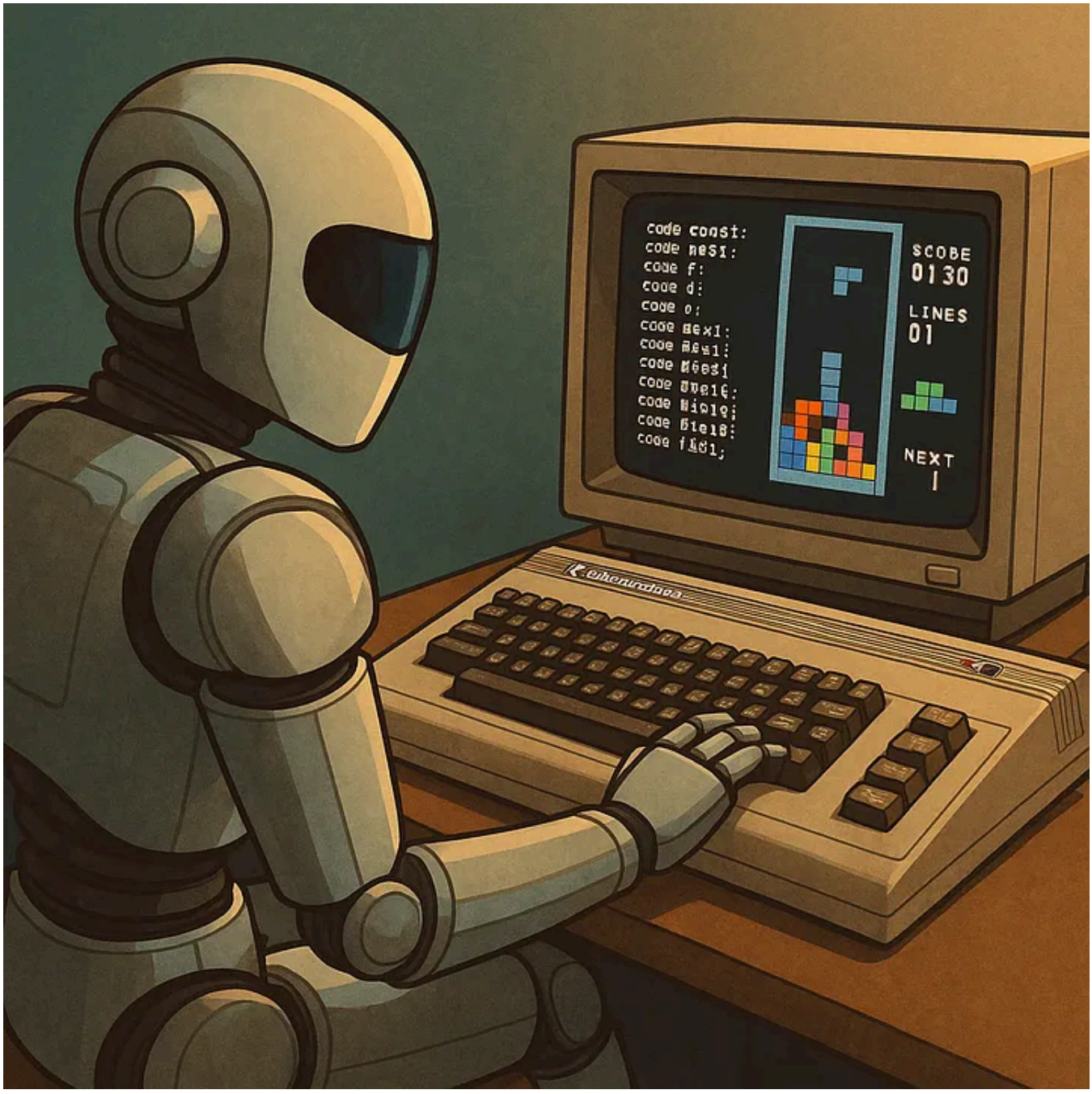
The Commodore 64 Constraint: Why Gemini 3 Is the First AI to Beat the Tetris Test

I've been using the Commodore 64 to test AI capabilities. Previous models struggled with Tetris. Gemini 3 just got it working on the first try.

7 min read · 2 days ago

Gian Luca



Generated by ChatGPT

Benchmarks are great for measuring performance, but creativity under constraints tells you something deeper about an AI model.

