ODYSSEY

Agora-2: An Action-Conditioned World Model for Real-Time Interactive Environments

Team Odyssey
research@odyssey.systems

Abstract

We present Agora-2, an interactive world model in which human players and autonomous agents interact in a persistent shared world. Building on Agora-1, Agora-2 separates shared world simulation from the generation of each player's view. A simulation model predicts how human and agent actions change the world, while a rendering model translates those changes into generated video for each player. Actions from all participants update a common world state, allowing one participant's movement or interaction to affect what others subsequently observe. We implement the system in an isometric action-game environment that supports movement, combat, death, and respawn, with up to four concurrent human controllers alongside autonomous agents. Each player receives an independently generated view grounded in the same evolving world state. Agora-2 brings these components together in a working system that tackles two central challenges: rendering states produced by a learned simulation and coordinating independently generated views of a shared world.

1 Introduction

Shared worlds create opportunities for interaction in which participants must respond both to the environment and to one another. A multiplayer world model must predict how actions change a shared environment and depict those changes consistently from each player's viewpoint. These requirements can fail independently: a model may generate plausible images of an incorrect transition or render different views of a correct state inconsistently. Agora-2 separates these tasks by learning state transitions over a structured world state and generating each view from the resulting state. We study this architecture in an instrumented isometric action-game environment, where movement, combat, animation, and the entity lifecycle have explicit representations.

Agora-1 explored action-conditioned world modeling in a first-person multiplayer environment [Cameron, 2026]. Its video model generated each player's view, while a dynamics head combined features from that model with player actions, pose, and local wall-depth information to predict movement and hits on other players. Predicted movement advanced the player's camera pose, which determined the geometry supplied to the video model at the next step. Simulation and rendering therefore formed a coupled loop: visual features informed motion prediction, and predicted motion informed subsequent views. Agora-2 reorganizes this loop around an independent simulation model that predicts movement, combat, and animation from structured entity histories, actions, and local geometry. Human controls and a learned agent policy feed the same shared simulation, whose updated state then drives each player's renderer. This design allows world updates to be computed and inspected independently of the generation of any individual view.

We develop this architecture in an instrumented isometric action-game environment, supporting up to four human-controlled entities alongside sixteen autonomous entities. The shared simulation explicitly represents entity identity, position, health, animation, death, and respawn, providing a common basis for movement and combat across players' views. Inspecting the shared state before rendering allows errors in predicted interactions to be distinguished from errors in their visual depiction.

This report presents the architecture and implementation of `Agora-2`, describing how the simulation model, rendering model, and agent policy work together through shared state and action interfaces. We detail the renderer's structured attention mechanism, the training data, and the serving system that supports per-player video generation. Component evaluations measure short-horizon rendering reconstruction, the balance between visual history and state conditioning, simulation prediction accuracy, and rendering efficiency. Together, these components enable humans and autonomous agents to interact through a shared learned simulation, with each player receiving an independently generated view of the evolving world.

2 Related Work

Interactive world models. Generative world models explore how video prediction can support controllable environments. Genie learns interactive environments from unlabeled video using a latent action model [Bruce et al., 2024], while Microsoft Research's Muse jointly models game visuals and controller actions for gameplay ideation [Kanervisto et al., 2025]. Hunyuan-GameCraft studies keyboard- and mouse-controlled video generation across diverse game environments [Li et al., 2025], and LingBot-World develops real-time interaction and extended rollouts across varied visual settings [Robbyant Team et al., 2026]. GAIA-2 addresses controllable multi-camera generation for autonomous driving, conditioning on vehicle dynamics, surrounding agents, and environmental structure [Russell et al., 2025]. `Agora-2` focuses on shared interaction: actions from multiple human and autonomous participants update an explicit world state that supplies the conditions for each player's generated view.

Multiplayer world models. Extending interactive generation to multiple participants requires accounting for their joint actions and maintaining agreement about their effects. MIRA conditions video generation on multiple players' action streams in Rocket League, studying action attribution and tightly coupled physical interactions [Hu et al., 2026]. Solaris generates multiple players' observations in Minecraft, exchanging information between their visual tokens through multiplayer attention and evaluating movement, memory, interaction, and view consistency [Savva et al., 2026]. `Agora-2` coordinates participants through a separate learned simulation: joint actions advance shared entity state, which each renderer receives alongside its own visual history. This makes the predicted world update available for inspection independently of its depiction in any player's view.

Explicit state and shared simulation. MultiGen decomposes world modeling into memory, observation, and dynamics, using persistent external memory to support editable environments and multiplayer interaction [Po et al., 2026]. `Agora-2` follows this broader direction, with structured entity state connecting learned dynamics to per-player rendering. Concurrent work, MASS, similarly separates a learned state-update model from view generation and evaluates state accuracy and cross-view consistency in multiplayer Snake [Cai et al., 2026]. `Agora-2` studies this separation in an isometric action-game environment with movement, combat, animation, and entity lifecycle, integrating human controls and a learned agent policy into the same persistent simulation.

3 Problem Formulation

`Agora-2` predicts how a shared world evolves under actions from players and autonomous agents, then generates each player's view of that world. Let S_k denote the world state at simulation update k, including entity identities, poses, health, animation, and view parameters, and let G describe the scene geometry and appearance. The joint action input

$$A_k = \left(A_k^{\text{player}}, A_k^{\text{agent}}\right) \tag{1}$$

collects player controls and actions from a learned agent policy, with each action associated with the entity it controls. The policy receives observations derived from the world state and local geometry.

Given h steps of state and action history, the simulation model D_θ selects movement branches, predicts facing changes and interaction outcomes, and decodes these predictions into world-space updates. The server then applies these outputs to advance the shared state:

$$\bar{U}_k = D_\theta(S_{k-h+1:k}, A_{k-h+1:k}, G), \tag{2}$$

$$S_{k+1} = \mathcal{H}(S_k, \bar{U}_k). \tag{3}$$

Here, $\bar{U}_k$ contains the selected motion and predicted interaction outcomes. Movement-branch selection, facing, and interaction outcomes are learned; fixed decoding converts the predictions into world-space updates. The server's update function $\mathcal{H}$ integrates these outputs and manages lifecycle events under the session's configured rules.

For each player's view c, let t index generated video latents, distinct from the world-update index k. The conditioning information C_t^c contains the local map, entity, effect, and viewpoint information needed to generate latent t. It is constructed from the shared world state and includes entity states aligned with each of the two output frames. Given these conditions and L preceding latents, the renderer generates the next latent and the decoder produces the corresponding frames:

$$z_t^c \sim R_\phi(z_{t-L:t-1}^c, C_t^c), \tag{4}$$

$$(I_{2t}^c, I_{2t+1}^c) = \mathrm{Dec}(z_t^c; \text{decoder history}). \tag{5}$$

Each view maintains its own visual history while receiving conditions derived from the same shared world state. Generation begins from a learned beginning-of-sequence representation.

4 Method

4.1 Method Overview

`Agora-2` combines player controls and a learned agent policy with a shared simulation model and a rendering model for each player's view (Figure 1). The simulation model predicts movement and interaction outcomes, which the server applies to the shared world state. Scene geometry, appearance, and the updated entity and effect states provide the conditions for each view. The rendering model generates video from these conditions and its own visual history, using a representation autoencoder to map between video frames and latents. The agent policy, simulation model, and rendering model are trained separately and connected through explicit state and action interfaces.

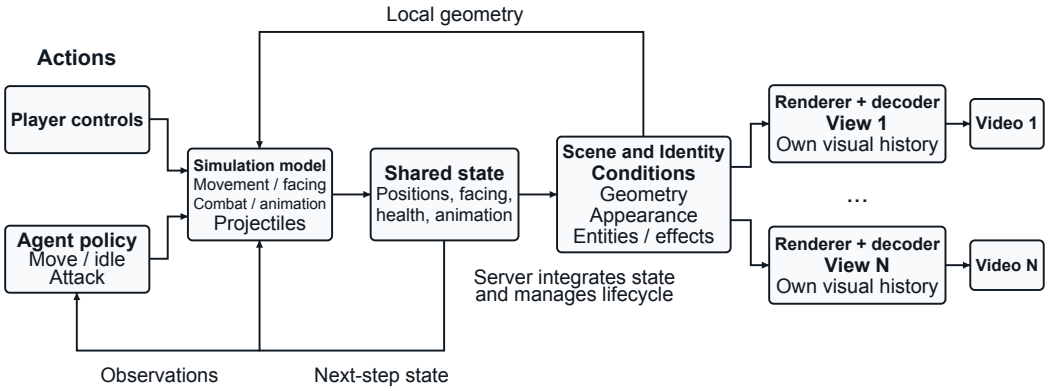

Figure 1: `Agora-2` architecture. Player controls and the agent policy supply actions to the simulation model. The server integrates model predictions into shared state and manages entity lifecycle. Scene and identity conditions combine shared entity and effect states with geometry and appearance information for each view's renderer and decoder. Local geometry also informs the simulation model. Shared state supplies subsequent simulation inputs and agent observations; each rendered view maintains its own visual history.

