physical KeyPad numbers to adding/subtracting function
bind KP_END calc1
bind KP_DOWNARROW calc2
bind KP_PGDN calc3
bind KP_LEFTARROW calc4
bind KP_5 calc5
bind KP_RIGHTARROW calc6
bind KP_HOME calc7
bind KP_UPARROW calc8
bind KP_PGUP calc9
bind KP_INS calc0
bind KP_MINUS minus //bind physical KeyPad key - to the minus mode function
bind KP_PLUS plus //bind physical KeyPad key + to the plus mode function
bind KP_ENTER sum //bind physical KeyPad key Enter to output sum function

//performing of the logic - echo is the output

//set mode
alias plus "alias rotate sumrotateup; echo PLUS"
alias minus "alias rotate sumrotatedown; echo MINUS"

//increment or decrement based upon mode with input from a number key
alias calc0 "echo 0"
alias calc1 "rotate; echo 1"
alias calc2 "rotate; rotate; echo 2"
alias calc3 "rotate; rotate; rotate; echo 3"
alias calc4 "rotate; rotate; rotate; rotate; echo 4"
alias calc5 "rotate; rotate; rotate; rotate; rotate; echo 5"
alias calc6 "rotate; rotate; rotate; rotate; rotate; rotate; echo 6"
alias calc7 "rotate; rotate; rotate; rotate; rotate; rotate; rotate; echo 7"
alias calc8 "rotate; rotate; rotate; rotate; rotate; rotate; rotate; rotate; echo 8"
alias calc9 "rotate; rotate; rotate; rotate; rotate; rotate; rotate; rotate; rotate; echo 9"

//map of rotation
alias sumpositive20 "alias sum sump20; alias sumrotatedown sumpositive19; alias sumrotateup sumpositive20"
//sumpositive20 bounces back to itself in sumrotateup rather than going to an unmapped 21
alias sumpositive19 "alias sum sump19; alias sumrotatedown sumpositive18; alias sumrotateup sumpositive20"
alias sumpositive18 "alias sum sump18; alias sumrotatedown sumpositive17; alias sumrotateup sumpositive19"
alias sumpositive17 "alias sum sump17; alias sumrotatedown sumpositive16; alias sumrotateup sumpositive18"
alias sumpositive16 "alias sum sump16; alias sumrotatedown sumpositive15; alias sumrotateup sumpositive17"
alias sumpositive15 "alias sum sump15; alias sumrotatedown sumpositive14; alias sumrotateup sumpositive16"
alias sumpositive14 "alias sum sump14; alias sumrotatedown sumpositive13; alias sumrotateup sumpositive15"
alias sumpositive13 "alias sum sump13; alias sumrotatedown sumpositive12; alias sumrotateup sumpositive14"
alias sumpositive12 "alias sum sump12; alias sumrotatedown sumpositive11; alias sumrotateup sumpositive13"
alias sumpositive11 "alias sum sump11; alias sumrotatedown sumpositive10; alias sumrotateup sumpositive12"
alias sumpositive10 "alias sum sump10; alias sumrotatedown sumpositive9; alias sumrotateup sumpositive11"
alias sumpositive9 "alias sum sump9; alias sumrotatedown sumpositive8; alias sumrotateup sumpositive10"
alias sumpositive8 "alias sum sump8; alias sumrotatedown sumpositive7; alias sumrotateup sumpositive9"
```

```
alias sumpositive7 "alias sum sump7; alias sumrotatedown sumpositive6; alias sumrotateup sumpositive8"
alias sumpositive6 "alias sum sump6; alias sumrotatedown sumpositive5; alias sumrotateup sumpositive7"
alias sumpositive5 "alias sum sump5; alias sumrotatedown sumpositive4; alias sumrotateup sumpositive6"
alias sumpositive4 "alias sum sump4; alias sumrotatedown sumpositive3; alias sumrotateup sumpositive5"
alias sumpositive3 "alias sum sump3; alias sumrotatedown sumpositive2; alias sumrotateup sumpositive4"
alias sumpositive2 "alias sum sump2; alias sumrotatedown sumpositive1; alias sumrotateup sumpositive3"
alias sumpositive1 "alias sum sump1; alias sumrotatedown sumzero; alias sumrotateup sumpositive2"

