SoftwareX 32 (2025) 102421

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

Observer: An open-source framework for automating spectator for Real-time Strategy game of StarCraft

Cheong-mok Bae [a], Ho-Taek Joo [a], SungHa Lee [b], Kyung-Joong Kim [a,b,*]

[a] School of Integrated Technology, Gwangju Institute of Science and Technology, 123 Cheomdan-gwagi-ro, Buk-gu, 61005, Gwangju, South Korea
[b] AI Graduate School, Gwangju Institute of Science and Technology, 123 Cheomdan-gwagi-ro, Buk-gu, 61005, Gwangju, South Korea

ARTICLE INFO

Keywords:
Esports
Spectators
Automatic camera control
Game AI
Human behavior modeling

ABSTRACT

Selecting engaging scenes is a critical component of esports broadcasting, traditionally performed by human observers. While recent research has explored AI-based automation, existing approaches often lack comprehensive frameworks for data extraction, human behavior modeling, and interface integration. We present **Observer**, an open-source framework that collects and preprocesses raw in-game data from StarCraft along with human observer viewport data to train AI-based automatic observers. The system transforms gameplay into multi-channel (hereafter, feature channels) representations and uses a modified Intersection over Union (IoU) metric to evaluate the overlap between predicted and aggregated human viewports. As reported in prior work, a learned observer achieves 56.9% similarity to human behavior, surpassing representative rule-based methods (52.4% and 49.1%) on standard benchmarks. In this software paper, we focus on a standardized, reproducible pipeline and system-level metrics.

Current code version	v0.9.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-25-00542
Permanent link to Reproducible Capsule	
Legal Code License	MIT License
Code versioning system used	git
Software code languages, tools, and services used	C++, Python, Shell, Docker
Compilation requirements, operating environments & dependencies	Windows 10/11, Linux (Ubuntu 22.04 LTS)
If available Link to developer documentation/manual	https://github.com/baecm/observer-docker/blob/0.9.1/README.md https://github.com/baecm/Observer/blob/0.9.1/README.md
Support email for questions	cmbae0307@gm.gist.ac.kr

1. Motivation and significance

The esports industry has grown rapidly [1,2], supported by fast-paced gameplay that attracts a wide audience. Its inclusion in the Asian Games reflects increasing popularity and cultural impact. Esports combines strategy, skill, and real-time action. Core titles feature combat, strategic depth, and team coordination [3–5].

Real-time strategy (RTS) games are a key genre in esports, requiring fast decisions and complex planning. Unlike turn-based games, RTS titles require players to respond in real time to evolving situations. Players must simultaneously manage resources, build structures, and control units, making gameplay both strategically deep and mechanically demanding [6].

Among RTS games, StarCraft (Blizzard Entertainment, 1998) is a widely studied benchmark. It features three factions — Terran, Protoss, and Zerg — with distinct units and tactics. Success depends on both macro-management (e.g., economy and expansion) and micro-management (e.g., precise unit control), producing a dynamic strategic

* Corresponding author.
E-mail addresses: cmbae0307@gm.gist.ac.kr (C.-m. Bae), hotaek87@gm.gist.ac.kr (H.-T. Joo), shlee0414@gm.gist.ac.kr (S. Lee), kjkim@gist.ac.kr (K.-J. Kim).

https://doi.org/10.1016/j.softx.2025.102421
Received 29 July 2025; Received in revised form 10 October 2025; Accepted 21 October 2025
Available online 4 November 2025
2352-7110/© 2025 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

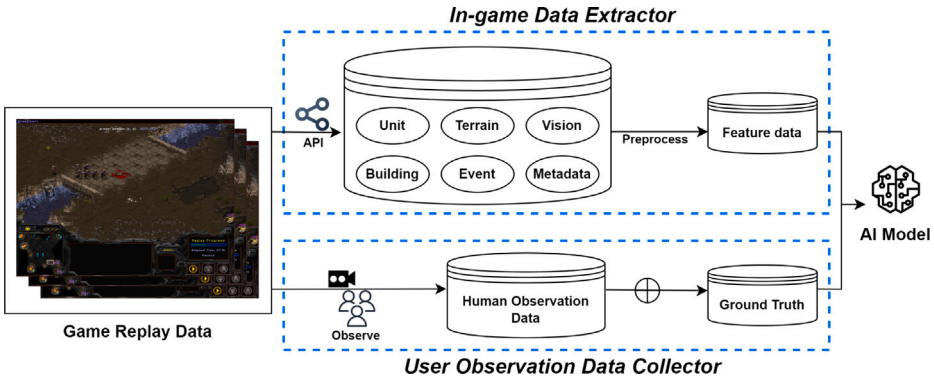

Fig. 1. Overall framework.

environment in which both short- and long-term planning matter [7–11].

This complexity poses challenges for match observation. Unlike many traditional sports, key events in StarCraft can occur concurrently and may be subtle or off-screen. Battles, resource shifts, and strategic movements often unfold in parallel across the map, requiring real-time prioritization. Human observers must track multiple evolving game states, creating a risk of missed or misjudged highlights.

Traditionally, observers manually select scenes for broadcast, but this process is subjective and varies with expertise. Training skilled observers is time-consuming and costly, which can be impractical for smaller tournaments [12]. To mitigate these issues, automated systems have been developed to detect engagements, track resources, and identify strategic changes in real time. These systems may provide more consistent and cost-efficient coverage [12].