4.2 Actions

Player controls and a learned agent policy supply actions through the same interface. Each action is associated with the entity it controls, forming the joint action input A_k defined in Section 3. The simulation model predicts the consequences of these actions for the shared world.

Agent policy. Autonomous entities use a learned policy to select movement, idle, or attack actions from local scalar and spatial observations. The policy specifies the intended action, and the simulation model predicts its outcome. Action targets and spatial direction labels provide supervision during training; at inference, the policy relies only on local observations. Separate checkpoints control monsters and autonomous hero companions; the companion also observes human-follow information, while human-controlled heroes use browser inputs. Appendix A.3 describes the policy roles and training settings.

4.3 Simulation

The simulation model is a learned state-transition model: given the current world state, recent motion history, and player and agent actions, it predicts the changes that the world server applies at the next simulation step. It operates on structured states in world coordinates, independently of RGB frames and video latents. A shared transformer trunk processes character tokens jointly, with separate output heads for movement and facing, combat, and animation. Separate multilayer perceptrons (MLPs) predict projectile behavior. Figure 2 shows these two paths from inputs to predicted state changes and server integration.

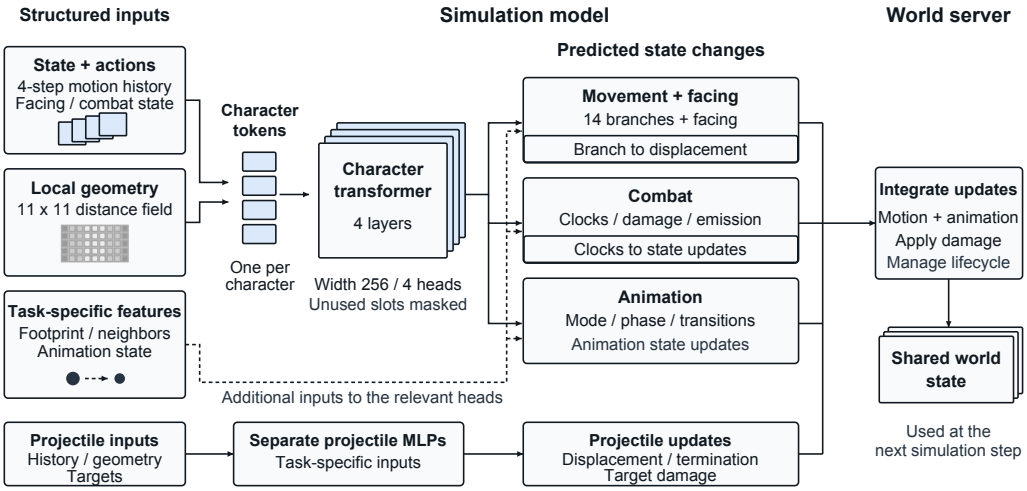

Figure 2: Simulation model. State, action, and local-geometry inputs form one token per character; character tokens share a transformer backbone. Dashed arrows show task-specific features supplied directly to the relevant prediction heads. Character heads predict movement and facing, combat (including projectile emission), and animation. Fixed decoding converts movement-branch and combat-clock choices into state updates; other predictions pass through to the server. Separate projectile MLP branches predict displacement, contact/termination, and target damage. The world server integrates the model outputs and manages entity lifecycle.

Structured inputs. Each character is described by four steps of recent motion history, its facing direction, intended movement, and combat state. Local geometry is represented by an 11×11 grid of distances to obstacle boundaries. Together, state, action, and geometry inputs form one token per character. Task-specific features—including collision footprint, relative positions of nearby characters, and animation state—also feed directly into the relevant output heads, following the dashed paths in Figure 2.

Character transformer. A four-layer transformer processes all character tokens jointly, so predictions for one character can depend on the others. Each layer has a hidden width of 256 and four attention heads; unused character slots are masked. The deployed configuration supports four player-controlled characters and sixteen other characters. The transformer's representations feed three groups of prediction heads, described below.

Movement and facing. The movement head predicts how a character responds to its movement command and surrounding obstacles. It selects among 14 predefined movement branches: six for ordinary motion, including unobstructed movement, sliding, and stopping, and eight escape direc-

tions for recovery from obstacle overlap. A fixed decoding function converts the selected branch into a displacement using the movement command and local geometry. Appendix A.2 describes the branches.

Facing is predicted separately. The model chooses whether to turn, follow the movement direction, retain the current heading, or face a nearby target. Where needed, a second head predicts the turn fraction.

Combat. Combat heads predict changes to three clocks: attack progress, cooldown, and recovery after a hit. Fixed decoding converts the predicted clock choices into state updates. Separate damage heads predict whether damage occurs and how much damage is dealt to nearby targets. Another head predicts whether a character emits a projectile; the projectile branch then predicts its subsequent behavior.

Animation. Animation heads predict which animation plays and how it progresses, including when it advances, pauses, restarts, wraps around, or finishes. These predictions update the character's animation state and are supervised separately from combat timing.

Projectile updates. Projectiles follow the separate MLP path at the bottom of Figure 2, rather than the character transformer. These branches predict projectile displacement, termination on contact with walls or characters, and damage to nearby targets. Displacement uses the projectile's motion history; termination and damage also use relevant geometry and target information. The deployed configuration supports thirty-two projectile slots.

The world server integrates the character and projectile predictions into shared state and manages entity lifecycle, as described in Section 4.4. The updated state supplies inputs for the next simulation step. Together, the simulation components used during deployment contain approximately 4.46 million parameters.

Training objective. The model learns from recorded state transitions. During training, momentum inputs are perturbed with dropout and additive noise; Appendix A.2 gives their settings. The training objective for the components used during deployment sums losses for movement, facing, combat, animation, and projectile behavior:

$$\mathcal{L}_{\text{sim}} = \mathcal{L}_{\text{move}} + \mathcal{L}_{\text{facing}} + \mathcal{L}_{\text{combat}} + \mathcal{L}_{\text{animation}} + \mathcal{L}_{\text{projectile}}. \tag{6}$$

Categorical choices, such as movement branches and animation modes, use cross-entropy. Binary events, such as a hit or projectile emission, use binary cross-entropy. Continuous quantities, such as projectile displacement and damage magnitude, use Smooth L1 losses. Damage magnitudes are supervised only on examples where damage occurs. Appendix A.2 specifies the task weights and supervision masks.

4.4 Shared State

The server applies the simulation model's selected motion and predicted interaction outcomes to the persistent world state. It integrates displacement and orientation, applies predicted damage to health, and updates animation state. It also manages spawning, death, corpse expiry, and respawn according to the session's configured rules. Movement-branch and facing selection belong to the learned simulation model; the server performs state integration and lifecycle management. Section 6.2 details the serving responsibilities.

Players and autonomous agents interact through this common state. For example, predicted damage changes a target's health, and the updated state supplies conditions for subsequent views. All renderers receive a common reference for entity positions, identities, and interaction outcomes while retaining separate visual histories.

Figure 3 illustrates how this shared state conditions two views of a generated encounter. Two scripted player controllers occupy different positions: one approaches and attacks Duriel, while the other moves to an offset observation position. The generated views show the boss before and after defeat, with the resulting corpse and portal visible in both. The shared-world inset relates the two cameras to the same entity positions. Both dynamics and rendering are learned. This is a qualitative example rather than a measurement of cross-view consistency.

