Computable Logic Through Dynamic Measureless Connections

https://github.com/johnphantom/Dynamic-Stateless-Computer

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 #1: 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 2026
//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

Worked Example #2: Throwing a smoke grenade then throwing a high explosive grenade

This word problem illustrates what this script accomplishes: You have a combination padlock with four dials on it. Each dial has the numbers 0 through 4 on them. The lock can have as many 0s as dials, and is set to 0000 by default. The lock does not allow you to use any number between 1 and 4 two or more times in the combination. The following combinations are valid: 0123 1234 0103 0010 4031. The following combinations are invalid: 0113 4014 0202 4444. How many possible combinations are there?

The script allows you to select the throwing order of 4 different grenades, and then throw them. The 0 in the combination lock represents an empty slot, which when throwing will be skipped over. The first slot button you push will take high explosive, the second slot button you push even if it is the same button as the high explosive grenade will put a flash bang in the slot, then teargas, then smoke. Pressing any slot a 5th time will reset the slots.

These formulas are answers to the word problem, where `n=4` and the result of 209 possible permutations:

$$\sum_{k=0}^{n} C(n,k)P(n,n-k) \equiv \sum_{k=0}^{n-1} C(n-1,k)P(n,n-k)(n-k+1).$$

This can also be calculated using Algebra and Pascal's Triangle in a new way, unconventionally accounting for zero. If scaled up to 18 items and 18 slots, there are approximately **2.97×10^{18}** possible permutations, each reachable within 18 keystrokes - the access depth scales linearly with *n*, not exponentially.

Initialization: A plus sign (+) in front of a command means that command is executed when the key is pressed down, the minus sign (-) in front of a command means that command is executed when the key is released.

The script begins with the system linked to `grnrt0`. This executes `grst1` and executes an "echo FULL RESET of Grenade Slots". The command `grst1` links `grnsltа` through `grnsltd` each incrementally to `grnrta1` through `grnrtd1` setting up the selection of grenades. The `gsrst` command, the `gbrst` command, and `skiprst` command are executed.

The `gsrst` command links the the grenade slot place holders `+gs1` to `+gs4` to the null command with their release action of `-gs1` to `-gs4` to `gbutrot`. The `gbutrot` command causes the rotation of the grenades to be thrown.

The `gbrst` command links `gbutrot` to the first grenade throw slot linker, gbutrot**1** and `+gtoss` is set to `+gtall` and `-gtoss` is set to `-gtall` to default throw any grenade you have.

The command `skprst` links `skip1` through `skip4` to `skip1rst` through `skip4rst`

The commands `gbutrot1` through `gbutrot4` increment the current `+gtoss` command to `+gs1` through `+gs4` slot placeholders and `-gtoss` command to `-gs1` through `-gs4` slot placeholders, with `gbutrot` linked to the next `gbutrot2` through `gbutrot4` with a `skip1` through `skip4` to jump over empty slots as the user is throwing or halt rotation for a selected grenade throw.

The commands `skip1rst` though `skip4rst` work with `gbutrot1` through `gbutrot4` incrementally with the `skip1` through `skip4` commands to rotate grenades for throwing. The commands `skip1rst` through `skip4rst` incrementally set the `+gs1` through `+gs4` slot placeholders for grenades to `+gtoss` and `-gtoss`, then sets `gbutrot` to the next `gbutrot3` or `gbutrot4` or `skip4rst`, and then executes `skip2` through `skip4` to skip empty slots or halt at that slot for throwing. The command `skip4rst` resets everything to throw any grenade you have.

Grenade Selection:

The commands `grnslta1` through `grnsltd4` incrementally execute `grst2` through `grst5` incrementing the commands `grnsltа` through `grnsltd` connected to operator keys to only use each type of grenade once. The commands `grnslta1` through `grnsltd4` also link the `+gs1` and `-gs1` through `+gs4` and `-gs4` commands to which specific type of grenade to throw; `+gthe` and `-gthe` (high explosive), `+gtfb` and `-gtfb` (flash bang), `+gttg` and `-gttg` (tear gas), or `+gtsmk` and `-gtsmk` (smoke grenade). The commands `grnslta1` through `grnsltd4` also link the corresponding `skip1` through `skip4` to execute the command `null`, which stops the execution of the loop.

First the V key is pressed to put a high explosive in slot 3 through executing `grnsltc` which executes `grnsltc1`. This links `+gthe` and `-gthe` to `+gs3` and `-gs3`, and `skip3` to `null` so the script stops to throw the grenade.

