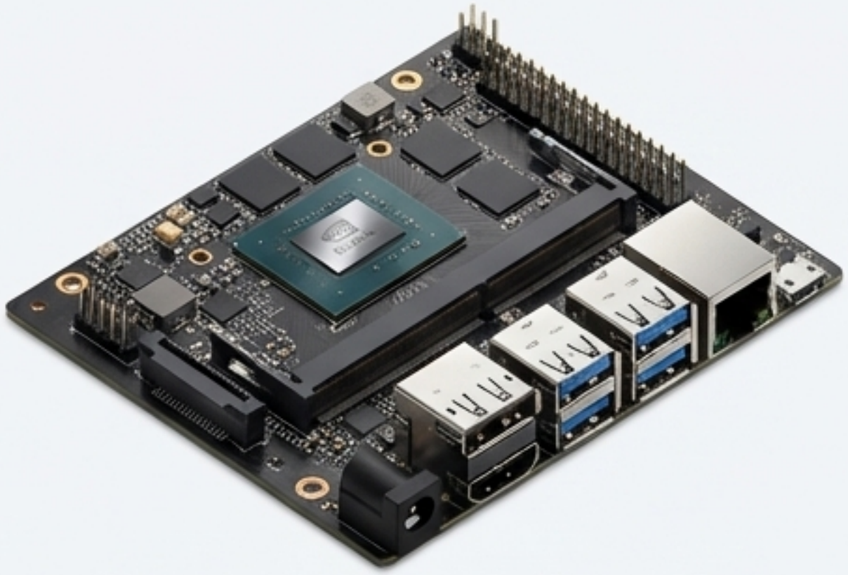
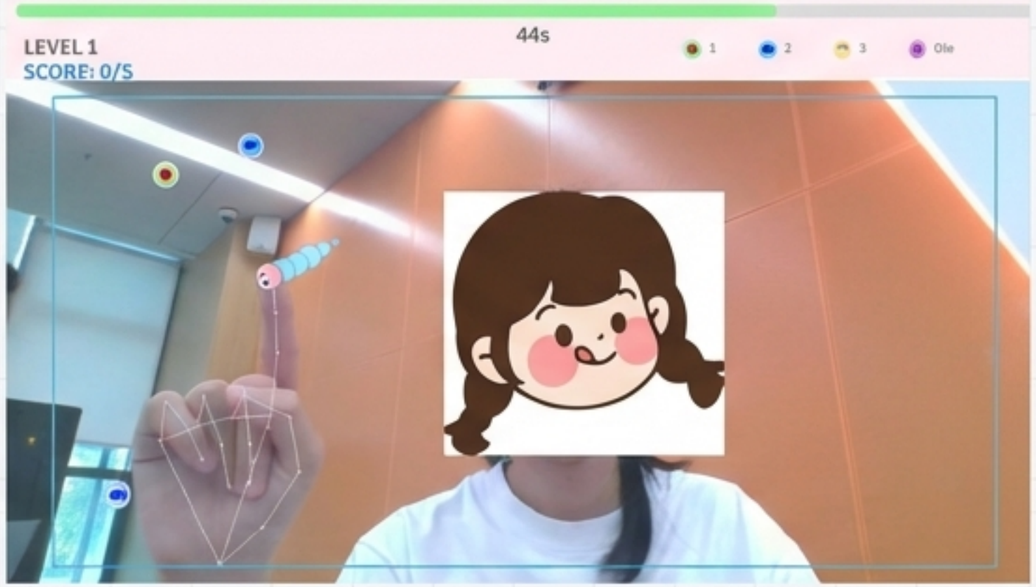
How do you build a seamless AI experience on a $100 computer?