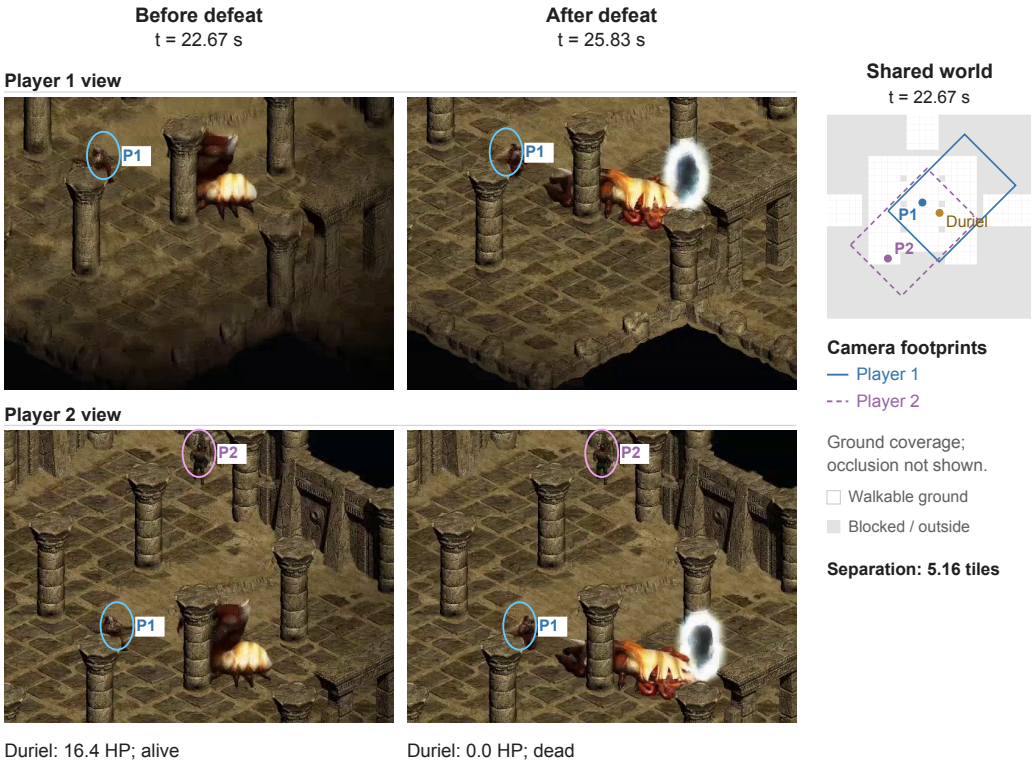

Figure 3: Generated views of a shared boss encounter. Rows show the two players' views and columns show synchronized frames before and after Duriel's defeat. Labeled blue and purple markers identify Player 1 and Player 2; Player 2 is outside the first view. The inset shows recorded world-space positions and camera ground footprints at 22.67 s, over sampled native-map navigation geometry. Solid and dashed outlines denote the two camera footprints. The players are 5.16 tiles apart. Footprints describe viewport ground coverage, not visibility through walls or objects. Both dynamics and rendering are learned; portal creation follows scenario-owned gameplay rules. Camera-pan inputs look toward the boss within the configured tether. The complete generated frames are shown without cropping or brightness adjustments; identity markers, state values, and the inset are annotations.

4.5 Scene and Identity Conditions

Scene geometry and appearance metadata provide persistent conditions, while entity and effect states change as the shared simulation advances. The simulation model reads local geometry as described in Section 4.3; the renderer receives the view-specific conditions described here.

For each pair of output frames, the rendering model receives a structured description of the scene for that view. The condition encoder represents map tiles, entities, and transient effects as separate tokens, together with information about the view and which entity slots are occupied. The resulting 342 conditioning tokens form the *structured prefix*, which preserves individual scene elements for attention.

Input type	Tokens	Information carried
Map	144	A 12×12 local tile grid describing geometry, walkability, and appearance
Entities	36	Four user-controlled and 32 other entity slots, each with ordered states for the two output frames
Transient effects	160	Effect identity and state aligned with the output frames
View action	1	View-related control information
Roster	1	Fractions of occupied entity slots and indicators for empty groups
Total	**342**	Structured scene information for one output frame pair

Each conditioning token includes a learned embedding that identifies its input type: map tile, entity, effect, view, or roster. Unused entity and effect slots are masked so that image tokens cannot attend to them.

Entity representation. Entity tokens encode appearance through categorical attributes and identity labels. They also include animation mode and phase, with separate, ordered states for the first and second output frames. These states describe changes within the frame pair, such as movement or animation progress. Lifecycle information distinguishes an unused slot from a visible corpse. Ability and effect information describes events that position alone cannot capture.

4.6 Renderer and Decoder

The renderer generates each view incrementally from structured scene conditions and a bounded history of that view. It is an approximately one-billion-parameter flow-matching transformer with sixteen blocks of width 2,048. Every block applies spatial attention, and five also apply causal temporal attention to incorporate earlier latents. Figure 4 shows the generation process; Appendix A gives the block layout.

Map tiles, controllable entities, other entities, and effects enter the renderer as individual conditioning tokens. At each spatial attention layer, these tokens supply keys and values alongside the image tokens, while only image tokens supply queries and receive updates. This read-only structured prefix makes scene state available throughout the transformer. Section 4.5 specifies the token layout, and Section 4.6.2 describes the attention operation.

We train the renderer from scratch with a flow-matching objective [Lipman et al., 2023]. The global velocity-prediction loss gives the first latent four times the weight of subsequent latents. An additional entity-region loss, weighted by a coefficient of 2, emphasizes errors within entity regions. Appendix A.1 describes the training schedule, objective, and normalization.

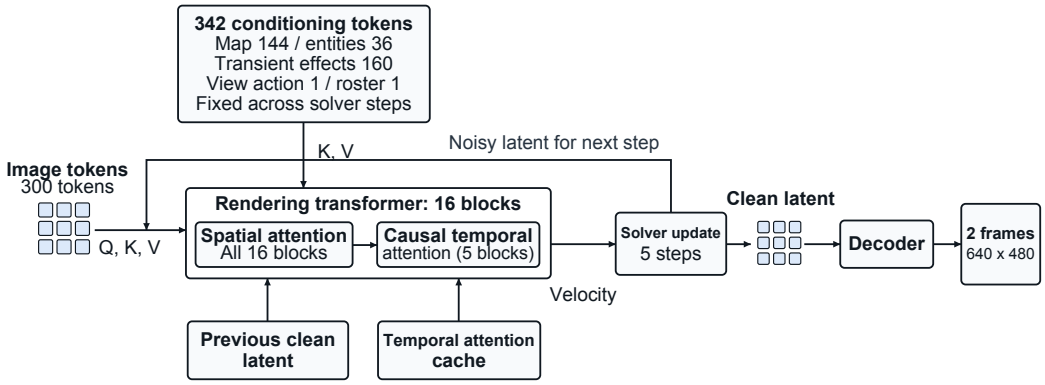

Figure 4: Rendering model. The 300 image tokens attend to 342 conditioning tokens through spatial attention, while per-view history supplies the previous latent and temporal context. Spatial attention appears in all sixteen blocks; causal temporal attention appears in five. The diagram summarizes these attention types rather than the full block sequence. Five solver updates integrate predicted velocity. The uncached implementation evaluates the transformer at each update; the hardware-specific serving profile can reuse intermediate velocity predictions (Appendix A.4). The decoder maps the final clean latent to two video frames. History updates between generated latent frames are omitted.

4.6.1 Latent Representation

The codec uses a video representation autoencoder based on MIRA's architecture [Hu et al., 2026], building on the RAE approach [Zheng et al., 2025]. It compresses two consecutive 640×480 video frames into one 15×20×32 latent frame, giving 300 spatial tokens with 32 channels. Training windows contain forty video frames, or twenty latent positions. Entity and effect conditions are aligned with both frames so that the renderer can depict changes within the pair. Appendix A.1 gives the normalization and training settings.

4.6.2 Attention over Structured State

At each spatial attention layer, the renderer updates the image tokens of a latent frame using both visual features and structured scene information. Each image token can attend to other image tokens and to individual scene elements, such as map tiles, entities, and effects.

We use standard scaled dot-product attention. Image tokens provide the queries; image and conditioning tokens together provide the keys and values. The softmax is unchanged and normalizes attention weights jointly across both groups:

$$Y_x = \mathrm{softmax}\left(\frac{Q_x[K_x; K_s]^\top}{\sqrt{d}} + M\right)[V_x; V_s]. \tag{7}$$

Here, Q_x, K_x, V_x are the image queries, keys, and values, while K_s, V_s are the conditioning keys and values. Concatenation along the token dimension is denoted by $[\cdot;\cdot]$, and d is the attention-head dimension. The mask M prevents attention to unused conditioning slots.

Conditioning tokens supply scene information without being updated by the attention layers. Only image tokens receive updates and pass through temporal attention; conditioning tokens are neither decoded into images nor stored in the temporal attention cache. Two-dimensional rotary position embeddings are applied to image queries and keys, but not to conditioning keys.

Appendix A.1 provides the token counts, masking rules, and an illustration of the attention operation.

4.6.3 Diffusion Forcing and Sequence-Causal Training

Each latent position receives an independently sampled flow time during training. In addition to causal temporal attention, the model receives the clean latent from the preceding position through a shifted predecessor path, with a learned beginning-of-sequence representation at the first position. This separates the clean preceding observation from the noised latent being predicted [Chen et al., 2024].

During training, both visual-history paths are removed together on 80% of samples. Removing both paths prevents the model from recovering preceding images through the remaining path and is intended to encourage use of structured state. When history is retained, the clean predecessor is available alongside the independently noised latent sequence. At inference, generation begins from a learned beginning-of-sequence representation and retains visual history.

4.6.4 Generation and Decoding

Simulation updates collect player controls and agent-policy actions and advance the shared state. Frame-aligned, view-specific conditions then drive each rendering model. Five solver steps generate a latent, which decodes to two video frames. The generated latent is then added to the view's visual history for the next update.

Figure 5 compares learned and native rendering of one recorded engine trajectory. At each timestep, both renderers receive the same entity states, appearance identifiers, animation states, effects, and camera. The learned renderer generates continuously from its learned beginning-of-sequence representation without engine RGB initialization. Matched close-ups show character and environment detail under the same state and camera conditioning. This comparison isolates rendering and does not evaluate learned dynamics.