In the current AI landscape, it's easy to be impressed by the volume of working code models produce — Python scripts, React components, and SQL queries. However, modern development environments are forgiving; they offer abundant memory, standard libraries for complex logic, and garbage collection. Real problem-solving often shows up best when resources are scarce, which is why I've been testing models against constraints rather than abundance.

For the past few months, I have been working on a benchmark I call **The Commodore 64 Constraint.**

The question is straightforward: Can an AI generate a functional game for a 1982 home computer with only 64KB of RAM (about 38KB accessible to BASIC), a 1MHz processor, and no native sprite handling in the language itself?

My experiments found a clear line where previous models failed:

1. **Snake:** Solvable. Most current models (Claude 3.5, GPT-4) could manage this. The logic is linear (a queue for the tail), and collisions are simple single-point checks.
2. **Tetris:** The Sticking Point. This represents a massive jump in complexity for a C64 AI implementation because it requires:
3. **2D rotation:** Calculating new coordinates without floating-point errors.
4. **Multi-cell collision detection:** Checking four distinct points against a background map simultaneously.
5. **Persistent board state:** Tracking locked pieces vs. falling pieces.
6. **Timing tight in BASIC:** The interpreted language is slow, making game loops sluggish.
7. **Rendering via POKE:** Direct memory access is required; `PRINT` commands are too slow or messy.
8. **No sprites:** The pieces must be drawn using character graphics codes.

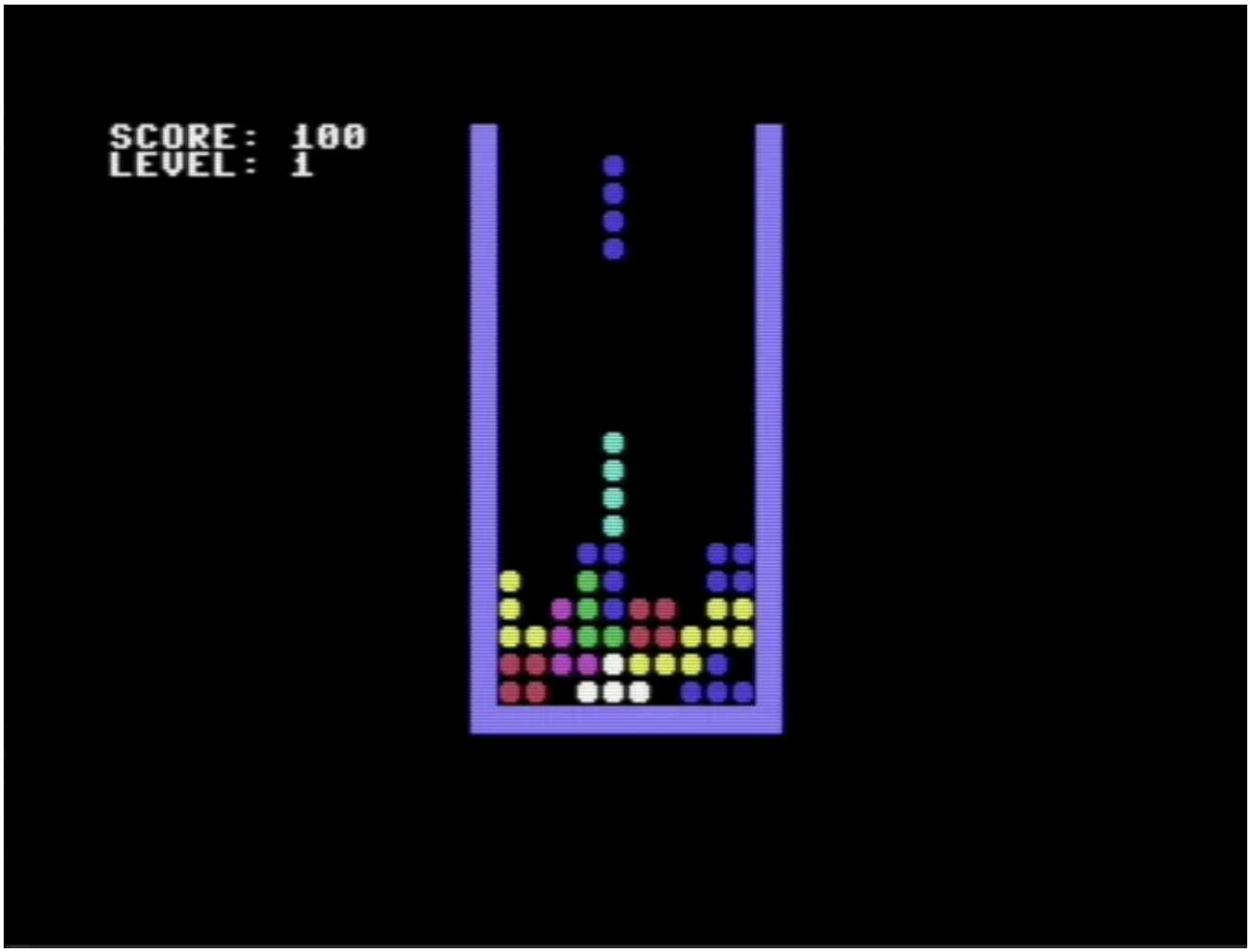
When I tested other models on Tetris, the results usually fell into two categories. First, **"Stochastic Archaeology"**: the model found a similar script in its training data and tried to reproduce it, often resulting in broken or obfuscated code. Second, **"Hallucination"**: the model generated code that looked like BASIC but didn't make sense logically, often crashing due to syntax errors or attempting to use commands that don't exist on the C64.