alias sumzero "alias sum sum0; alias sumrotatedown sumnegative1; alias sumrotateup sumpositive1"

alias sumnegative1 "alias sum sumn1; alias sumrotatedown sumnegative2; alias sumrotateup sumzero"
alias sumnegative2 "alias sum sumn2; alias sumrotatedown sumnegative3; alias sumrotateup sumnegative1"
alias sumnegative3 "alias sum sumn3; alias sumrotatedown sumnegative4; alias sumrotateup sumnegative2"
alias sumnegative4 "alias sum sumn4; alias sumrotatedown sumnegative5; alias sumrotateup sumnegative3"
alias sumnegative5 "alias sum sumn5; alias sumrotatedown sumnegative6; alias sumrotateup sumnegative4"
alias sumnegative6 "alias sum sumn6; alias sumrotatedown sumnegative7; alias sumrotateup sumnegative5"
alias sumnegative7 "alias sum sumn7; alias sumrotatedown sumnegative8; alias sumrotateup sumnegative6"
alias sumnegative8 "alias sum sumn8; alias sumrotatedown sumnegative9; alias sumrotateup sumnegative7"
alias sumnegative9 "alias sum sumn9; alias sumrotatedown sumnegative10; alias sumrotateup sumnegative8"
alias sumnegative10 "alias sum sumn10; alias sumrotatedown sumnegative11; alias sumrotateup sumnegative9"
alias sumnegative11 "alias sum sumn11; alias sumrotatedown sumnegative12; alias sumrotateup sumnegative10"
alias sumnegative12 "alias sum sumn12; alias sumrotatedown sumnegative13; alias sumrotateup sumnegative11"
alias sumnegative13 "alias sum sumn13; alias sumrotatedown sumnegative14; alias sumrotateup sumnegative12"
alias sumnegative14 "alias sum sumn14; alias sumrotatedown sumnegative15; alias sumrotateup sumnegative13"
alias sumnegative15 "alias sum sumn15; alias sumrotatedown sumnegative16; alias sumrotateup sumnegative14"
alias sumnegative16 "alias sum sumn16; alias sumrotatedown sumnegative17; alias sumrotateup sumnegative15"
alias sumnegative17 "alias sum sumn17; alias sumrotatedown sumnegative18; alias sumrotateup sumnegative16"
alias sumnegative18 "alias sum sumn18; alias sumrotatedown sumnegative19; alias sumrotateup sumnegative17"
alias sumnegative19 "alias sum sumn19; alias sumrotatedown sumnegative20; alias sumrotateup sumnegative18"
alias sumnegative20 "alias sum sumn20; alias sumrotatedown sumnegative20; alias sumrotateup sumnegative19"
//sumnegative20 bounces back to itself in sumrotatedown rather than going to an unmapped 21

//end of logic

//initialize the system for first use
sumzero //initialize position of sum
plus //initialize for addition

//output echos
alias sump20 "echo sum 20"
alias sump19 "echo sum 19"
alias sump18 "echo sum 18"
alias sump17 "echo sum 17"
alias sump16 "echo sum 16"
alias sump15 "echo sum 15"
alias sump14 "echo sum 14"
alias sump13 "echo sum 13"
alias sump12 "echo sum 12"
alias sump11 "echo sum 11"
alias sump10 "echo sum 10"
alias sump9 "echo sum 9"
alias sump8 "echo sum 8"
alias sump7 "echo sum 7"
alias sump6 "echo sum 6"
alias sump5 "echo sum 5"
alias sump4 "echo sum 4"
alias sump3 "echo sum 3"
alias sump2 "echo sum 2"
```