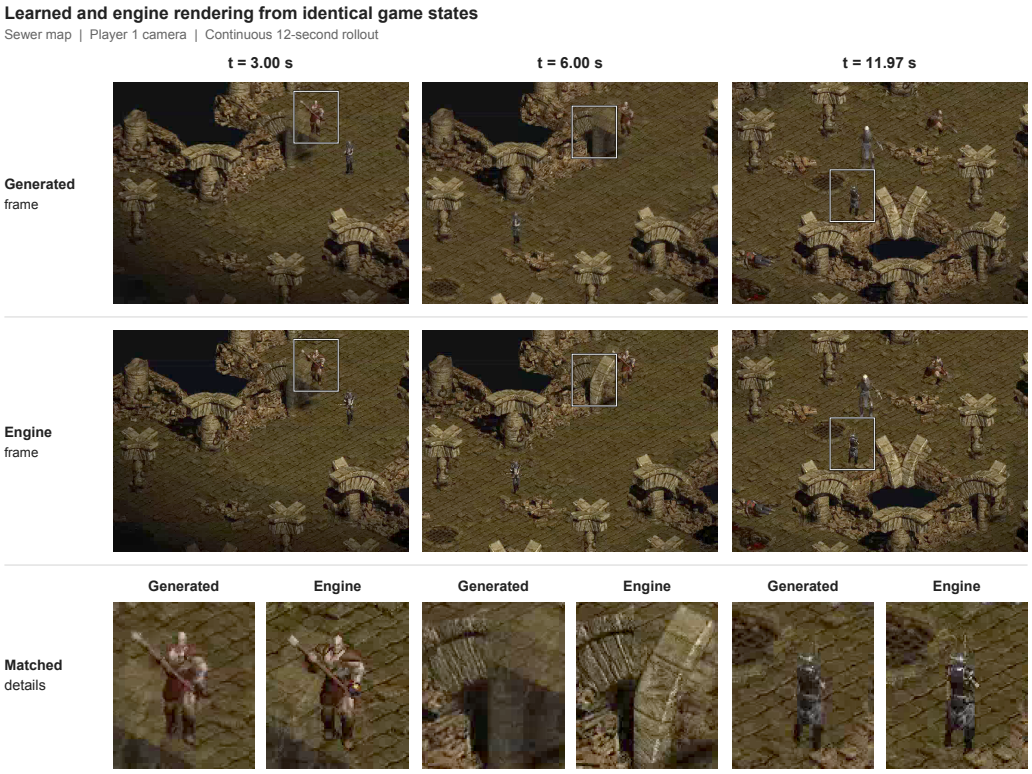

Figure 5: Learned and native rendering of the same engine trajectory. Columns show synchronized moments at 3.00, 6.00, and 11.97 s in a sewer encounter. The top two rows show learned-renderer outputs and native-engine frames. The bottom row compares matched details from the outlined regions, with generated crops on the left and engine crops on the right. Each pair uses identical 96×112 pixel bounds in the original 640×480 frames and nearest-neighbor enlargement, showing character appearance and environment geometry. Both renderers use identical engine states and the Player 1 camera, verified across all 360 frame pairs. For print readability, all full frames and crops use the same display-only gamma correction, $y = x^{0.70}$ for RGB values normalized to $[0, 1]$; source images are unchanged.

5 Data

We collect training data from an instrumented Rust-based isometric game engine that exposes both rendered views and structured world state. The engine provides direct access to scene geometry, entity identities, positions, health, animation, and interaction events, allowing actions and their consequences to be recorded alongside the corresponding video. It serves as the source of training supervision. During interaction, `Agora-2` uses learned simulation and rendering models while retaining native-engine scenario scaffolding, as detailed in Section 6.2.

Collection pipeline. We collect data by running several engine instances in parallel. Each instance generates a procedural scene and executes a configured sequence of movement and combat actions. At each simulation tick, it records the actions, entity states, and interaction outcomes. For rendering data, it also captures each player's view and an instance-ID image identifying the visible entities. Simulation data can be collected without rendering, using the recorded state transitions directly. Map and episode seeds make individual recordings reproducible across collection workers.

Training corpora. The rendering model, simulation model, and agent policy use separate corpora, summarized in Table 1. Procedural layouts span cave, crypt, dungeon, jail, sewer, and tomb themes. Rendering segments contain 60 frames at 30 Hz, from which 40-frame training windows are selected. Multiple views and overlapping windows can come from the same episode, so segment counts do not measure independent trajectories. Evaluation splits must separate source trajectories to avoid leakage between related windows.

State	Tick 18	Tick 30	Tick 34	Tick 80
P position	(43.60, 31.72)	(43.78, 31.54)	(43.78, 31.54)	(43.78, 31.54)
P health	55	51	51	47
E1 health	12	12	6	0
P animation	Walk	Attack	Attack	Idle
E1 animation	Attack	Attack	Hit reaction	Dead
E1 life state	Alive	Alive	Alive	Dead

Figure 6: A close-range native-engine encounter: rendered frames (left) and recorded abstract state (right), ordered from top to bottom. The player moves during ticks 1–24, requests attacks during ticks 25–44, then idles. Enemies follow the game engine's built-in behavior. The selected frames span 2.07 seconds. Player attack damage is set to six for illustration: two hits reduce E1 health from 12 to 6 to 0. The final frame shows its corpse after the death animation. Abstract views show navigation, positions, facing (lines), health (bars), and the dead entity (cross), using the same projection and crop as the video.

Table 1: Data by model. Training counts describe the renderer continuation and final policy stages. Renderer evaluation includes training-time monitoring and the reconstruction comparison; policy evaluation uses separate native-engine rollouts. Units and coverage differ across rows. Appendix B gives the breakdowns.

Model	Training data	Evaluation data	Supervision
Rendering	4,332,800 segments	6 monitoring windows; 30 comparison clips $\times$ 3 seeds	Video and frame-aligned scene state
Simulation	1,108,800 segments	3,456 segments	State and action histories with transition targets
Monster policy	23,771 examples	252 episodes	Local observations, target actions, and spatial labels
Companion policy	128,005 examples	24 cases	Human-follow and combat observations, target actions, and spatial labels

Collection coverage. The choice of actions matters as much as the number of recordings. Ordinary gameplay provides many examples of free movement and successful attacks, but fewer examples of blocked movement, failed attacks, or recovery from obstacles. We therefore use dedicated scenarios to expose these outcomes, including sustained movement against walls, traversal through narrow gaps, and attacks with and without targets in reach. Combat recordings identify which entity damaged which target and by how much, including cases where no damage occurs. Separate collections cover abilities, persistent corpses, and portal transitions, while scenario-specific collection budgets keep these less frequent events represented in the training data.

Aligned state and video. Figure 6 illustrates the recorded information through a short encounter. The player closes in on two enemies and exchanges attacks at close range, taking damage and defeating one enemy with two attacks. The abstract views show navigation geometry and entity positions, facing, health, and life state at the same ticks as the rendered frames; the state excerpt makes the resulting changes explicit. The example was recorded from the native engine for illustration, with player attack damage set to six to make the health change easier to see.

The rendering model receives entity and effect conditions aligned with both frames represented by each latent, with instance masks supporting visual supervision. The simulation model uses recent state and action intent to predict movement and interaction outcomes. The policy uses local observations with target actions and auxiliary spatial labels; route and direction labels provide training supervision only. Appendix B gives the constituent collections and additional data details.

6 System

6.1 Architecture Overview

A browser client sends controls to a world server, which maintains the shared session state and calls the learned models. The agent policy supplies autonomous-entity intent, the simulation model predicts entity updates, and view-specific renderers produce video. The server returns encoded frames and state metadata to each client. Sharing state across views provides a common source for entity position and identity, while independent visual histories allow each view to evolve incrementally.

GPU allocation. A four-player session runs on four NVIDIA RTX PRO 4500 GPUs, with one rendering model per GPU generating each player's view. One of these GPUs also hosts the simulation model and the agent-policy models for the shared world.

6.2 The Interaction Loop

Each update begins with movement and attack inputs from browser clients and actions from the agent policy. The simulation model predicts motion and interaction outcomes from these actions and the recent world state, and uses learned movement-branch and facing selection to determine mo-

tion updates. The server integrates the selected displacement and orientation and applies predicted damage to health. It also manages spawning, death, corpse expiry, and respawn. Predicted attack and cooldown clocks and animation states advance with the model outputs. The deployed configuration disables hit-recovery stagger and the separate incoming-damage magnitude head; health changes instead aggregate predicted per-target melee and projectile damage. View tethering follows session rules, and entity-separation adjustments are configurable.