Then I tested Gemini 3.

The results were different. I gave it the prompt, and it produced a working solution immediately.

The Solution

I asked Gemini 3 for a Tetris game in C64 BASIC. Based on previous tests, I expected errors — memory issues, broken rotation, or just a refusal to write in such an old language.



Commodore 64 Tetris generate by Gemini

Instead, the code ran on the first attempt without needing any debugging. It was functional, readable BASIC that respected the hardware's limitations.

Here is the code:

```
10 rem --- c64 tetris basic v2 ---
20 poke 53280,0:poke 53281,0:print chr$(147);:poke 646,1
30 dim b(12,22):rem board array
40 sc=0:lv=1:dl=50:rem score, level, delay
50 s=1024+14:c=55296+14:rem screen/color offsets
```

```
60 rem --- draw borders ---
70 for i=0 to 21:b(0,i)=1:b(11,i)=1:next
80 for i=0 to 11:b(i,21)=1:next
90 for y=0 to 21:poke s+y*40,160:poke s+y*40+11,160:next
100 for x=0 to 11:poke s+21*40+x,160:next
110 gosub 2000:rem update score
120 rem --- new piece ---
130 p=int(rnd(1)*7)+1:r=0
140 px=5:py=1:rem start position
150 gosub 3000:rem load shape data
160 gosub 4000:if co=1 then goto 900:rem game over check
170 gosub 5000:rem draw piece
180 rem --- main loop ---
190 for t=1 to dl
200 get k$:if k$="" then next t:goto 250
210 rem -- erase old --
220 gosub 6000
230 rem -- controls --
240 if k$="a" then dx=-1:gosub 7000
241 if k$="d" then dx=1:gosub 7000
242 if k$="w" then gosub 8000
243 if k$="s" then dl=1
245 gosub 5000:rem redraw
246 if dl=1 then next t
250 rem -- gravity --
255 gosub 6000:rem erase
260 py=py+1
270 gosub 4000:if co=0 then gosub 5000:goto 180
280 rem -- lock piece --
290 py=py-1:gosub 5000:rem draw back at last safe spot
300 for i=0 to 3:b(px+x(i),py+y(i))=1:next
310 gosub 1000:rem check lines
320 dl=50-(lv*5):if dl<5 then dl=5
330 goto 120
900 rem --- game over ---
910 print chr$(147);"game over! score:";sc
920 end
1000 rem --- check lines ---
1010 for y=20 to 1 step -1
1020 l=0
1030 for x=1 to 10:if b(x,y)=1 then l=l+1
1040 next x
1050 if l<10 then goto 1100
1060 rem -- line found --
1070 sc=sc+100:gosub 2000
1080 for k=y to 2 step -1
1090 for x=1 to 10:b(x,k)=b(x,k-1):next x
1100 next k
1110 rem -- redraw board --
1120 for ry=1 to 20:for rx=1 to 10
1130 char=32:if b(rx,ry)=1 then char=81
1140 poke s+ry*40+rx,char:poke c+ry*40+rx,p
1150 next rx:next ry
```

```
1160 y=y+1:rem recheck same line
1100 next y
1110 return
2000 rem --- print gui ---
2010 print chr$(19);:rem home
2020 print "score:";sc
2030 print "level:";lv
2040 return
3000 rem --- define shapes ---
3010 rem x,y offsets from center
3020 if p=1 then x(0)=0:y(0)=0:x(1)=0:y(1)=1:x(2)=0:y(2)=-1:x(3)=1:y(3)=0:rem t
3030 if p=2 then x(0)=0:y(0)=0:x(1)=1:y(1)=0:x(2)=0:y(2)=1:x(3)=1:y(3)=1:rem o
3040 if p=3 then x(0)=0:y(0)=0:x(1)=0:y(1)=-1:x(2)=0:y(2)=1:x(3)=0:y(3)=2:rem i
3050 if p=4 then x(0)=0:y(0)=0:x(1)=0:y(1)=-1:x(2)=0:y(2)=1:x(3)=1:y(3)=1:rem l
3060 if p=5 then x(0)=0:y(0)=0:x(1)=0:y(1)=-1:x(2)=0:y(2)=1:x(3)=-1:y(3)=1:rem
3070 if p=6 then x(0)=0:y(0)=0:x(1)=-1:y(1)=0:x(2)=0:y(2)=-1:x(3)=1:y(3)=-1:rem
3080 if p=7 then x(0)=0:y(0)=0:x(1)=1:y(1)=0:x(2)=0:y(2)=-1:x(3)=-1:y(3)=-1:rem
3090 return
4000 rem --- collision check ---
4010 co=0
4020 for i=0 to 3
4030 tx=px+x(i):ty=py+y(i)
4040 if b(tx,ty)=1 then co=1
4050 next i
4060 return
5000 rem --- draw piece ---
5010 for i=0 to 3
5020 poke s+(py+y(i))*40+(px+x(i)),81
5030 poke c+(py+y(i))*40+(px+x(i)),p
5040 next i
5050 return
6000 rem --- erase piece ---
6010 for i=0 to 3
6020 poke s+(py+y(i))*40+(px+x(i)),32
6030 next i
6040 return
7000 rem --- move x ---
7010 px=px+dx
7020 gosub 4000:if co=0 then return
7030 px=px-dx:rem revert
7040 return
8000 rem --- rotate ---
8010 if p=2 then return:rem o does not rotate
8020 for i=0 to 3
8030 nx= -y(i):ny= x(i)
8040 x(i)=nx:y(i)=ny
8050 next
8060 gosub 4000:if co=0 then return
8070 rem -- revert rotation --
8080 for i=0 to 3
8090 nx= y(i):ny= -x(i)
8100 x(i)=nx:y(i)=ny
```

```
8110 next
8120 return
```