```
alias sump1 "echo sum 1"
alias sum0 "echo sum 0"
alias sumn1 "echo sum -1"
alias sumn2 "echo sum -2"
alias sumn3 "echo sum -3"
alias sumn4 "echo sum -4"
alias sumn5 "echo sum -5"
alias sumn6 "echo sum -6"
alias sumn7 "echo sum -7"
alias sumn8 "echo sum -8"
alias sumn9 "echo sum -9"
alias sumn10 "echo sum -10"
alias sumn11 "echo sum -11"
alias sumn12 "echo sum -12"
alias sumn13 "echo sum -13"
alias sumn14 "echo sum -14"
alias sumn15 "echo sum -15"
alias sumn16 "echo sum -16"
alias sumn17 "echo sum -17"
alias sumn18 "echo sum -18"
alias sumn19 "echo sum -19"
alias sumn20 "echo sum -20"
```

End of calculator_simple.cfg

DISCUSSION

The logic system presented in this paper—termed "Logic Geometry"—is a computable logic that arises solely from how connections are made and/or broken over time, without the use of measurements or explicit numerical evaluation. This approach is distinct from traditional computational models such as Lambda Calculus or classical logic gates, which rely on measurable states and symbolic manipulation. Instead, Logic Geometry operates through the dynamic topology of connections, where the logic is encoded in the shape and evolution of the network itself.

The core mechanism of this system is the "alias" command, which allows for the creation and redefinition of command sequences, forming self-referential execution chains that propagate information exclusively through the topology of established connections. The alias command functions primarily as a convenient vehicle for emulation, enabling the construction of conditionals, loops, randomizers, relational databases, and math emulators, all without relying on explicit value measurement or state variables. Importantly, alias is not essential to the logic itself: an external "operator" could manually alter the connections and still achieve computable logic, showing that the logical behavior arises from connection topology rather than the use of alias specifically.

This minimalist design enables a wide range of logical constructs but is non-Turing complete, requiring an external operator to manipulate connections and provide input, which distinguishes it from general-purpose programming languages.

A key insight is that the logic is not defined by the values or symbols being processed, but by the structure and transitions of the connections themselves. This can be likened to a dynamic truth table, where the "truths" change as the system runs, reflecting the evolving state of the connection network. The shape of the logic is the logic, and the logic is the shape—a principle that highlights the abstract, relational nature of this computational framework.

This approach reveals that logical behavior can emerge from connectivity alone, with implications for understanding computation independent of measurement and representation. The system demonstrates that complex logic can be achieved through minimal primitives, focusing on relationships between states rather than specific values or measurements. While the system is not Turing complete, it provides a novel paradigm for computation that blurs the lines between logic, systems theory, and abstract philosophy.

Logic Geometry demonstrates that logical behavior can arise from connection topology alone, without measurement, symbols, or explicit state variables, fundamentally challenging the foundational assumption that computation requires a distinction between structure (the system's form) and content (the values or symbols the system processes). In classical logic and computation theory, information is encoded in measurable states—Boolean values, numerical registers, symbolic tokens—and the relationship between states. Logic Geometry inverts this: the connection configuration itself is simultaneously the structure, the content, and the state. There is no distinction between "the system" and "what the system represents"—the shape of connections is the information; the logic is the topology. This implies that logic may not be an abstract formalism imposed on reality by human minds, but rather an intrinsic property of how relationships and connections organize themselves. When connections reconfigure, logic unfolds. When topology shifts, truth-values change—but not because we're measuring something external; rather, because the topology itself is the truth-value. This raises a profound question: Could logic be fundamental to reality in the way space and time are? If so, then universes might compute themselves into existence through pure topological reconfiguration, with no need for external laws, numbers, or measurement. Causation, information, and logical inference could all be emergent from relational structure, suggesting that the deepest nature of reality is not material (stuff) or mathematical (numbers), but relational—a universe of connections computing itself through their own topology.

In summary, Logic Geometry offers a unique perspective on computable logic, emphasizing the role of connection topology and state transitions over traditional notions of measurement and symbolic manipulation. This framework not only expands the boundaries of what is considered computable but also invites further exploration into the nature of logic and computation in measurement-free environments.

RELATED WORK

Comparison to Membrane Computing:

Logic Geometry shares membrane computing's emphasis on structural topology as the computational substrate rather than symbolic manipulation, but differs fundamentally in its primitives and abstraction level. Membrane computing (P systems) encodes computation through hierarchical membrane structures, object multisets, and rewriting rules that govern how objects move between membranes or how membranes divide and dissolve—the membrane configuration and its evolution define the computation. Logic Geometry, by contrast, operates at a more minimal level: computation arises solely from dynamic connection redefinition (via alias) without objects, tokens, or hierarchical containers. Where membrane systems use membranes as boundaries that regulate object flow, Logic Geometry uses connection topology itself as both structure and information, with no distinction between "container" and "content"—the shape of the command graph is the state. Both frameworks demonstrate that computation can emerge from relational structure rather than value measurement, but membrane computing retains symbolic objects and rewriting rules, whereas Logic Geometry eliminates symbols entirely, making it a more radically measureless and connection-pure model.

Comparison to Chemical Reaction Networks:

Chemical Reaction Networks (CRNs) model computation through molecular species concentrations and reaction rules that transform one set of molecular counts into another, with the network topology (which species react to produce which products) determining computational behavior. Like Logic Geometry, CRNs demonstrate that computation can arise from structural connectivity—the graph of possible reactions—rather than explicit symbolic manipulation. However, CRNs fundamentally rely on quantitative measurements: molecular concentrations, reaction rates, and stoichiometric coefficients are central to their computational semantics, whether in deterministic (mass-action kinetics) or stochastic (Gillespie algorithm) implementations. Logic Geometry differs by eliminating measurement entirely—there are no concentrations, no rate constants, no numerical values of any kind. Where a CRN computes by tracking how many molecules of each species exist and evolving those counts according to reaction rules, Logic Geometry computes by redefining which commands connect to which other commands, with the connection topology itself (not any measured quantity) encoding state. Both frameworks show that network structure can be computational, but CRNs are measurement-based (concentration dynamics), while Logic Geometry is purely topological (connection reconfiguration without quantities).

Comparison to Petri Nets:

Petri nets represent computation as the movement and transformation of tokens through a directed graph of places and transitions, where token positions encode state and transition firings (governed by enabling rules) define state evolution. Like Logic Geometry, Petri nets abstract computation away from symbolic manipulation and ground it in graph structure and state transitions—the topology of places, transitions, and arcs determines what computations are possible. However, Petri nets retain explicit state representation via token counts: each place holds a discrete number of tokens, and computation proceeds by decrementing tokens from input places and incrementing tokens in output places according to transition rules. Logic Geometry eliminates this token-based state representation entirely. Where a Petri net computes by tracking token distribution across places and firing transitions when input conditions are met, Logic Geometry computes by dynamically redefining command connections without any intermediate state representation—the current connection configuration is the entire state, with no tokens, counts, or discrete objects to track. Both frameworks demonstrate that graph topology and state transitions can substrate computation, but Petri nets are token-based (discrete object movement), while Logic Geometry is purely relational (connection redefinition with no objects or measurements).

Comparison to Graph Rewriting Systems:

Graph rewriting systems compute by applying transformation rules to graph structures, where each rule specifies a pattern to match in the current graph and a replacement subgraph to substitute—computation proceeds as a sequence of graph rewrites until a normal form or terminal state is reached. Like Logic Geometry, graph rewriting grounds computation in structural transformation rather than value evaluation, and the graph topology itself (which nodes and edges exist, how they connect) is the primary computational object. However, graph rewriting systems typically operate on explicit graph nodes and edges as discrete entities, and the rewrite rules must specify pattern matching (which nodes match the rule) and substitution (how to replace matched subgraphs)—this requires either symbolic node labels or numerical node identifiers to distinguish nodes and apply rules correctly. Logic Geometry differs by eliminating discrete entities entirely: there are no nodes or edges to label, no pattern matching on symbols, no replacement operations on subgraphs. Instead, computation arises

from direct redefinition of connections (via alias commands), where the current connection configuration is simultaneously the state, the rule executor, and the transition mechanism—a single unified topology that evolves without reference to explicit symbols or intermediate representations. Both frameworks show that graph structure can be computational, but graph rewriting requires symbolic or numerical entity labels to enable pattern matching and substitution, while Logic Geometry is purely relational and unmarked—the shape and reconfiguration of connections alone suffice.

ACKNOWLEDGEMENTS

This research owes its foundational impetus to the scripting capabilities embedded within the Quake engine, designed by John Carmack, which enabled the systematic exploration of measurement-free logic systems.