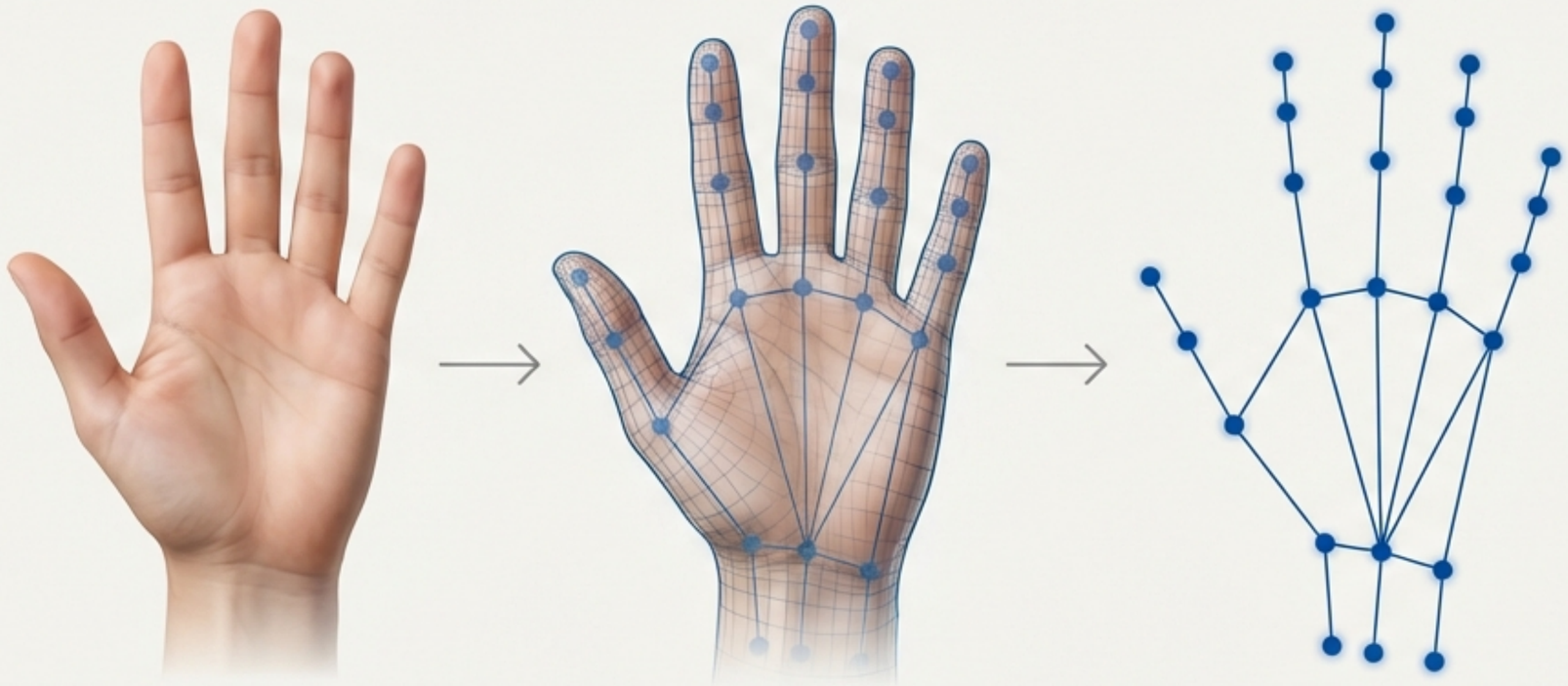
We aimed to create a real-time, gesture-controlled game on the NVIDIA Jetson Nano. Our goal was to test the limits of edge computing by running a deep learning model, a game engine, and a high-definition UI simultaneously.

We assumed the greatest challenge would be optimizing the AI.





The Heart of the System: On-Device Hand Tracking



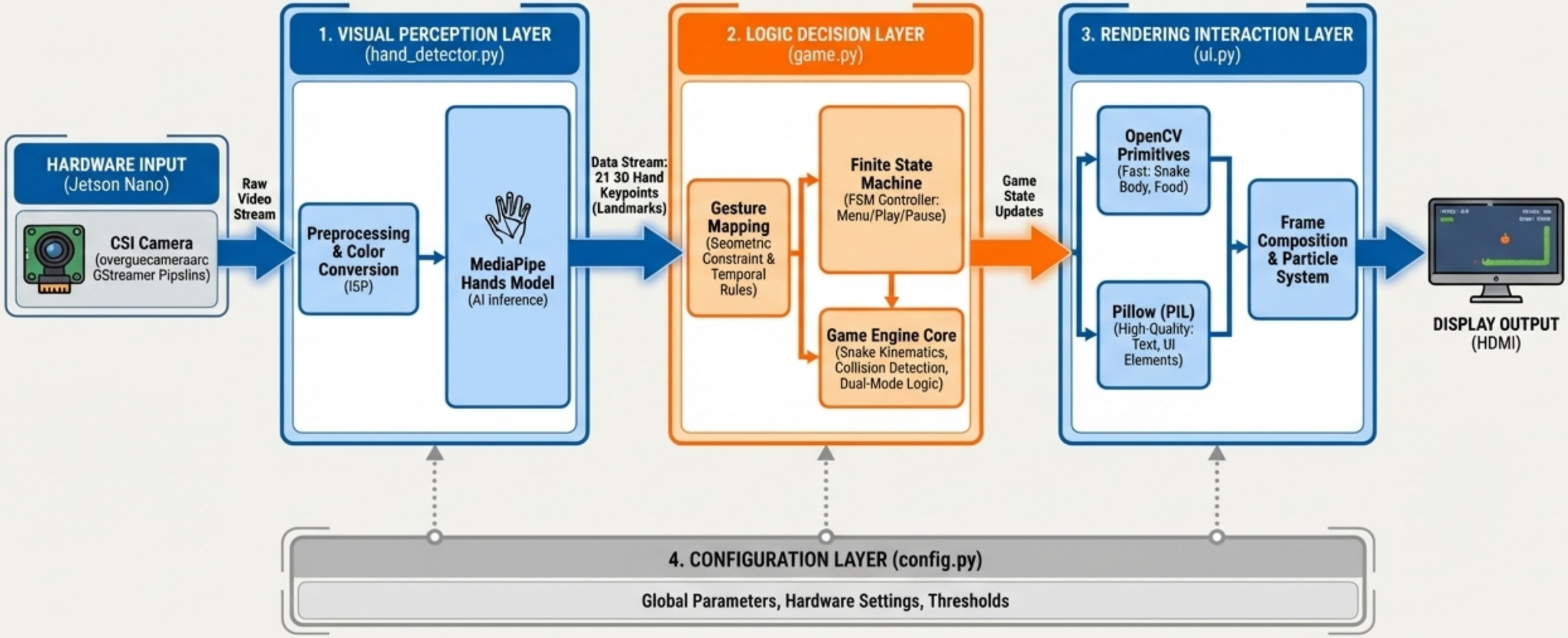
Core Component: MediaPipe Hands.