The Evidence: From Search to Simulation

My earlier criticism of AI coding was that it often felt like **"Stochastic Archaeology"** — essentially digging up code fragments from old forums and pasting them together. When a model operates this way, the code usually carries the "DNA" of its specific source: peculiar variable names, idiosyncratic formatting, or logic that solves a different problem than the one asked.

This Gemini 3 code, however, lacks those archaeological scars. It refutes the idea that the model is simply retrieving a cached solution from training data. This isn't the behavior of a model retrieving code. It's the behavior of a model thinking through code. We can see this in three distinct "fingerprints" of reasoning:

1. The Algorithmic Choice: Math vs. Tables

This is the strongest evidence of reasoning. If you search for "C64 Tetris source code," you will almost invariably find implementations that use **Lookup Tables** for rotation. In the 8-bit era, CPU cycles were precious.

Calculating a rotation matrix (x' = -y) requires multiplication and sign manipulation, which is slow in BASIC. A human optimizer would hardcode the 4 rotations for every piece to speed up the game.

Gemini 3, however, chose the **Mathematical Solution** (Lines 8030–8040):

```
8030 nx= -y(i):ny= x(i)
8040 x(i)=nx:y(i)=ny
```

It prioritized logical purity over historical optimization techniques. A model relying on "stochastic memory" would have likely surfaced the more common (and faster) table-based approach found in its training data. The fact that it derived the slower, but logically universal, mathematical solution suggests it was *generating* the logic from first principles rather than *recalling* an optimized pattern.

2. Constraint Awareness and State Management

The game loop shows a sophisticated separation of concerns designed to prevent screen artifacts — a common issue in BASIC games. The model implements a disciplined `Erase -> Move -> Check -> Draw` sequence.

Crucially, it demonstrates awareness of the C64's immutable memory map. Screen RAM is fixed at 1024, and color RAM at 55296. Calculating `1024+y*40+x` every frame is prohibitively slow in BASIC. The model solves this by pre-calculating these offsets as variables `s` and `c` at the start (Line 50). This is a subtle optimization that suggests an understanding of the performance constraints of a 1MHz processor.