The X key is pressed once to put `+gtfb` and `-gtfb` in `+gs1` and `-gs1`, then the X key is pressed again to put `+gttg` and `-gttg` in `+gs1` and `-gs1`. Once more, press the X key to put `+gtsmk` and `-gtsmk` in `+gs1` and `-gs1`, and `skip1` to `null` so the script stops to throw the grenade.

The system is now set to throw a smoke grenade first, skip the second slot, throw the high explosive second, skip the fourth and last slot, then go to throwing any grenade you might have.

Throwing Mechanics:

The command gbutrot executes gbutrot**1** through gbutrot**4**, which incrementally link to the grenade slots +gs**1** through +gs**4** and execute a corresponding skip**1** through skip**4.**

The Mouse2 key is pressed executing +gtoss which executes +gs**1** which executes +gtsmk. The Mouse2 key is released, −gs**1** executes gbutrot which executes gbutrot**2** to set up the next Mouse2 press. The commands +gs**2** and −gs**2** are skipped to the third slot executing skip**2** which executes skip**2**rst. The Mouse2 key is pressed executing +gtoss which executes +gs**3** which executes +gthe. When the Mouse2 key is released, -gs**3** executes gbutrot which executes gbutrot**4**. The command +gs**4** and −gs**4** are skipped by executing skip**4** which executes skip**4**rst that sets the Mouse2 grenade throwing to throw any grenade available.