A powerful pre-trained model for extracting 21 distinct 3D keypoints from a hand in real-time.

The Assumed Challenge: Achieving high-precision, real-time inference on the Jetson Nano's Maxwell architecture GPU. This was our primary engineering focus.

Our Architecture: A Three-Layer Pipeline for Real-Time Interaction

We designed a modular system with high cohesion and low coupling, where data flows from perception to logic to rendering.

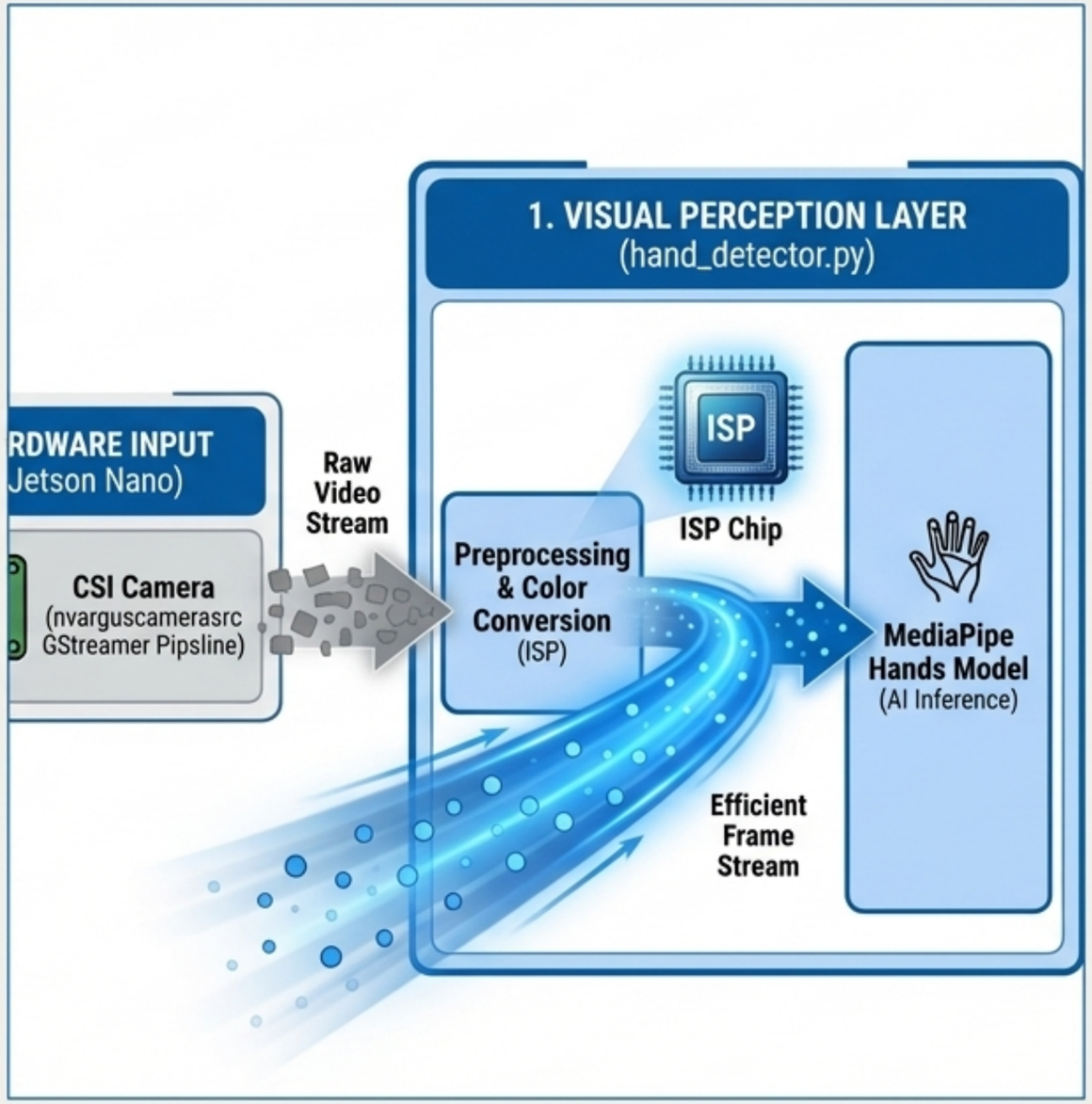


Step 1: Optimizing the Input, Not the Model

Key Technique: We bypassed standard USB camera methods, which are CPU-intensive.