After the shared state advances, the server assembles conditions for each player's view, and the rendering model generates the next latent. The two decoded frames are encoded and delivered to the corresponding client over WebRTC. Simulation advances on a 30 Hz tick timebase, while the renderer targets 15 latent updates and 30 displayed frames per second per view at 640×480. Appendix A.4 gives the inference and video-encoding settings.

Table 2: Model predictions and their use during interaction. Simulation terminology follows Figure 2.

Model	Learned predictions	Server and client operations
Simulation	**Movement and facing:** movement-branch and facing selection. **Combat:** clocks, damage, and projectile emission. **Animation:** mode and phase. **Projectile updates:** displacement, contact/termination, and target damage.	Decode selected branches; integrate motion; apply clocks, damage, and animation updates; spawn and terminate projectiles.
Rendering	Generate video latents from scene conditions and visual history.	Decode latents into frames; encode, transmit, and display video.
Agent policy	Select actions for autonomous entities.	Combine policy actions with human inputs and associate actions with entities.

Shared-world management. The server also manages health and entity lifecycle, including death, corpse expiry, spawning, respawn, and enabled health-restoration rules. Native-engine scaffolding supplies maps, scenarios, spawning, and scenario-owned world-object effects.

Table 2 summarizes this division of responsibilities. The simulation model predicts motion, attack phase, damage, animation, and projectile updates, and selects movement branches; the server applies selected motion, attack/cooldown clocks, per-target damage, and animation predictions to maintain the shared world state. A native-engine host remains active for map, scenario, and spawn scaffolding and scenario-owned world-object effects. Learned policies supply autonomous intent, and the learned simulation supplies movement and combat predictions. Interaction behavior therefore depends on both the model predictions and the session rules.

6.3 Inference Service

Each renderer retains a bounded visual history and decodes new latents incrementally. Conditioning projections that remain unchanged across solver steps are computed once and reused. Generation starts from a learned beginning-of-sequence representation; visual history is reset on transitions between alive and non-alive states, including respawn, and on camera discontinuities. These resets interrupt visual context while the shared world state persists. Appendix A.4 specifies the caches, compilation strategy, precision, and reset conditions. Appendix A.5 reports the implementation-efficiency benchmark.

6.4 Shared Entity State

The session supports one to four user-controlled entity slots. All views condition on the same world-space entity state, while view conditions and renderer histories remain independent. An entity moving or taking damage changes the shared state supplied to subsequent views. Every renderer therefore receives a common state reference.

6.5 Deployment

A session binds the renderer, codec, simulation model, and agent policy to a consistent configuration. Per-view visual histories and decoder caches persist between requests; shared entity state persists across the session. Reproducible serving therefore depends on both the trained weights and the sampling, history-reset, precision, and caching policies. Appendix A summarizes the model configuration used in this report.

7 Results and Design Analysis

We compare decoded reconstruction between two renderer architectures, compare conditioning-method execution costs, and study the trade-off between visual history and state conditioning in a controlled training experiment. We evaluate the simulation model through single-step movement and animation prediction on recorded examples.

7.1 Rendering Model

Renderer comparison. We evaluate a VAE-based causal video diffusion baseline and our structured-prefix renderer on the same 30 episode-distinct clips, with five clips per theme across six themes and three sampling seeds per clip. The baseline conditions on map tiles, actions, and visual history; our renderer uses a representation autoencoder with structured map, entity, and effect tokens as attention prefixes. Both are scored against identical 20-frame reference windows at 640×480 and 30 fps. Metrics are computed from decoded RGB tensors, with PSNR averaged over frames and clips.

Table 3: Decoded reconstruction on 30 monitoring clips with three sampling seeds each (90 paired evaluations). The comparison evaluates complete systems; architecture is not isolated.

Renderer	RGB MSE ↓	PSNR (dB) ↑
VAE-based baseline	0.010272	20.67
Structured-prefix renderer (ours)	**0.001279**	**30.11**

Our structured-prefix renderer achieves lower RGB MSE and higher PSNR on this collection, improving mean per-frame PSNR by 9.44 dB (Table 3). These results characterize the two complete systems on the final renderer's monitoring collection. Appendix C documents their differing training and evaluation configurations and provides supporting diagnostics.

Conditioning-method inference cost. We benchmark cross-attention, pooled AdaLN, and structured self-attention K/V prefix conditioning on a shared renderer backbone. This controlled execution-cost test uses randomly initialized denoisers and identical synthetic inputs with all entity and effect slots active; it does not measure trained image quality. All variants use BF16, five denoising steps, and the same compiled execution settings on an NVIDIA RTX PRO 6000 Blackwell Server Edition GPU.

Table 4: Conditioning-method execution cost in milliseconds per two-frame update. Means cover three fresh-process repetitions of 100 updates each. Core time sums separately measured world-rollout and decoder means; it is not live end-to-end latency.

Conditioning	Denoiser (ms)	Core (ms)
Cross-attention	37.06	53.71
Pooled AdaLN	18.82	34.08
Structured self-attention K/V prefix	20.09	35.37

Structured-prefix conditioning reduces denoiser time by 45.8% relative to cross-attention, and the world-plus-decoder core sum by 34.1% (Table 4). Pooled AdaLN is slightly faster than structured-prefix conditioning in this workload. Modality encoding and fusion are excluded from timing. Appendix C.5 specifies the protocol and timing boundaries.

Balancing visual history and state conditioning. Visual history supports continuity but can preserve content that conflicts with the current simulation state. We vary the probability of dropping visual history during training in a controlled experiment using an earlier renderer with cross-attention conditioning and an earlier codec. This study varies history dropout within that configuration; it does not compare conditioning architectures.

Four runs start from the same checkpoint and use identical data, training schedules, and 10,000 optimizer updates, with one training seed per setting. Streaming evaluation supplies visual history; cold-start evaluation supplies none.

Table 5: Controlled history-dropout study in an earlier renderer configuration. The map perturbation error ratio measures sensitivity to map conditioning, not the correctness of the response.

History dropout	Streaming latent MSE ↓	Cold-start latent MSE ↓	Map perturbation error ratio
0%	0.06041	0.21082	1.45×
20%	**0.05641**	0.20001	1.50×
50%	0.05695	**0.19535**	1.55×
80%	0.06101	0.19550	1.62×

The map perturbation error ratio compares flow-velocity prediction error under a reflected map with error under the correct map, holding noise and timestep fixed. It measures sensitivity to map conditioning.

Higher history dropout produces greater sensitivity to map conditioning in this experiment. All nonzero settings improve cold-start reconstruction relative to no dropout, while 20% gives the lowest streaming error and 50% and 80% produce similar cold-start errors. These results illustrate a trade-off between temporal context and explicit state conditioning. They do not establish an optimal dropout rate for the final prefix-conditioned renderer.

7.2 Simulation Model

Movement and animation prediction. We evaluate single-step predictions from the deployed simulation checkpoint on 16 batches of 128 held-out recorded examples. Reported metrics are averaged across batches. Movement branch accuracy measures agreement between the predicted discrete movement branch and its recorded target on supervised examples; the blocked subset covers examples labeled as obstructed movement.

Table 6: Single-step movement and animation prediction on held-out recorded examples; these metrics do not measure autoregressive rollout accuracy.

Metric	Result
Movement branch accuracy	85.17%
Movement branch accuracy on blocked examples	68.17%
Animation mode accuracy	95.84%
Predicted animation mode-switch rate	7.49%
Recorded animation mode-switch rate	3.54%

The lower movement accuracy on blocked examples identifies obstruction handling as a harder prediction regime. Despite high per-frame animation accuracy, the model predicts more frequent animation transitions than the recorded simulation. Reporting both mode accuracy and switching frequency captures this difference in temporal behavior.

8 Limitations

State and domain dependence. `Agora-2` requires an instrumented environment with explicit scene geometry, a predefined state schema, and a fixed appearance vocabulary. State not represented by the chosen inputs or finite history can make otherwise identical inputs lead to different outcomes.

Procedural layouts provide variation within the training domain, but transfer to new assets, rules, or environments remains untested.

Simulation errors and renderer input shift. Errors in movement, combat, and animation predictions can accumulate as the simulation advances. The renderer is trained on recorded scene states but receives predicted states during interaction, which may contain unfamiliar entity configurations or inconsistent combinations of motion and animation. Errors in simulation can therefore also affect the generated video.

Consistency across views and time. A shared state provides all views with the same entity identities, positions, and interaction outcomes, but independently generated images can still disagree in appearance, visibility, or event timing. Separate visual histories can retain different errors, while bounded context and history resets can interrupt visual continuity even when the underlying world state persists.