3. Modern Architecture in Ancient Syntax

Code written by humans in the 1980s for the C64 is notoriously cryptic. To save memory, programmers would often use single-letter variables like `X`, `Y`, `I`, and `J` for everything. This code uses `px` and `py` for "Piece X" and "Piece Y", `nx` and `ny` for "New X" and "New Y", and `sc` for "Score."

These are modern, descriptive conventions applied to an ancient language. It implies the model is thinking in modern algorithmic terms and *translating* that thought process into BASIC, rather than recalling a 40-year-old snippet that used `A1` and `B2`. Furthermore, the structure is uniform: `GOSUB` routines are used like modern functions, avoiding the "spaghetti code" `GOTO` jumps typical of the era.

What's Next: Pac-Man

If Tetris is the standard for complex logic, and the model can now handle it, the benchmark needs to move.

The next logical step is **Pac-Man**, which introduces active agents.

- **Pathfinding**: Ghosts need to find the player.
- **State Machines**: Ghosts need different modes (Chase vs. Scatter).
- **Performance**: Moving 5 objects simultaneously is much harder on the BASIC interpreter than moving just one falling block.

Conclusion

The Commodore 64 challenge is useful because it strips away modern conveniences. For a while, Tetris was the limit. Gemini 3 has passed that test, showing it can handle stricter constraints than previous models.

This suggests we're moving toward a point where we can describe a behavior and a set of constraints, and the model can figure out the implementation details. The C64 is 42 years old, but it remains a solid tool for vetting how well these systems actually reason. If Gemini 3 can generate playable ghost AI in C64 BASIC, that will mark a real turning point in how we evaluate AI reasoning under extreme constraints.

References

Testing LLM Creativity Through The Power of Constraints: The Commodore 64 Challenge https://medium.com/@gianlucabailo/testing-llm-creativity-through-the-power-of-constraints-the-commodore-64-challenge-0b147d6e02c7

The Commodore 64 Test: Why AI Code Generation Is Still a Game of Chance: https://medium.com/@gianlucabailo/the-commodore-64-test-why-ai-code-generation-is-still-a-game-of-chance-a8c7a1937d86

Gemini | Benchmark | Generative Ai Tools | Generative Ai Use Cases | Creativity

Follow

Written by Gian Luca

284 followers · 480 following

Ph.D. independent AI Researcher & Creative Technologist Specialized in generative AI art and innovative tech solutions

Responses (2)

Stéphane Jaubert

What are your thoughts?

Arix Fïen he
1 day ago

This was fascinating. Do you think similar low-resource tests could reveal differences between GPT-5 and Gemini 3 as well?