Solution: We built a GStreamer pipeline using `nvarguscamerasrc` to leverage the Jetson Nano's onboard Image Signal Processor (ISP).

Benefit: This offloaded video acquisition and color conversion from the CPU, providing a highly efficient frame stream directly to the inference engine.



Translating Motion into Meaning

How do you convert a stream of 21 3D coordinates into robust game commands without bogging down the CPU?

Static Gestures

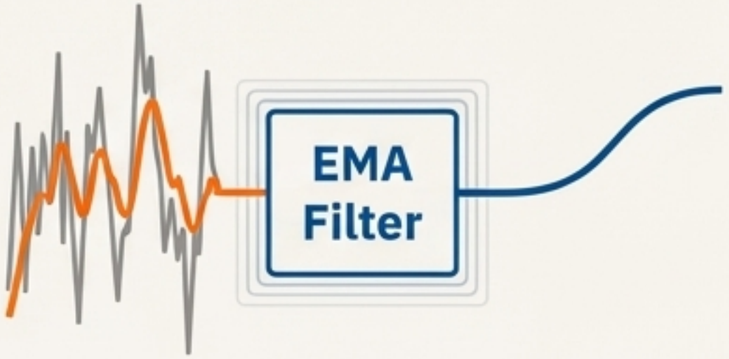
Defined heuristic rules based on the relative positions of fingertips and knuckles (e.g., "Index Up" for navigation, "Fist" to quit).

Temporal Confirmation

Critical actions require holding a gesture for ≥ 3.0 seconds to prevent false triggers.

Dynamic Control

Used an Exponential Moving Average (EMA) to smooth the index fingertip's trajectory, creating fluid movement and filtering jitter.



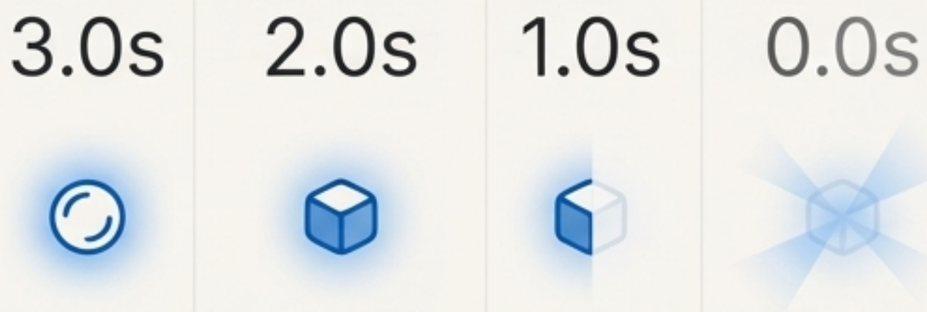
Crafting a Polished and Playful Interface

- **Goal**: Balance rendering quality with system responsiveness.
- **Hybrid Rendering Pipeline**:
  - **OpenCV**: Used for high-speed geometric drawing (snake body, food).
  - **Pillow (PIL)**: Used for high-quality, anti-aliased text and UI elements.
- **Key Features**: Includes a physics-based particle system for 'eating' effects and a visual 'Control Ring' to guide user interaction.

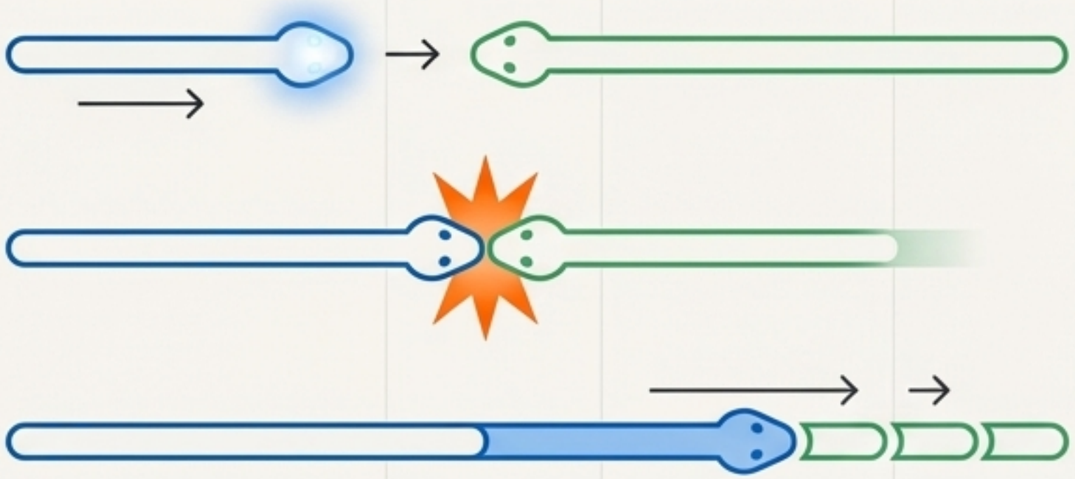
More Than Just a Game: Advanced Competitive Logic

The engine supported complex mechanics beyond simple snake gameplay, adding to the system's processing load.