Start of grenade-combination_lock.cfg

```
//Copyright 2026 All Rights Reserved
//this script was built in Counter-Strike GO but will work in any QuakeWorld engine derivative game, including the original Quake.
//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

//bind keys to commands for user input
bind x grnslta //bind the X key to grenade slot 1
bind c grnsltb //bind the C key to grenade slot 2
bind v grnsltc //bind the V key to grenade slot 3
bind b grnsltd //bind the B key to grenade slot 4
bind n grnrt0 //bind the N key to reset the grenade slots
bind MOUSE2 +gtoss //bind mouse2 to the action of throwing the grenades

// Grenade Throwing System
alias null ""
alias gsrst "alias +gs1 null;alias -gs1 gbutrot;alias +gs2 null;alias -gs2 gbutrot;alias +gs3 null;alias -gs3 gbutrot;alias +gs4 null;alias -gs4 gbutrot"
alias gbrst "alias gbutrot gbutrot1;alias +gtoss +gtall;alias -gtoss -gtall"
alias skiprst "alias skip1 skip1rst;alias skip2 skip2rst;alias skip3 skip3rst;alias skip4 skip4rst"

alias skip1rst "alias +gtoss +gs2;alias -gtoss -gs2;alias gbutrot gbutrot3;skip2"
alias skip2rst "alias +gtoss +gs3;alias -gtoss -gs3;alias gbutrot gbutrot4;skip3"
alias skip3rst "alias +gtoss +gs4;alias -gtoss -gs4;alias gbutrot skip4rst;skip4"
alias skip4rst "alias +gtoss +gtall;alias -gtoss -gtall;alias gbutrot null"

alias grst1 "alias grnslta grnrta1;alias grnsltb grnrtb1;alias grnsltc grnrtc1;alias grnsltd grnrtd1;gsrst;gbrst;skiprst"
alias grst2 "alias grnslta grnrta2;alias grnsltb grnrtb2;alias grnsltc grnrtc2;alias grnsltd grnrtd2"
alias grst3 "alias grnslta grnrta3;alias grnsltb grnrtb3;alias grnsltc grnrtc3;alias grnsltd grnrtd3"
alias grst4 "alias grnslta grnrta4;alias grnsltb grnrtb4;alias grnsltc grnrtc4;alias grnsltd grnrtd4"
alias grst5 "alias grnslta grnrt0;alias grnsltb grnrt0;alias grnsltc grnrt0;alias grnsltd grnrt0"

alias grnrt0 "grst1;echo FULL RESET of Grenade Slots"
alias grnrta1 "grst2;alias +gs1 +gthe;alias -gs1 -gthe;alias skip1 null;echo Grenade Slot 1 high explosive grenade"
```

```
alias grnrta2 "grst3;alias +gs1 +gtfb;alias -gs1 -gtfb;alias skip1 null;echo Grenade Slot 1 flashbang grenade"
alias grnrta3 "grst4;alias +gs1 +gttg;alias -gs1 -gttg;alias skip1 null;echo Grenade Slot 1 teargas grenade"
alias grnrta4 "grst5;alias +gs1 +gtsmk;alias -gs1 -gtsmk;alias skip1 null;echo Grenade Slot 1 smoke grenade"
alias grnrtb1 "grst2;alias +gs2 +gthe;alias -gs2 -gthe;alias skip2 null;echo Grenade Slot 2 high explosive grenade"
alias grnrtb2 "grst3;alias +gs2 +gtfb;alias -gs2 -gtfb;alias skip2 null;echo Grenade Slot 2 flashbang grenade"
alias grnrtb3 "grst4;alias +gs2 +gttg;alias -gs2 -gttg;alias skip2 null;echo Grenade Slot 2 teargas grenade"
alias grnrtb4 "grst5;alias +gs2 +gtsmk;alias -gs2 -gtsmk;alias skip2 null;echo Grenade Slot 2 smoke grenade"
alias grnrtc1 "grst2;alias +gs3 +gthe;alias -gs3 -gthe;alias skip3 null;echo Grenade Slot 3 high explosive grenade"
alias grnrtc2 "grst3;alias +gs3 +gtfb;alias -gs3 -gtfb;alias skip3 null;echo Grenade Slot 3 flashbang grenade"
alias grnrtc3 "grst4;alias +gs3 +gttg;alias -gs3 -gttg;alias skip3 null;echo Grenade Slot 3 teargas grenade"
alias grnrtc4 "grst5;alias +gs3 +gtsmk;alias -gs3 -gtsmk;alias skip3 null;echo Grenade Slot 3 smoke grenade"
alias grnrtd1 "grst2;alias +gs4 +gthe;alias -gs4 -gthe;alias skip4 null;echo Grenade Slot 4 high explosive grenade"
alias grnrtd2 "grst3;alias +gs4 +gtfb;alias -gs4 -gtfb;alias skip4 null;echo Grenade Slot 4 flashbang grenade"
alias grnrtd3 "grst4;alias +gs4 +gttg;alias -gs4 -gttg;alias skip4 null;echo Grenade Slot 4 teargas grenade"
alias grnrtd4 "grst5;alias +gs4 +gtsmk;alias -gs4 -gtsmk;alias skip4 null;echo Grenade Slot 4 smoke grenade"

alias gbutrot1 "alias +gtoss +gs1;alias -gtoss -gs1;alias gbutrot gbutrot2;skip1"
alias gbutrot2 "alias +gtoss +gs2;alias -gtoss -gs2;alias gbutrot gbutrot3;skip2"
alias gbutrot3 "alias +gtoss +gs3;alias -gtoss -gs3;alias gbutrot gbutrot4;skip3"
alias gbutrot4 "alias +gtoss +gs4;alias -gtoss -gs4;alias gbutrot skip4rst;skip4"

alias +gthe "echo throw high explosive grenade"
alias -gthe "gbutrot"
alias +gtfb "echo throw flashbang grenade"
alias -gtfb "gbutrot"
alias +gttg "echo throw teargas grenade"
alias -gttg "gbutrot"
alias +gtsmk "echo throw smoke grenade"
alias -gtsmk "gbutrot"
alias +gtall "echo throw any available grenade"
alias -gtall "null"

//initialize the system
grnrt0
```

End of grenade-combination_lock.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 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.

Formal System

This paper presents constructive examples and a preliminary framework. Full formalization and universality proofs are left to future work.

Part 1: Formal System Definition

1.1 Basic Objects

Definition 1.1 (Connection Configuration)
A *connection configuration* C is a finite directed graph (N,E) where:

- N is a finite set of *nodes* (command names)
- E ⊆N × N is a set of *directed edges* (connections)
- Each node n ∈N has an associated *command body* body(n) ⊆N, specifying which nodes it invokes when executed

Definition 1.2 (Primitive Operations)
Three primitive operations modify configurations:

1. **DEFINE(n, B)**: Create or redefine node n with body B ⊆N
    - Effect: body(n) := B; adds edges (n,b) for all b ∈B
2. **BIND(i, n)**: Associate input i with node n
    - Effect: External input i triggers execution of n
3. **ECHO(n)**: Output the name of node n
    - Effect: Produces observable output (not modifying topology)

Definition 1.3 (Execution Semantics)
Given configuration C and operator input i:

- If BIND(i, n) holds, execute n:
    - For each m ∈body(n), execute m (depth-first, left-to-right)
    - If mm is DEFINE, modify C accordingly
    - If m is ECHO, produce output
    - Execution continues until all reachable commands complete

Definition 1.4 (Computation)
A *Logic Geometry computation* is a sequence:

$$C_0 \xrightarrow{i_1} C_1 \xrightarrow{i_2} C_2 \xrightarrow{i_3} \cdots \xrightarrow{i_k} C_k$$

where:

- C_0 is initial configuration
- Each i_j is operator input
- Each C_j+1 results from executing the command bound to i_j in C_j

---

1.2 State Encoding

Theorem 1.1 (Topology as State)
Claim: In Logic Geometry, the entire computational state is encoded in the connection topology; there are no hidden variables or auxiliary storage.