Evaluation scope. Section 7 reports short-horizon rendering reconstruction, a controlled history-dropout study, single-step simulation component accuracy, and conditioning-method execution costs. The renderer comparison uses 30 clips from an existing monitoring collection and three sampling seeds. Appendix C details the complete-system comparison and reports native codec reconstruction and a single-seed denoising-step check on the same 30 clips. Training budgets, data, conditioning, and temporal contexts remain unmatched, and the VAE-based baseline is evaluated outside its camera-only training distribution. These comparisons do not isolate architecture or establish performance at equal compute. The dropout study uses one training seed per setting. Long-horizon visual quality, agreement across views, end-to-end action adherence, and agent-policy performance remain unevaluated in this report. The stated update and display rates are targets; sustained frame rate and input-to-display latency during live multi-agent interaction have not been quantitatively evaluated.

9 Conclusion

We present `Agora-2`, an interactive world model in which up to four human players interact alongside autonomous agents in a persistent shared world. A learned simulation model predicts movement, combat, animation, and projectile behavior, while a rendering model generates each player's view from the resulting state and its own visual history.

Our central design choice is to separate shared simulation from per-player rendering. Human controls and agent actions enter the same simulation, giving their consequences a common representation before they appear in any view. Structured scene tokens connect that state to each renderer, preserving individual entities, geometry, and effects for attention. This lets us inspect and improve predicted world updates independently of their visual depiction, while bringing both together in an interactive system.

Our evaluations show improved decoded reconstruction over a VAE-based baseline in a complete-system comparison, and lower denoiser cost for structured-prefix conditioning than cross-attention.

These results are a starting point. We study a single instrumented game environment with explicit state and geometry; extending the approach will require evaluating longer interactions, more varied agent behavior, and consistency across players' generated views.

We hope `Agora-2` provides a useful baseline for understanding how to develop generalizable algorithms for shared, interactive world modeling. Its separation of simulation and rendering gives us a concrete setting for testing which representations and learning methods can extend beyond this environment, toward learned worlds that people and autonomous agents can inhabit and shape together. The system can be explored at agora.odyssey.systems.

10 Contributors

`Agora-2` was developed by the Odyssey team, with contributors listed alphabetically below.

Core contributors: Ahmad Nazeri, Ali Panju, Aravind Kaimal, James Grieve, Kaiwen Guo, Vinh-Dieu Lam.

Leadership: Jeffrey Hawke, Oliver Cameron.

References

Jake Bruce, Michael Dennis, Ashley Edwards, Jack Parker-Holder, Yuge (Jimmy) Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, Chris Apps, et al. Genie: Generative Interactive Environments. *ICML*, 2024. URL https://arxiv.org/abs/2402.15391.

Ziqi Cai, Siqi Yang, Yimu Wang, Zixian Gao, Yunheng Liu, Shuchen Weng, Erwin Wu, Kaipeng Zhang, and Boxin Shi. MASS: Multiplayer World Models with Authoritative Shared State, 2026. URL https://arxiv.org/abs/2608.06257.

Oliver Cameron. Agora-1: The Multi-Agent World Model. Odyssey blog, May 18, 2026. https://odyssey.systems/introducing-agora-1.

Boyuan Chen, Diego Martí Monsó, Yilun Du, Max Simchowitz, Russ Tedrake, and Vincent Sitzmann. Diffusion Forcing: Next-token Prediction Meets Full-Sequence Diffusion, 2024. URL https://arxiv.org/abs/2407.01392.

Anthony Hu, Václav Volhejn, Adrien Ramanana Rahary, Chris Mulder, Aditya Makkar, Alyx Liao, Amélie Royer, Manu Orsini, Adam Jelley, Eloi Alonso, et al. MIRA: Multiplayer Interactive World Models with Representation Autoencoders, 2026. URL https://arxiv.org/abs/2607.05352. Code: github.com/mira-wm/mira (Apache License 2.0).

Anssi Kanervisto, Dave Bignell, Linda Yilin Wen, et al. World and Human Action Models towards gameplay ideation. *Nature* 638, 656–663, 2025. doi:10.1038/s41586-025-08600-3.

Jiaqi Li, Junshu Tang, Zhiyong Xu, Longhuang Wu, Yuan Zhou, Shuai Shao, Tianbao Yu, Zhiguo Cao, and Qinglin Lu. Hunyuan-GameCraft: High-dynamic Interactive Game Video Generation with Hybrid History Condition, 2025. URL https://arxiv.org/abs/2506.17201.

Yaron Lipman, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. Flow Matching for Generative Modeling. *ICLR*, 2023. URL https://arxiv.org/abs/2210.02747.

Ryan Po, David Junhao Zhang, Amir Hertz, Gordon Wetzstein, Neal Wadhwa, and Nataniel Ruiz. MultiGen: Level-Design for Editable Multiplayer Worlds in Diffusion Game Engines, 2026. URL https://arxiv.org/abs/2603.06679.

Robbyant Team, Zelin Gao, Qiuyu Wang, Yanhong Zeng, et al. Advancing Open-source World Models, 2026. URL https://arxiv.org/abs/2601.20540.

Lloyd Russell, Anthony Hu, Lorenzo Bertoni, George Fedoseev, Jamie Shotton, Elahe Arani, and Gianluca Corrado. GAIA-2: A Controllable Multi-View Generative World Model for Autonomous Driving, 2025. URL https://arxiv.org/abs/2503.20523.

Georgy Savva, Oscar Michel, Daohan Lu, Suppakit Waiwitlikhit, Timothy Meehan, Dhairya Mishra, Srivats Poddar, Jack Lu, and Saining Xie. Solaris: Building a Multiplayer Video World Model in Minecraft, 2026. URL https://arxiv.org/abs/2602.22208.

Boyang Zheng, Nanye Ma, Shengbang Tong, and Saining Xie. Diffusion Transformers with Representation Autoencoders, 2025. URL https://arxiv.org/abs/2510.11690.

A Implementation Configuration

The table summarizes the model and serving configuration. Renderer blocks 0, 4, 8, 12, and 15 add causal temporal attention after spatial attention.

Component	Model configuration	Training or inference setting
Rendering model	16 blocks; width 2,048	20-position training windows; five inference steps
Simulation model	Four layers; width 256; four heads	Four-step history; 14 movement branches; separate facing prediction
Video representation	Retrained RAE; 32 latent channels	Two raw frames per latent; 15×20 spatial grid
Agent policy	Recurrent scalar/spatial policy	16-observation history; four-simulation-tick action interval
Session	One to four user-controlled entities	640×480; target 15 latent updates/s and 30 displayed frames/s
Serving topology	One RTX PRO 4500 per view	Four GPUs for four views; one also hosts simulation and agent-policy models

A.1 Renderer Training

For the twenty-position training windows described in Section 4.6.1, the shifted clean-predecessor sequence is $[\mathrm{BOS}, z_0, \ldots, z_{18}]$. Latent normalization and codec identity are held consistent between cached training data and inference.

The renderer is trained from scratch for 60,000 optimizer updates on 4,232,000 segments, followed by a 60,000-update continuation on 4,332,800 segments with fresh optimizer and schedule state. Training uses AdamW with learning rate 1×10^{-4}, betas $(0.9, 0.99)$, weight decay 0.1, BF16, EMA decay 0.9999, and effective global batch 128 on 32 B200 GPUs. The continuation uses 1,000 warmup steps followed by 59,000 constant-rate steps.

Training objective. We use the linear noise-to-data interpolation and velocity-prediction objective of flow matching [Lipman et al., 2023]. For clean latent z and Gaussian noise ϵ, an independent flow time is sampled at each latent position, following the per-position noise-level training used in diffusion forcing [Chen et al., 2024]:

$$z_\tau = \tau z + (1-\tau)\epsilon, \quad v^* = z - \epsilon, \quad \tau \sim \mathcal{U}[0,1], \tag{8}$$

$$e_t = \mathrm{mean}_{\text{spatial,channel}} \left\| v_\phi(z_{\tau,t}, \tau_t, \text{conditions}) - v^* \right\|^2, \tag{9}$$

$$\mathcal{L}_{\text{global}} = \frac{\sum_t w_t e_t}{\sum_t w_t}, \tag{10}$$

$$\mathcal{L} = \mathcal{L}_{\text{global}} + \lambda_{\text{entity}} \mathcal{L}_{\text{entity}}. \tag{11}$$

Here, v_ϕ predicts the velocity field, w_t weights each latent position, and λ_{entity} weights the additional entity-region loss. The interpolation runs from Gaussian noise at time zero to clean data at time one. Inference integrates the predicted field using the solver settings in Section 4.6.4.

The weights described in Section 4.6 correspond to $w_0 = 4$, $w_{t>0} = 1$, and $\lambda_{\text{entity}} = 2$. The first latent is generated without preceding visual history. Captured instance silhouettes are pooled to latent resolution to define the entity regions, where an additional mean-squared velocity-prediction error term increases their contribution to training. Global and regional errors are reduced in FP32, and the regional term is normalized by weighted mask mass and channel count.

Entity-region supervision complements structured entity conditions and visual-history dropout. Their individual effects on appearance, animation, and continuity are not quantified in this report.