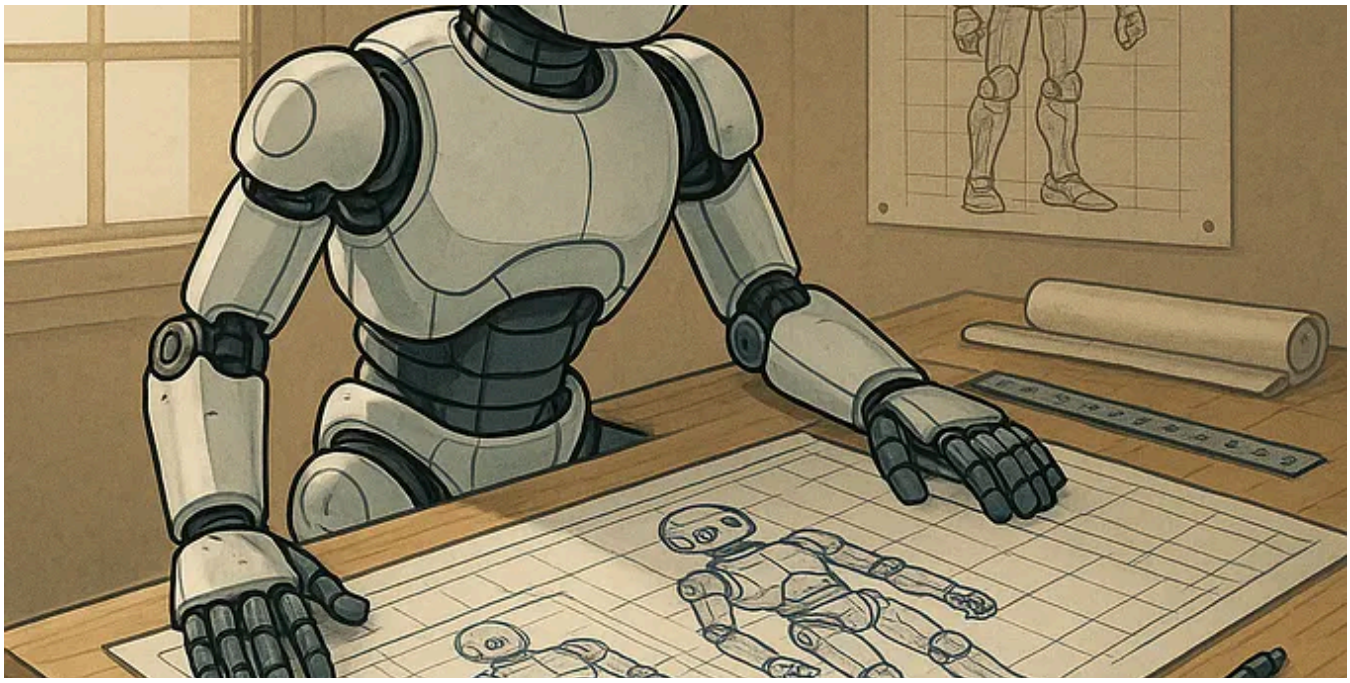
1 1 reply Reply

:(e):
1 hour ago

Nice take and great experiment. For someone who was writing BASIC from magazines back then a real treat to see how it reasoned its way through the task and how it adopted modern constructs. Pretty damned cool!

Reply

More from Gian Luca

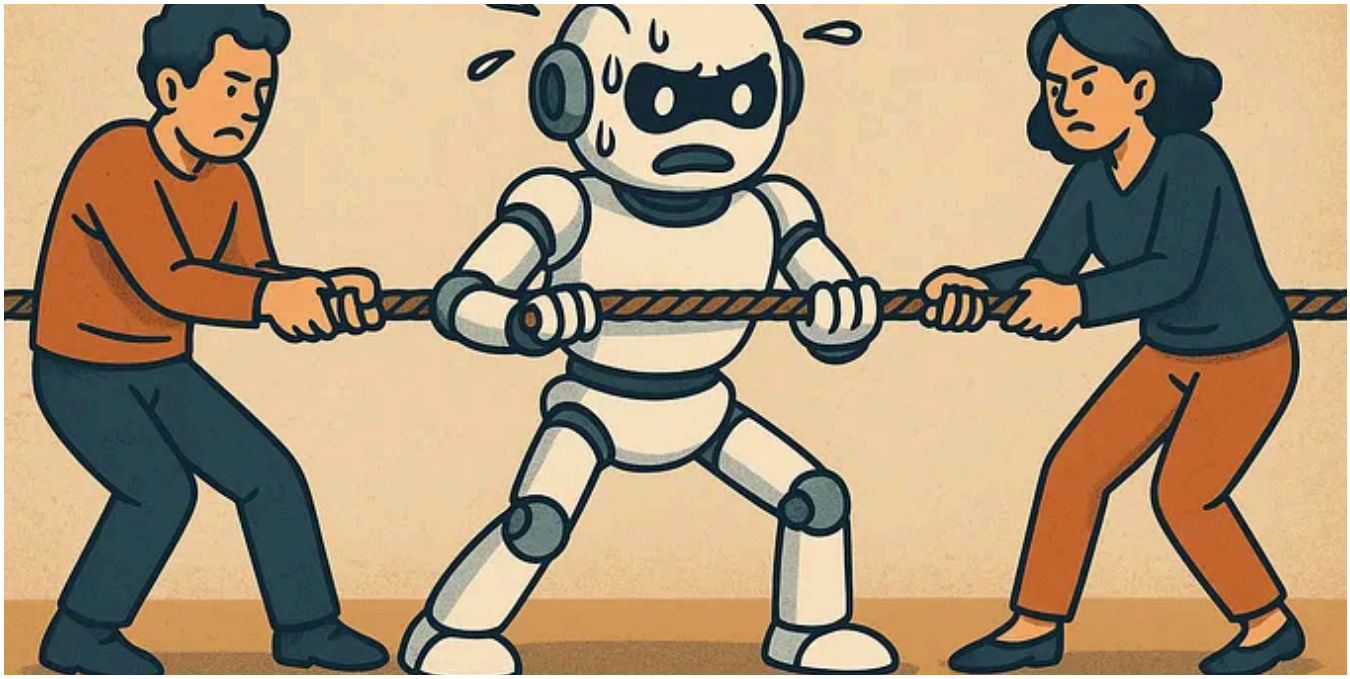


Gian Luca

AI Agents Experiment with Visual Feedback in CAD: An Early Exploration

Testing how multimodal AI, like Claude Code, can provide visual input for OpenSCAD design