Proof sketch:

- The only persistent information is which nodes exist and which edges connect them.
- Execution is deterministic given C and input sequence.
- No numerical registers, symbol tables, or stack frames exist.
- The configuration C itself is program, data, and state simultaneously.

Corollary 1.1
Two configurations C and C′ are computationally equivalent if they are graph-isomorphic up to node renaming.

1.3 Expressiveness

Definition 1.5 (Logic Primitives)
We say Logic Geometry *implements* a logical primitive if there exists a configuration and input sequence producing equivalent behavior:

1. **Conditional**: Configuration that executes different nodes based on prior topology.
2. **Loop**: Configuration that cycles through states until operator halts.
3. **Boolean operations**: Configurations implementing AND, OR, NOT via topology.

Lemma 1.1 (Conditionals)
Logic Geometry can implement conditionals.

Construction (calculator_simple.cfg):

- Node `rotate` is redefined to point to either `sumrotateup` or `sumrotatedown`.
- Subsequent calls to `rotate` execute different branches.
- This is an if-then based purely on which edge currently exists.

Lemma 1.2 (Bounded Iteration)
Logic Geometry can implement loops with operator-determined termination.

Construction (grenade script):

- Node `gbutrot` cycles through `gbutrot1 → gbutrot2 → gbutrot3 → gbutrot4`.
- Each cycle redefines which node is next.
- Operator input triggers each iteration.
- Loop continues until operator stops providing input.

Lemma 1.3 (Arithmetic)
Logic Geometry can implement bounded arithmetic.

Construction (calculator example):

- Configuration encodes number line as a bounded topology: `sumnegative20 → … → sum0 → … → sumpositive20`.
- Addition = traversing upward kk steps.

- Subtraction = traversing downward kk steps.
- Current position encodes result.

Part 2: Comparison to Standard Models

2.1 Relationship to Finite State Machines

Theorem 2.1 (FSM Representation)
Claim: Every deterministic finite automaton (DFA) can be represented as a Logic Geometry configuration.

Proof strategy:

1. Let $M=(Q, \Sigma, \delta, q_0, F)$ be a DFA.
2. Construct configuration C_M:
    - Create node n_q for each state $q \in Q$.
    - For each q, define body(n_q) to contain nodes implementing state transitions.
    - For each input symbol $\sigma \in \Sigma$, create node `input_`σ that redefines "current" to point to $n\delta(q,\sigma)$.
    - Accepting states trigger ECHO("accept").
3. Show that input sequence $w \in \Sigma^*$ is accepted by MM if the corresponding operator inputs to C_M produce "accept" output.
4. Conclude that DFAs embed into Logic Geometry.

Corollary 2.1
Logic Geometry is at least as expressive as regular languages.

2.2 Relationship to Turing Machines

Theorem 2.2 (Non-Turing Completeness)
Claim: Logic Geometry as defined is not Turing-complete.

Proof:

1. Every Logic Geometry configuration has finite N (finite nodes).
2. Topology changes are operator-dependent (no autonomous infinite execution).
3. No unbounded tape or infinite workspace exists.
4. Therefore, Logic Geometry cannot simulate arbitrary Turing machines.
5. The model is equivalent to operator-interactive finite-state systems.

This locates Logic Geometry as a distinct computational class rather than a universal programming language.

2.3 Relationship to Lambda Calculus

Theorem 2.3 (Lambda Calculus Embedding – Restricted)
Claim: Fragments of lambda calculus (specifically, combinatory logic with bounded reduction) can be embedded in Logic Geometry.

Proof sketch:

1. Represent each combinator (S, K, I) as a node with specific connection patterns.
2. Beta-reduction becomes topological reconfiguration:
    - Application of K to two arguments: redefine connections to eliminate one branch.
    - Application of S: redefine to create duplication pattern.
3. Show that reduction sequences in combinatory logic correspond to operator-driven reconfigurations in Logic Geometry.
4. Limitation: unbounded recursion requires unbounded operator input.