Spatial attention tokens and masking. Table 7 summarizes the query and key/value roles for one latent frame. The additive mask M is $-\infty$ for unused conditioning slots and zero elsewhere. The renderer uses 16 query heads and four key/value heads. Figure 7 illustrates the attention operation.

Table 7: Spatial attention token counts per latent frame. Each image query attends jointly to image and conditioning keys. Section 4.5 gives the composition of the structured conditions.

Token group	Count	Queries	Keys/values	Masking
Image	300	300	300	All spatial positions available
Conditioning tokens	342	0	342	Unused entity and effect slots excluded
Total before masking	**642**	**300**	**642**	

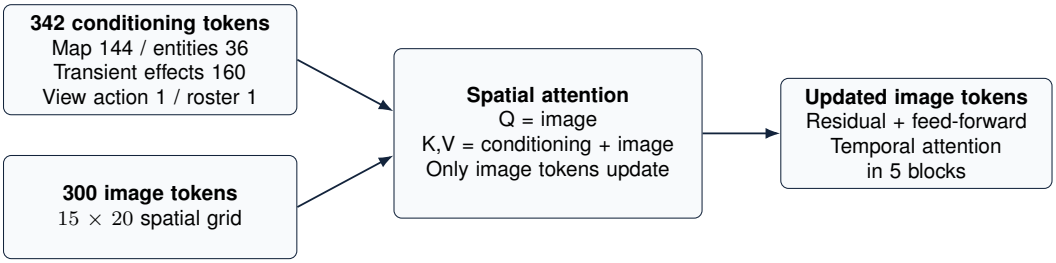

Figure 7: Spatial attention with structured conditioning. Image tokens attend to image and conditioning keys and values; only image tokens are updated. Conditioning keys and values can be computed once for each generated latent and reused across solver steps.

A.2 Simulation Configuration

The settings below describe the deployed simulation model at checkpoint 15,000.

The simulation components used during deployment in Section 4.3 contain 4,457,777 parameters. The character transformer uses pre-normalized layers, GELU activations, and feed-forward width 1,024. Motion-history and local-geometry inputs are standardized. Momentum dropout is 0.25 and additive momentum noise has standard deviation 1.0 in the model's input representation. Movement uses six collision neighbors; aiming and combat use four target neighbors. Projectiles are excluded from the character-target neighbor set.

Movement branches. Six branches describe ordinary movement: the full commanded step, either axis component, either lateral-seeking direction, or stopping. Eight additional branches provide escape directions for recovery from obstacle overlap. A fixed decoding function converts the selected branch into a displacement using the movement command and local geometry.

Movement and facing losses. Ordinary movement branches use cross-entropy against labels derived from commanded and realized motion. Recovery examples can admit multiple tied escape directions; their loss is the negative logarithm of the total predicted probability assigned to those valid directions. Facing uses categorical cross-entropy, with target-slot examples weighted by 10, and a Smooth L1 term for the turn fraction on applicable non-slot examples. Target-slot predictions snap to the selected target direction.

Combat losses. Each of the three clocks uses inverse-frequency-weighted cross-entropy over transition verbs. Binary cross-entropy supervises the took-damage gate and per-target dealt-damage gates, with positive-example weights of 4 and 20, respectively. Dealt-damage magnitudes are normalized by 5 health units and use Smooth L1 with $\beta = 0.5$ on positive examples. Auxiliary geometry supervision and a same-faction penalty further train the melee damage gate. A separate incoming-damage magnitude head is disabled. At inference, the selected checkpoint uses a probability threshold of 0.9 for both melee and projectile dealt-damage gates; these checkpoint-specific thresholds are part of the serving configuration.

Projectile threshold calibration. The projectile hit threshold is calibrated using 128 recorded segments containing 174 readable projectile flights. At the selected threshold of 0.9, matching predicted damage events to recorded projectile-damage outcomes gives 159 true positives, five false positives, and seven false negatives (97.0% precision and 95.8% recall). These measurements describe the calibration set used for threshold selection, not an independent test set.

Animation losses. Animation mode uses class-balanced cross-entropy. Phase-transition verbs use cross-entropy with a mixture of uniform and inverse-frequency class weighting, with mixture coefficient 0.5. An additional penalty discourages resetting an animation in the middle of a clip. Phase advance uses Smooth L1 with $\beta = 0.01$ on smooth-transition examples; looping modes use a circular phase residual.

Projectile losses. Emission and wall/contact termination use binary cross-entropy. Flight uses Smooth L1 with $\beta = 0.01$. Per-target damage uses binary cross-entropy with positive-example weight 20 and Smooth L1 magnitude regression with $\beta = 0.5$, normalized by 5 health units. Magnitudes are supervised only on landed hits. Flight and termination losses mask inactive projectiles and windows spanning a change in projectile identity.

The objective sums these task losses with unit outer weights in this configuration; class weights and auxiliary terms are applied within each task.

A.3 Agent-Policy Configuration

The agent policy combines scalar and spatial observations with a windowed gated recurrent unit (GRU) and a learned dense movement field. Its action interface contains eight movement directions, idle, and attack. The selected readout retains the actor's idle/attack logits while using the learned field to select movement. Critic and optimizer state are not required for inference.

The monster policy uses a body-aware movement representation. Its final training stage emphasizes recovery while retaining previously learned behavior, using actor learning rate 1×10^{-5}. Planning parameters remain trainable alongside the movement readout.

Autonomous hero companions use a separately trained policy. Its observations include information about the human player to follow. Human-controlled hero slots bypass the companion policy. Both roles supply actions to the shared simulation through the same action interface.

A.4 Inference and Streaming Settings

Each renderer maintains a three-latent ring decoder cache and a fixed-shape temporal attention cache. Structured key/value projections and diffusion-time normalization terms are reused across solver steps when unchanged. The BF16 serving configuration uses no quantization. The RTX PRO 4500 profile retains BF16 loading and autocast while enabling selective MXFP8 linear operations in the denoiser and decoder. The full denoiser and the ring decoder are compiled as separate graphs; the fixed-shape history cache allows the same compiled graphs to be used as the number of valid history entries changes.

The history, action, and controlled-entity guidance scales are all set to 1.0 at inference. The uncached implementation uses one conditional network evaluation per solver update, with no classifier-free guidance adjustment. The RTX PRO 4500 profile enables TeaCache with threshold 0.15: eligible intermediate updates can reuse velocity predictions, while the final cache-producing evaluation is computed afresh. The solver still performs five updates.

Visual history resets on transitions between alive and non-alive player states, including respawn, and on camera discontinuities exceeding 4.0 tiles.

Output frames use H.264 intra-only encoding: x264 CRF 18 in the BF16 serving configuration and NVENC constant QP 18 in the RTX PRO 4500 profile. Section 6 describes the interaction pipeline and target cadence.

A.5 Renderer Benchmark Details

This implementation-efficiency benchmark measures two-frame calls on an NVIDIA RTX PRO 4500 Blackwell Server Edition GPU at a 165 W power limit. Both implementations use the final checkpoint, 640×480 output, five denoising steps, full visual history, MXFP8 world-model and decoder execution, and TeaCache threshold 0.15. The unoptimized implementation disables timestep-dependent modulation reuse, compiled temporal-cache updates, and grouped decoder attention operations. All three are enabled in the optimized implementation.

Each repetition measures 3,200 calls per implementation. Mean times are 63.30 ms versus 57.03 ms in the first repetition and 62.54 ms versus 56.35 ms in the second. Averaging the repetitions gives 62.92 ms versus 56.69 ms, a 9.9% reduction. A separate comparison across 1,100 frames yields mean per-call RGB RMSE of 0.00418 relative to the unoptimized implementation. This numerical-output comparison is not a perceptual-quality benchmark; the timing measurements cover renderer calls rather than the complete interactive system.

A.6 Timing and Information Flow

Figure 8 distinguishes simulation updates, latent predictions, and output frames. Simulation uses a 30 Hz tick timebase (33.3 ms per tick), also used by the native-engine recording in Figure 6. Rendering targets 15 latent updates per second, with two frames per latent for a target display rate of 30 FPS. Simulation and rendering are distinct streams; encoding, buffering, scheduling, and transport contribute to input-to-display delay.

The agent policy selects an action every four simulation ticks and holds it between decisions. This corresponds to 133.3 ms, or 7.5 decisions per second, in simulation time; it is not a measured wall-clock inference rate. Entity and effect conditions are aligned with the two output frames. The serving code records simulation states on the display-frame timebase rather than requiring interpolation between a single pair of session endpoints.

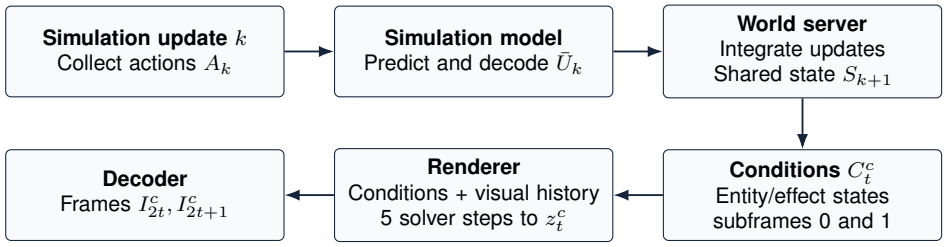

Figure 8: Timing and information-flow contract. Simulation index k and latent index t identify distinct streams. The two subframe states must align with the two decoded frames. Nominal intervals describe target cadence, not measured latency.