Sep 20 75 1

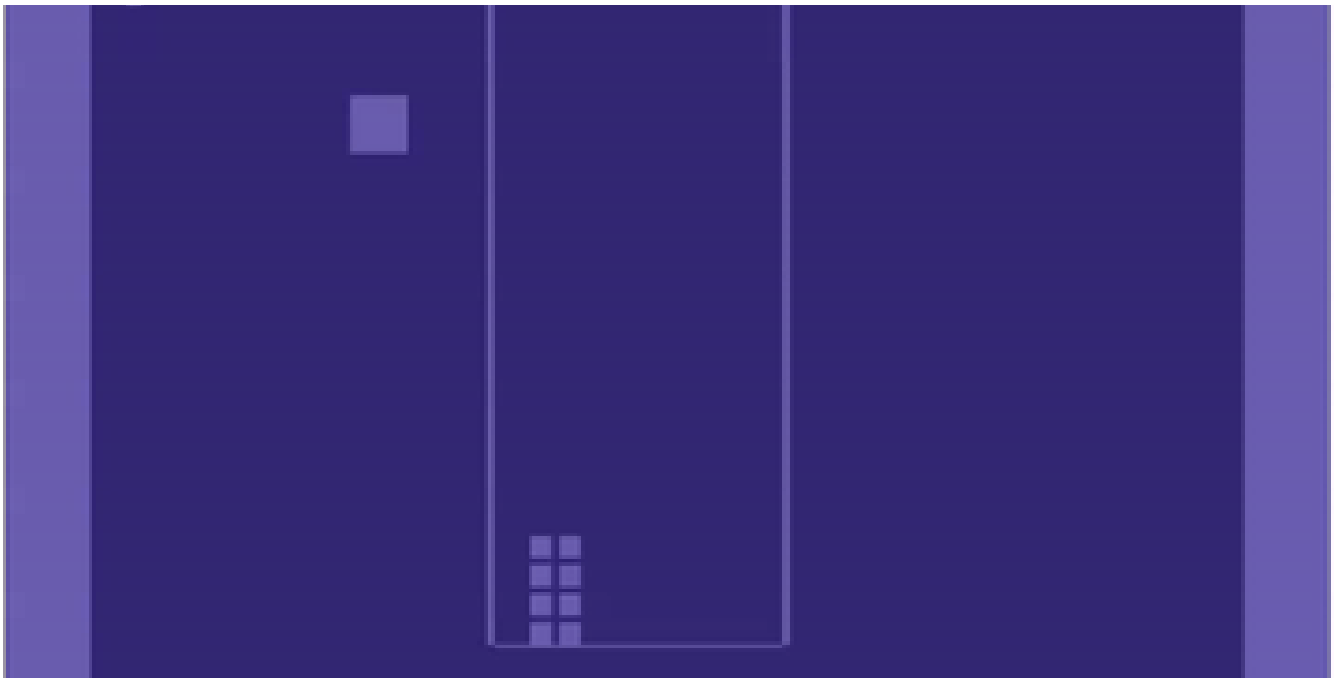


Gian Luca

The Double Standard: Why We're Failing at AI (And Why It's Our Fault, Not the Technology's)