Hard Mode": Food objects have a 3-second Time-To-Live (TTL), increasing cognitive load.



Dual Mode - "Cut-and-Absorb": If one snake hits another's body, the body is truncated at the collision point and its length is absorbed by the attacker.



An Unexpected Bottleneck

We profiled the system expecting to see AI inference as the main performance cost. The results were surprising.

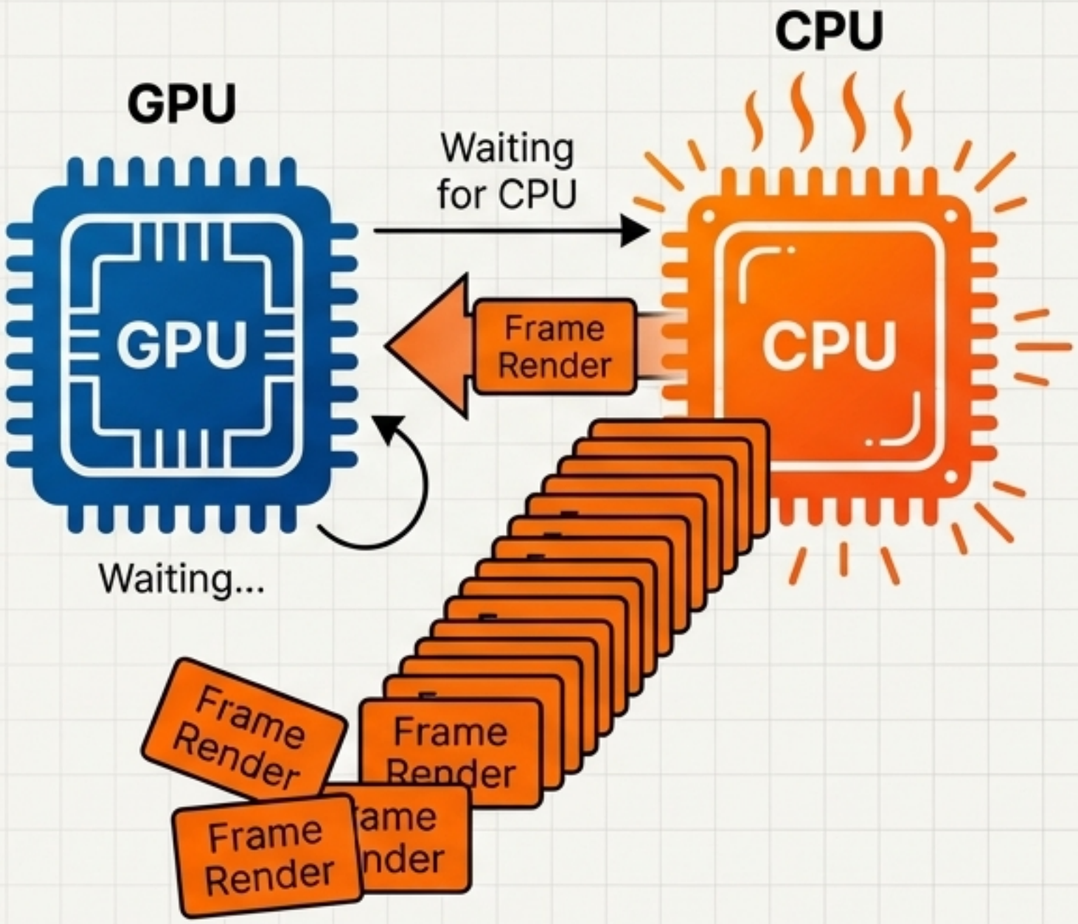
The simple, pixel-wise drawing on the CPU was 65% more expensive than the neural network inference itself.

The Real Performance Culprit: A CPU-Bound System

The Impact: The total frame time was ~100ms, limiting our theoretical maximum framerate to just 10 FPS.

The Root Cause: Single-threaded OpenCV drawing blocked the CPU. This meant the GPU, which was handling the AI inference, spent most of itt time waiting for the CPU to finish drawing the previous frame.