B Dataset Details

Section 5 summarizes the collection process and corpus sizes.

B.1 Rendering Dataset

Cached encodings are stored for frame pairs starting at both even and odd frame indices, so each selected window uses consecutive-frame pairs with the correct alignment.

Sampling is proportional to the number of segments in each constituent collection. Initial renderer training uses the first four collections below (4,232,000 segments). The continuation adds 100,800 boss-portal lifecycle segments, yielding 4,332,800 segments; the two stages share the initial corpus and their totals are not additive.

Data regime	Segments	Purpose
Full-roster combat	1,200,000	Controlled and autonomous entities, combat and special content
Stratified special abilities	2,016,000	Concentrated ability and effect coverage
Corpse traversal	504,000	Persistent bodies during view or controlled-entity movement
Movement without autonomous entities	512,000	Scenes without autonomous entities
Boss-portal lifecycle	100,800	Portal appearance, persistence and disappearance
Total	**4,332,800**	Continuation rendering corpus

The renderer's training-time evaluation selects six fixed windows, one per theme, from the full-roster monitoring collection. These windows provide qualitative rollout monitoring; they do not constitute a broad held-out quality benchmark.

B.2 Simulation and Policy Dataset Notes

Simulation. Beyond the collection priorities in Section 5, simulation recordings include pursuit, dense combat, ranged interactions, and undirected actions for controlled and autonomous entities. Damage dealt to other entities is recorded throughout collection. Ranged examples provide training coverage; their inclusion does not establish the accuracy of ranged interactions during deployment.

Policy training. Policy examples are labeled observation–action decisions with spatial supervision, rather than video segments. The monster policy's final stage uses 23,771 examples: 15,245 base-teacher examples, 2,823 recovery examples, and 5,703 retention examples. The companion's final stage uses 128,005 examples across five archives: 23,771 examples retained from the monster curriculum, 88,807 mixed following/combat examples, and 15,427 recovery examples. These counts describe the data loaded for the final stages, not cumulative data across their warm-start lineages. The policy corpora overlap and their totals should not be added. Appendix A.3 describes the policy configurations.

Policy evaluation. The monster panel contains 252 fresh stationary-target episodes: two seeds for each of 21 identities across six themes. The companion panels contain 24 fresh cases: eight following, four mixed-combat, and twelve long-distance recovery cases. Cases can contain multiple bots; the twelve recovery cases contain twenty evaluated bots. These counts are per evaluated policy, excluding repeated runs of comparison policies. Both evaluations use native-engine dynamics and measure policy behavior, rather than the quality of the complete learned simulation and rendering system.

C Supplemental Rendering Evaluation

C.1 Evaluation Protocol

We select 30 episode-distinct clips from the final renderer's existing held-out monitoring collection, with five clips per theme across six themes. Selection uses deterministic hash ranking before inference. The collection was used for model monitoring and is not an untouched benchmark. Both renderers are scored against identical 20-frame reference windows at 640×480 and 30 fps. We compute RGB MSE from decoded tensors in $[0, 1]$, and PSNR per frame with peak value one, then average over frames and equally over clips. Scoring applies no resizing or additional clipping. Reference tensors pass byte-level equality checks across paired evaluations.

C.2 Descriptive Complete-System Comparison

The VAE-based causal video diffusion baseline conditions on map tiles, actions, and visual history. Our renderer uses a representation autoencoder and structured map, entity, and effect tokens as attention prefixes. We evaluate three sampling seeds for each selected clip, giving 90 paired evaluations with no exclusions or substitutions.

Table 8: Training and evaluation settings for the renderer comparison in Table 3. Optimizer updates include the full training lineage.

Renderer	Optimizer updates	History frames	Denoising steps
VAE-based baseline	20,000	13	5
Structured-prefix renderer (ours)	120,000	38	10

These results compare complete trained systems and do not isolate the contribution of architecture. Training data and conditioning also differ; the monitoring collection's full-roster combat scenes represent a distribution shift from the baseline's camera-only training data.

C.3 Denoising-Step Sensitivity

We evaluate our renderer with five and ten denoising steps on the same 30 clips and one sampling seed, holding model weights, conditioning, visual history, precision, and cache-update noise fixed. Only the denoising-step count changes. The ten-step outputs are reused from the complete-system comparison.

Table 9: Within-model denoising-step sensitivity on 30 clips with one sampling seed per clip. PSNR is averaged over frames and clips.

Denoising steps	RGB MSE ↓	PSNR (dB) ↑
5	0.001294	30.12
10	0.001319	30.08

Five and ten steps give similar aggregate reconstruction scores on this collection (Table 9), with a paired mean PSNR difference of 0.047 dB in favor of five steps. This single-seed check does not establish equivalence across seeds, long-horizon quality, or a measured runtime improvement. It does not remove the other differences between systems; equal denoising-step counts across models would not imply equal inference FLOPs.

C.4 Native Codec Reconstruction

To characterize the representations independently of generated rollouts, we encode and decode the real reference sequences with each system's codec. Each codec reconstructs all 30 selected clips once, and both are scored on the same 20 target frames used above.

Table 10: Native codec reconstruction on 30 clips. These are reconstructions of real frames, not generated predictions. Native temporal contexts differ.

Codec	RGB MSE ↓	PSNR (dB) ↑
VAE-based baseline codec	0.001768	28.13
Representation autoencoder (ours)	0.000743	31.66

The codecs have different native temporal contracts. The baseline accepts sequences of length $1 + 4k$; its reconstruction input contains 13 history frames and the 20 target frames. Our representation autoencoder accepts two-frame blocks and receives 38 history frames plus the same 20 targets. We preserve these native contexts without padding or fabricating frames. Both frames in each representation-autoencoder block share a latent, so reconstruction of the first frame can use information from the second.

Codec reconstruction is a native-pipeline diagnostic, not a matched-history ablation, a ceiling on generation quality, or a basis for subtracting codec error from prediction error.

C.5 Conditioning-Method Timing Protocol

The timing comparison in Table 4 holds the renderer backbone fixed at 16 layers, width 2,048, 16 query heads and four key/value heads, with 32-channel latent frames of size 15×20. Only the conditioning-fusion mechanism and its associated modules change. Cross-attention uses 1.547 billion denoiser parameters; pooled AdaLN and structured-prefix conditioning each use 1.177 billion. These native module-count differences are part of the execution-cost comparison.

All variants receive identical seeded synthetic visual-history, action, map, entity, entity-subframe, and effect tensors, verified by hashes. Each latent timestep includes 144 map tokens, 36 active entities and 160 active effects. Native modality fusion produces one pooled summary for AdaLN or 342 structured prefix tokens; cross-attention retains separate modality inputs. Inputs are pre-encoded; modality fusion runs outside the timed region. Denoisers are randomly initialized with a fixed seed; the shared decoder uses the trained representation autoencoder. This dense-capacity workload measures execution cost rather than scene-weighted latency or trained reconstruction quality.

Measurements use batch size one, 640×480 output, 19 latent history frames (38 raw frames), five denoising steps, BF16 parameters and autocast, and final-solver cache reuse. Model and decoder use `torch.compile` in default mode and the same SDPA backend priority (FlashAttention, cuDNN, efficient, math). The decoder uses a three-position ring cache. Exact affine and context-key/value precomputation are enabled where supported; TeaCache and quantization are disabled. The GPU is an NVIDIA RTX PRO 6000 Blackwell Server Edition (96 GB, 600 W limit), with PyTorch 2.9.1, CUDA 13.0 and driver 595.91.07. These settings and hardware differ from the implementation-efficiency benchmark in Appendix A.5.

Each variant runs in three fresh processes, with order rotated across repetitions. Each process performs 20 warmup updates followed by 100 measured two-frame updates. CUDA events measure denoiser, world-rollout and decoder durations; compilation, startup and visual-history prefill are excluded. World rollout includes integration and cache bookkeeping beyond denoiser forwards. Core time is the sum of separately measured world-rollout and decoder means, with no overlap modeled; it excludes raw-state encoding, modality fusion, RPC, display and video encoding. We average the three repetition means rather than treating consecutive updates as independent trials.

Table 11: Conditioning benchmark component means and the range of core means across three repetitions, in milliseconds per two-frame update.

Conditioning	World rollout	Decoder	Core range
Cross-attention	39.15	14.56	53.29–54.05
Pooled AdaLN	19.69	14.39	33.84–34.45
Structured self-attention K/V prefix	21.07	14.30	35.12–35.72