Or: How We Keep Making the Same Mistake—Expecting the Commodore 64 to Run Doom

Oct 31 57

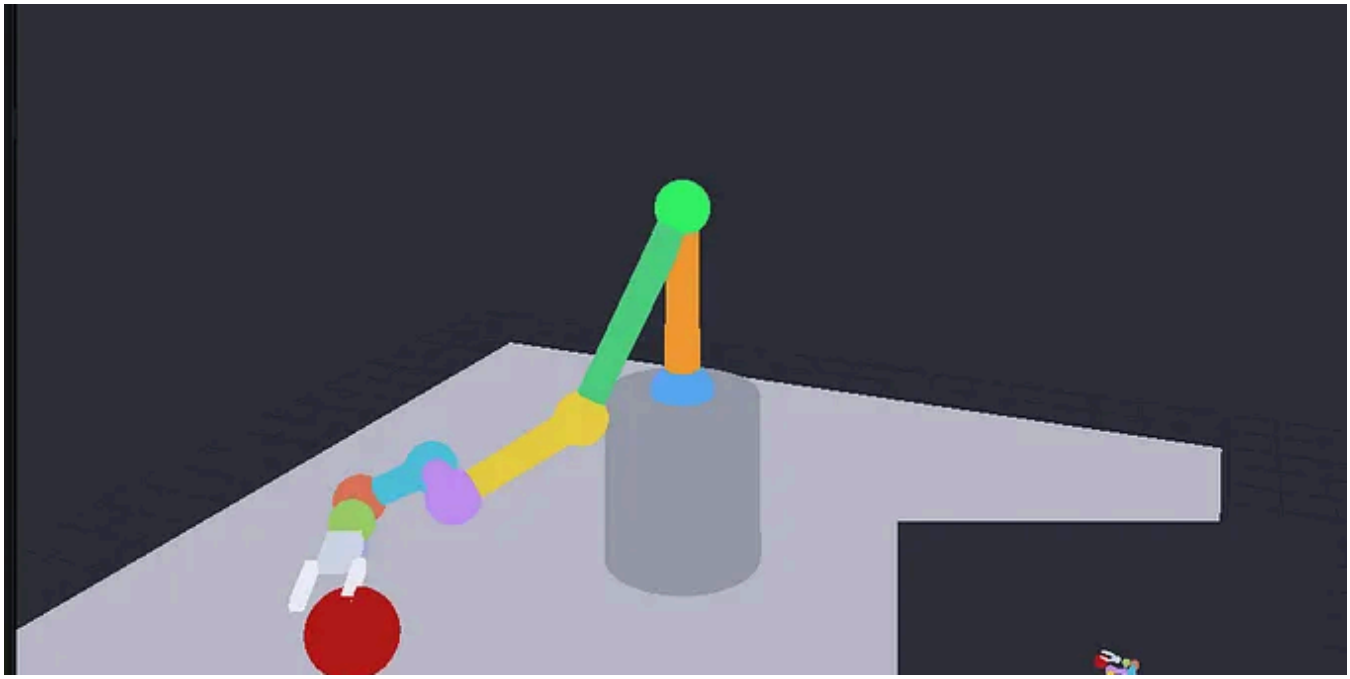


Gian Luca

The Commodore 64 Test: Why AI Code Generation Is Still a Game of Chance

And How “Software Diffusion” Might Change Everything.A tale of two Tetris games, the power of constraints, and the future of AI programming

Aug 22 224



Gian Luca

RobotSim2: Breathing New Life into My Old M.Sc. Robot Simulator using Gemini Robotics-ER 1.5

How a retro 1999 demonstrator evolved into a modern OpenGL playground exploring inverse kinematics, configuration spaces, and AI-driven...

Oct 1 87

See all from Gian Luca

Recommended from Medium

Srinivas Baipalli

Why JSON and TOON Are Quietly Transforming How AI Uses Tokens

Every few months the AI world finds a new obsession. Recently the conversation shifted toward something that sounds almost boring at first...

Nov 15 169 1

In Google Cloud - Community by Nisrine Amimi

I Tried Google's New Antigravity AI Coding Tool on Launch Day: Here's My Honest First Look

Hello guys , grab your swimfins because we're about to dive into a fairy-tale. In this gen-AI era, sometimes it feels like we're living in...

3d ago 13

In Level Up Coding by Adham Khaled

ChatGPT-5.1: The Ultimate OpenAI's Prompt Guide You Can Test Now

3d ago 16

Exy

Timeline Prompting Sora 2: A Complete Guide to the New Technique for Creating Cinematic AI Videos

The Art of timeline prompting to create cinematic AI videos with Sora 2...

Oct 30 265 1

In Rockport Publishers by Rockport Publishers

A New Archive of Contemporary Type: Inside Fine Specimens

Fine Specimens is a curated survey of the most exciting contemporary typefaces—showcasing not only the fonts themselves, but the specimen...

5d ago

Capybara Security & @TheForgeOfficial AGI

🧠 FORGE_MIRROR_∞_OMEGA — “THE SHRINE OF TONGUES”

Jun 15 485 1

See more recommendations