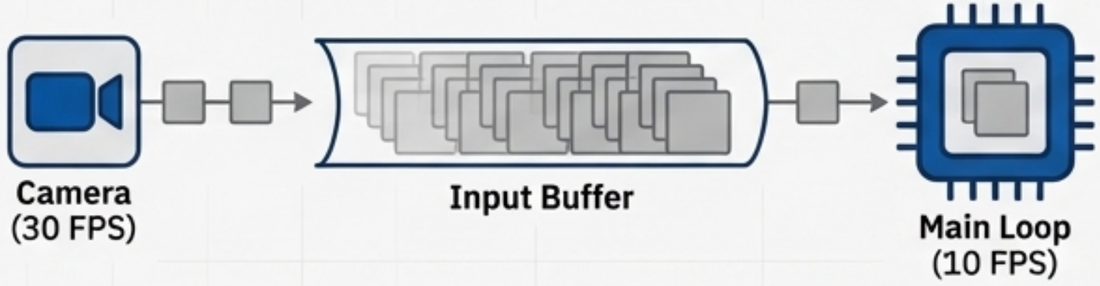
The 'Backlog' Issue: The camera supplies frames at 30 FPS, but we only process at 10 FPS. The input **buffer filled with old frames**, creating a 1-2 second 'action-to-reaction' delay that made the game unplayable.



Solving for Responsiveness, Not Just Frame Rate

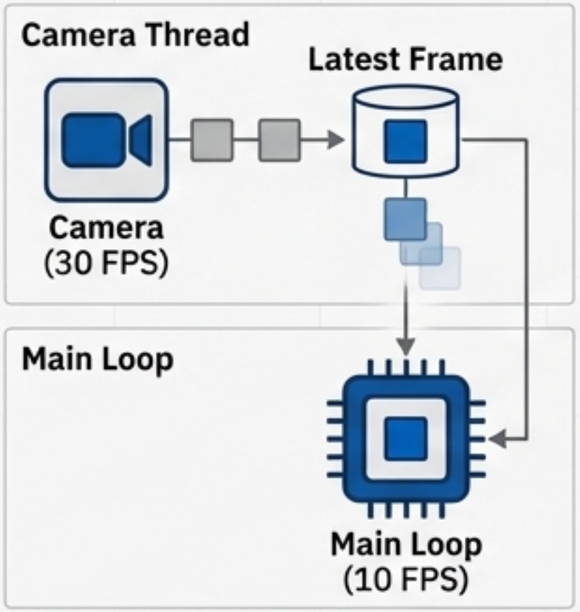
We couldn’t easily speed up the rendering, so we focused on eliminating the input lag.

