MIFE: A Multimodal VR Immersive Training System for Fire Escape

1 Introduction

In this supplemental document, we provide more implementation details of MIFE.

2 More Implementation Details

2.1 Heat

The system obtains real-time temperature data from the fire spread model. We define a composite distance that jointly accounts for geometric proximity and flame intensity.

$$D_c = \min_{i \in B} \left(\frac{\|\mathbf{u} - \mathbf{v}_i\|_2}{\phi_i + \varepsilon} \right) \cdot \delta, \tag{1}$$

where $\mathbf{u} \in \mathbb{R}^3$ denotes the user position, B denotes the set of burning voxels, $\mathbf{v}_i$ denotes the center coordinate of voxel i, $\phi_i \in [0,1]$ denotes the normalized firepower intensity, ε is a small stability constant, and δ is the voxel size.

The haptic temperature is then computed by mapping D_c to a bounded, distance-decaying response with near-field saturation and far-field baseline.

$$T_{fb} = \begin{cases} T_{\max}, & D_c \le d_{\text{near}} \\ \max\left(T_{\text{base}},\ T_{\max} - \left|\frac{D_c - d_{\text{near}}}{\lambda}\right|\right), & d_{\text{near}} < D_c \le d_{\text{far}}, \\ T_{\text{base}}, & D_c > d_{\text{far}} \end{cases} \tag{2}$$

where T_{fb} is the output temperature for haptic feedback, $T_{\max}$ is the safety-capped upper bound, T_{base} is the baseline temperature, d_{near} and d_{far} specify the effective range, and λ controls the decay step size.

In our implementation, we set $\delta = 0.2$ m, $d_{\text{near}} = 5$, $d_{\text{far}} = 65$, $T_{\max} = 45^\circ$C, $T_{\text{base}} = 25^\circ$C, and $\lambda = 3$.

2.2 Olfactory

The system calculates odor concentration values based on the trainee's current location. We model the odor concentration as a distance-dependent signal derived from D_c as defined in Eq. (1), linearly decaying to zero at a cutoff distance.

$$O_{conc} = \max\left(0, 1 - \frac{\min(D_c, d_{cutoff})}{d_{cutoff}}\right), \tag{3}$$

where $O_{conc} \in [0,1]$ is the normalized odor concentration and d_{cutoff} is the maximum effective range.

The concentration is then mapped to the actuator commands, including the atomizer spray count and the ventilation fan speed.

$$N_{\text{s}} = \begin{cases} 0, & \text{if } O_{\text{conc}} \le 0 \\ \min\left(N_{\text{limit}},\ \max\left(1,\ \lfloor O_{\text{conc}} \cdot N_{\text{limit}} \rfloor\right)\right), & \text{if } O_{\text{conc}} > 0 \end{cases}, \tag{4}$$

$$S_{fan} = \lfloor O_{conc} \cdot S_{max} \rceil, \tag{5}$$

where $\lfloor \cdot \rceil$ denotes rounding to the nearest integer, N_{limit} is the maximum N_{s} per cycle, and S_{max} is the maximum fan speed (PWM value).

In our implementation, we set $d_{cutoff} = 90$, $N_{limit} = 3$, and $S_{max} = 255$.

2.3 Full-body Motion Capture

On the PC side, the system initializes the RGB camera device using the OpenCV library, with acquisition parameters configured to a resolution of 1900×1080. The video stream undergoes real-time preprocessing, which includes cropping the original frames to focus on the human body region, converting the color space from BGR to RGB, resizing the image to the model input size of 512×512, and normalizing the pixel values. The preprocessed image data are subsequently transferred from host memory to GPU memory.

In the inference module, to enhance real-time performance, the system converts the open-source TokenHMR and HAMER models into Open Neural Network Exchange (ONNX) format for cross-platform compatibility and further optimizes them into TensorRT engines to maximize GPU parallel computing. At runtime, the system loads the pre-compiled TensorRT engines and executes the TokenHMR and HAMER models synchronously to estimate full-body pose and shape parameters, as well as hand pose information.

After inference, the system employs a distributed transmission architecture based on the WebSocket protocol. The PC acts as the server distributing serialized JSON posture data, while the VR headset functions as a lightweight client, receiving and parsing the data stream in real time to drive the virtual character's movements. This asynchronous, non-blocking architecture supports high-concurrency, millisecond-level data transmission, ensuring real-time synchronization.

2.4 Hand Tracking

The HTC VIVE Focus Vision headset provides hand gesture tracking functionality on the Android platform. Utilizing the official Wave SDK, we obtain the local rotation data of hand joints in real time. For each joint, the rotational transformations defined in the paper (Section 3.2) are applied successively to derive the updated global rotations. These computed rotations are then used to drive the corresponding hand movements of the character model.

2.5 Fire Spread Model

The simulation framework is architected as a high-performance, parallelized 3D Cellular Automata system implemented using PyTorch tensors to leverage GPU acceleration for massive parallel processing. The input stage initializes the simulation environment by voxelizing complex 3D scene geometry into a dense tensor grid, where each voxel encapsulates a dynamic state vector comprising material properties, temperature (T), fuel mass (F), and a discrete status identifier (e.g., non-flammable, potential fuel, active burning, or burnt). Unlike traditional sequential CA implementations, this system utilizes high-dimensional tensor operations to map material attributes—such as thermal conductivity, ignition thresholds, and burn rates—directly onto the spatial grid. This data-parallel design allows the simulation to accept voxelized semantic scene data as input and maintain a continuously updated global state matrix, ensuring computational efficiency even in high-resolution volumetric environments.

The core temporal evolution is governed by a discrete update mechanism that couples thermodynamic principles with stochastic state transitions through vectorized boolean masking. At each simulation step, thermal conduction is computed via a convolution-like neighbor-accumulation process, utilizing tensor rolling operations to aggregate temperature gradients from the local neighborhood without explicit iteration. Simultaneously, combustion dynamics are resolved for cells identified by active state masks: internal heat generation is derived from material-specific burn rates and fuel availability, while fuel mass is depleted continuously based on thermodynamic intensity. State transitions - specifically ignition and extinction - are determined by evaluating these continuous variables against physical thresholds (e.g., $T > \text{Tignition}$) and stochastic probability factors. The cycle concludes by regulating temperature limits and updating the discrete state tensor, outputting a coherent snapshot of fire propagation, fuel consumption, and thermal distribution for the subsequent time step.

2.6 Knowledge Recommendation

The language model (LLM) component is deployed on the Coze platform as a set of conversational agents for different tasks, with DeepSeek serving as the underlying foundational model for each task agent.

The backend, implemented as a lightweight orchestration layer using FastAPI and Pydantic, exposes RESTful endpoints to collect structured user context and VR training data, and manages all interactions with the large language model agents. Specifically, user attributes (e.g., age, gender, occupation, prior VR usage, and training experience) are submitted via the '/user/info' endpoint and stored in an in-memory session container, while VR performance feedback is incrementally accumulated through '/vr-performance' and subsequently used for downstream generation requests. When a generation task is triggered (initial assessment quiz generation, post-training targeted quiz generation, improvement suggestion generation, or free-form query), the server serializes the relevant context into a UTF-8 JSON string and forwards it to the corresponding Coze bot using the official Coze Python SDK (cozepy).

To enhance responsiveness, we utilize Coze's streaming chat interface and incrementally aggregate tokens on the server side into a complete response. For reliable integration with the VR frontend, each agent is constrained to return a machine-readable JSON payload with rule-based validation; the backend enforces this by extracting the outermost JSON block from the streaming text and decoding it into a typed object validated against a predefined schema.