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. [1, 2, 3]

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). [4, 5, 6]

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). [7, 8, 9]

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. [10, 11, 12]

Comparison to Lambda Calculus:

Lambda Calculus formalizes computation as the application and reduction of symbolic function expressions, where variables, abstractions, and applications are manipulated according to syntactic rewrite rules; computation is defined by beta-reduction on symbolic terms in an unbounded, abstract

space of expressions. Like Logic Geometry, Lambda Calculus provides a minimal, foundational model of computation, but it remains inherently symbolic and value-centric: terms have structure independent of the reduction process, and the semantics of computation rely on substitution of variable bindings and evaluation of expression structure. Logic Geometry differs by eliminating variables, symbols, and substitution entirely—there are no expressions to rewrite, no bound variables, no environments. Where Lambda Calculus computes by rewriting expressions according to syntactic rules, Logic Geometry computes by reconfiguring a network of connections, with the current connection topology simultaneously serving as program, data, and state. Both frameworks aim at extreme minimality, but Lambda Calculus is axiomatically algebraic and symbolic, whereas Logic Geometry is purely topological and relational, suggesting a complementary foundation for computation that does not presuppose symbols, measurement, or substitution at all. [13, 14, 15]

Framework	Substrate	Basic Unit	Information Encoding	Requires Measurement?	Requires Symbols?
Membrane Computing	Hierarchical membranes	Objects/multisets	Object counts & membrane structure	Yes (implicitly)	Yes (object types)
Chemical Reaction Networks	Molecular species	Molecules	Concentration dynamics	Yes (essential)	Yes (species)
Petri Nets	Places & transitions	Tokens	Token distribution	No	Partially (labels)
Graph Rewriting	Graph nodes & edges	Subgraphs	Node/edge structure	No	Yes (node labels)
Lambda Calculus	Abstract expression space	Lambda terms	Symbolic term structure and substitutions	No	Yes (variables, symbols)
Logic Geometry	**Connection topology**	**Connections**	**Topology shape alone**	**No**	**No**

Any attempt to recast Logic Geometry back into conventional formalisms (e.g., via state transition graphs or algebraic specifications) is possible in principle, but would miss the point: what is being exhibited here is a computing behavior whose native description is purely topological and operational, not symbolic.

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. [16, 17]

REFERENCES

1. Păun, G. (2000). Computing with membranes. *Journal of Computer and System Sciences*
2. Păun, G. (2002). Membrane computing: An introduction. *Springer-Verlag*, Berlin.
3. Păun, G., Rozenberg, G., & Salomaa, A. (Eds.). (2010). *The Oxford Handbook of Membrane Computing*. Oxford University Press.

4. Soloveichik, D., Cook, M., Winfree, E., & Bruck, J. (2008). Computation with finite stochastic chemical reaction networks. *Natural Computing*
5. Cardelli, L., & Csikász-Nagy, A. (2012). The cell cycle switch computes approximate majority. *Scientific Reports*
6. Chen, Y.-J., Dalchau, N., Srinivas, N., Phillips, A., Cardelli, L., Soloveichik, D., & Seelig, G. (2013). Programmable chemical controllers made from DNA. *Nature Nanotechnology*

7. Petri, C. A. (1962). *Kommunikation mit Automaten* [Communication with automata]. PhD thesis, Universität Bonn, Germany
8. Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*
9. Reisig, W., & Rozenberg, G. (Eds.). (1998). *Lectures on Petri Nets I: Basic Models*. Springer-Verlag, Berlin. Lecture Notes in Computer Science, Vol. 1491.

10. Ehrig, H., Pfender, M., & Schneider, H. J. (1973). Graph-grammars: An algebraic approach. In *14th Annual Symposium on Switching and Automata Theory*. IEEE.
11. Plump, D. (1999). Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, & G. Rozenberg (Eds.), *Handbook of Graph Grammars and Computing by Graph Transformation* (Vol. 2,). World Scientific.
12. Kreowski, H.-J., & Kuske, S. (1999). Graph transformation units with interleaving semantics. *Formal Aspects of Computing*,

13. Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*

14. Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics* (2nd ed.). North-Holland.

15. Cardelli, L. (2004). Type systems. In J. van Leeuwen, G. Rozenberg, & A. Salomaa (Eds.), *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier.

16. Abrash, M. (1997). Michael Abrash's graphics programming black book, special edition. Coriolis Group Books.
17. Carmack, J. (1996). Quake engine source code and design notes. id Software. (Released under GPL in 1999; archived at https://github.com/id-Software/Quake)