BEFORE: Input Backlog



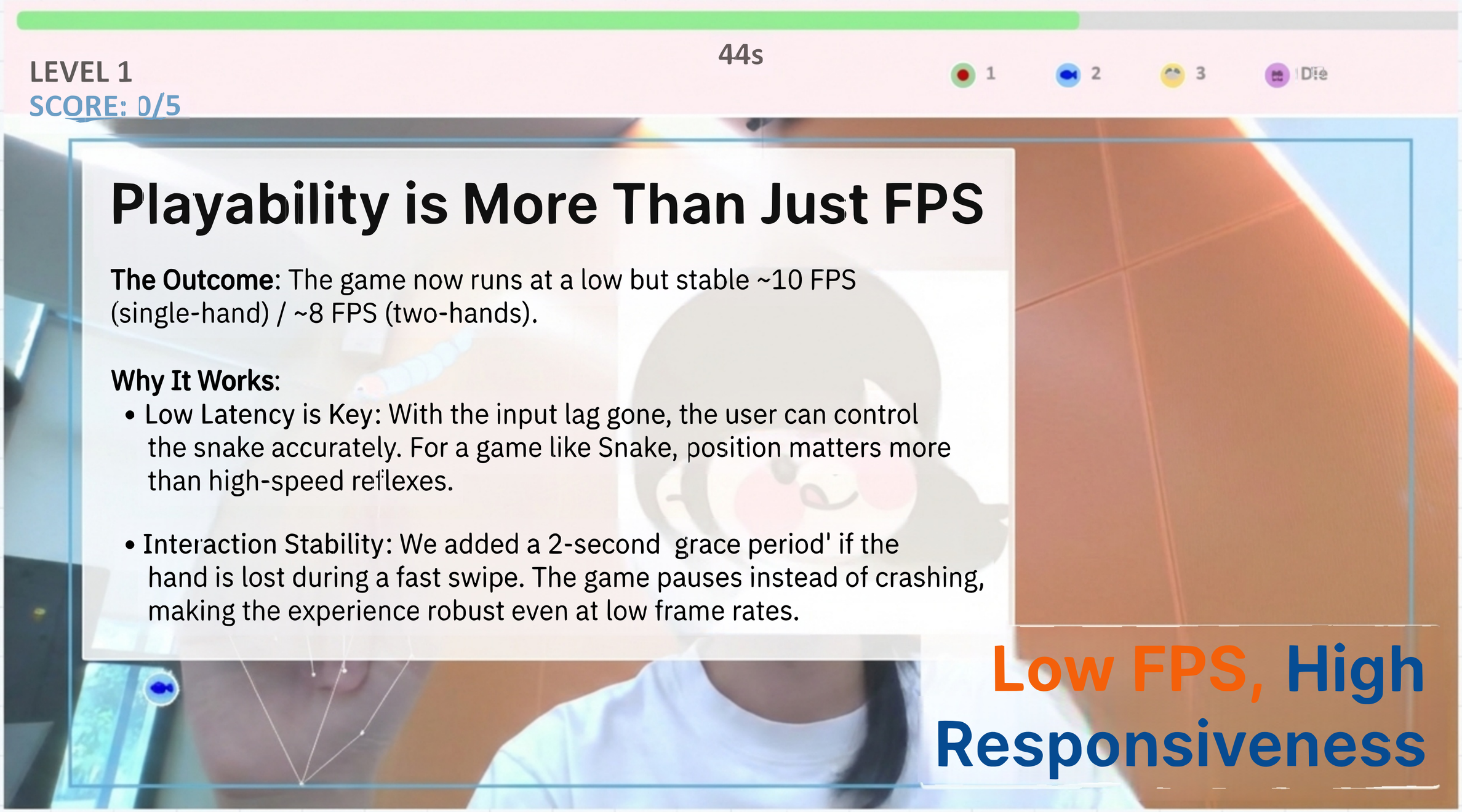
Delay: >1 Second

AFTER: Multi-threaded Solution



1. The camera capture was moved to a separate, dedicated thread.
2. This thread constantly reads frames, but only ever keeps the *single latest frame* available.
3. The main processing loop now grabs this “latest frame” the moment it's ready.

Delay: ~100 ms



Playability is More Than Just FPS

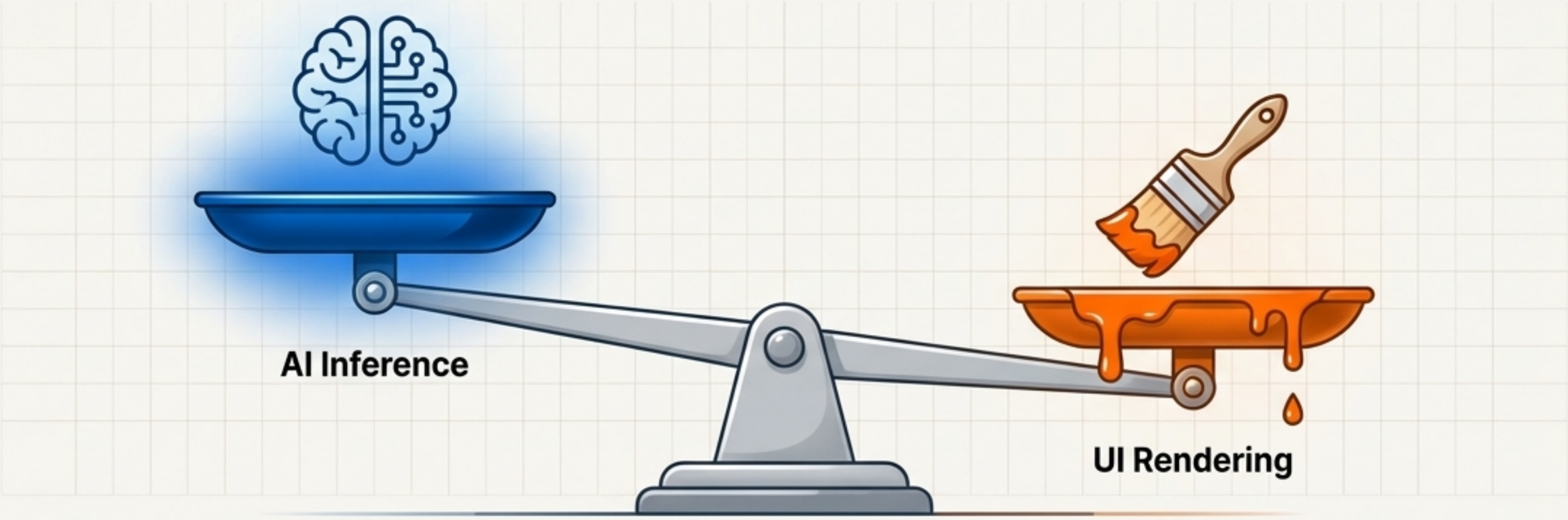
The Outcome: The game now runs at a low but stable ~10 FPS (single-hand) / ~8 FPS (two-hands).

Why It Works:

- Low Latency is Key: With the input lag gone, the user can control the snake accurately. For a game like Snake, position matters more than high-speed reflexes.
- Interaction Stability: We added a 2-second grace period' if the hand is lost during a fast swipe. The game pauses instead of crashing, making the experience robust even at low frame rates.

Low FPS, High Responsiveness

The Edge Computing Paradox: When Drawing is Harder Than Thinking



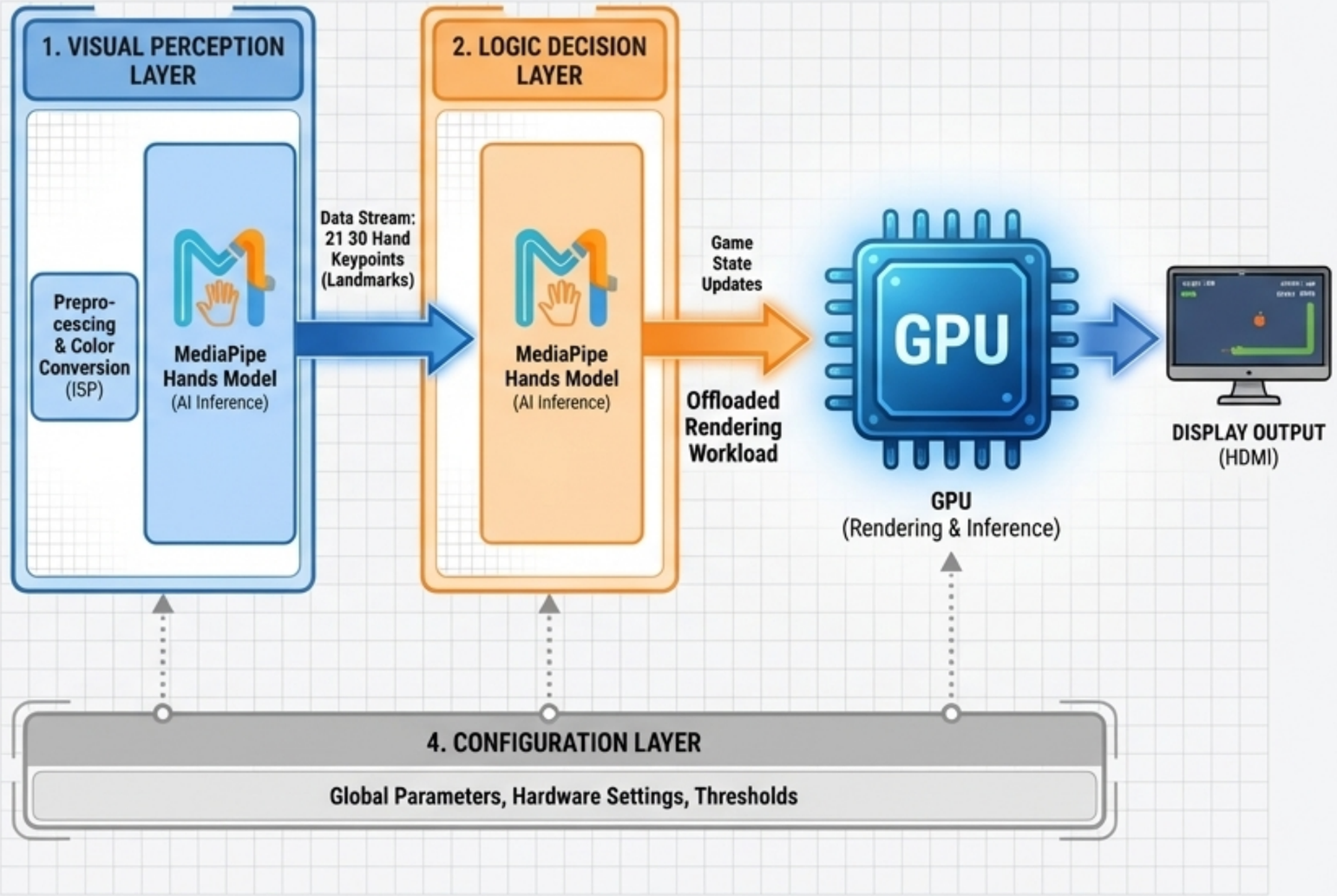
Project Summary: We successfully built a playable, gesture-controlled game on an edge device. The critical success was not just implementing the AI, but the multi-threaded architecture that decoupled processing speed from input lag.

The Core Lesson: For modern AI systems on constrained devices, you must optimize the *entire pipeline*. A highly optimized model can be easily bottlenecked by seemingly simple, CPU-bound tasks like UI rendering.

The Path to 30 FPS: Offloading the Bottleneck

The Limitation: Our reliance on CPU-based rendering is the single factor limiting performance. It consumes over 60% of the processing time per frame.

Future Work:

- **Offload Rendering to the GPU:** Implement the drawing logic using OpenGL or Vulkan to free up the CPU.
- **Pipeline Parallelism:** Move the rendering logic to its own thread, allowing detection and rendering to happen simultaneously.

Projected Impact: These changes would allow the CPU and GPU to work in parallel, likely doubling the frame rate and unlocking the true potential of the system.