Earlier automation methods [12–14] relied on manually defined event rules, which can lead to inconsistent performance under novel tactics or simultaneous events [15]. More recently, learning-based approaches have been proposed that imitate human observer behavior without predefined criteria [16]. However, training and evaluating such approaches require a robust and *standardized* pipeline to extract game data, transform it into analysis-ready features, and synchronize observation labels (e.g., camera focus). Despite this need, the field lacks widely adopted standards and reference implementations that support reproducibility and fair comparison.

We present **Observer**, an open-source framework that extracts feature channels, captures human observer actions, and transforms raw data into *standardized* formats suitable for AI applications (Fig. 1). To support reproducibility, we provide a minimal working example that processes a single replay end-to-end, samples of the dataset and collection scripts (subject to privacy and license constraints), and documentation of *pipeline-level metrics*—including extraction throughput, compression ratios, preprocessing failure/coverage rates, and end-to-end latency—so that practicality can be assessed independently of any specific model. While the current release focuses on StarCraft: Brood War 1.16.1 due to available tooling and public replays, the framework isolates game-specific components (APIs, coordinate systems, and entity taxonomies), facilitating extension to StarCraft II and other RTS titles with limited changes to downstream components.

We assume that the observer's viewport provides a reasonable proxy for regions of interest during broadcast; that replay- and API-exposed states are sufficient for extraction and supervision; and that terrain is static while per-player vision is combined by logical OR under the fixed viewport size in StarCraft: Brood War 1.16.1. Spatial conventions (pixels vs. tiles) are summarized in Section 2.1.

Overall, **Observer** supports more consistent coverage while promoting *standardization* and *reproducibility* in observer research. By reporting system properties alongside model outcomes, the framework provides a basis for replication, fair benchmarking, and deployment planning.

2. Software description

The **Observer** system comprises two core modules: in-game data collection and user observation tracking, both implemented using BWAPI (Brood War Application Programming Interface).[1] [2]

In StarCraft: Brood War 1.16.1, replays store player actions and game state updates. The in-game data module logs positions of all units and buildings (entities) along with terrain layout.

The observation module tracks the area viewed by the observer during replay playback. Replays serve as pre-recorded matches for review or analysis.

Both collectors are distributed as Docker images based on `sc-docker`, which runs StarCraft via Wine[3] and supports BWAPI-based bots [17].

After collection, Python scripts preprocess data for machine learning models that predict observation points.

Fig. 1 outlines the Observer framework's three stages: data collection, preprocessing, and model training. First, it extracts in-game data and human viewports from replays. Next, the data are converted into feature channels representing entities, terrain, and vision. Finally, these inputs train an AI model to predict human-like camera behavior.

2.1. In-game data extractor

This module is implemented in C++ using BWAPI's `AIModule`, functioning as an AI player with access to most in-game information visible to humans. It detects standard events such as entity creation, destruction, and resource use.

We modified the module to extract and store relevant data using key API methods,[4] which are invoked in real time. All logs are exported in text format for further processing.

Spatial conventions. We use two spatial units: pixels for entity positions and bounding boxes, and tiles for terrain and vision. Here, the term "vision" refers to the player's in-game field of view rather than conventional computer vision. One tile corresponds to 32×32 pixels. Coordinates use a top-left origin (x increases rightward, y downward). Unless stated otherwise, entities and events are logged in pixels, whereas terrain elevation and visibility are indexed by tiles. This convention applies throughout.

[1] https://github.com/bwapi/bwapi
[2] https://bwapi.github.io/
[3] https://www.winehq.org/
[4] https://bwapi.github.io/class_b_w_a_p_i_1_1_a_i_module.html

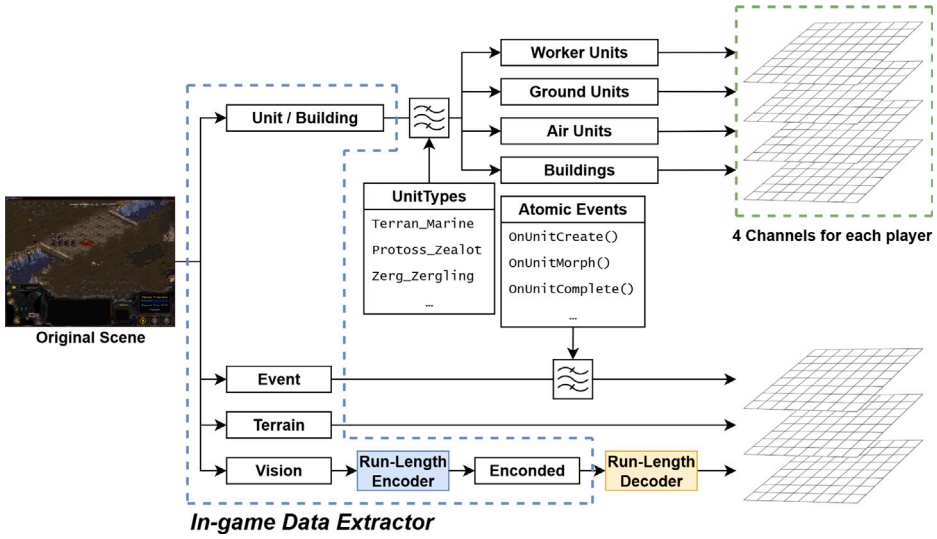

Fig. 2. The comprehensive procedure manages different types of data and consolidates them into a feature-channel format. The processes enclosed within the blue dashed boundary are performed by the *in-game data extractor*, whereas the subsequent steps are handled by the *preprocessor*. The icon with interlaced wavy lines denotes data filtering, categorization, and aggregation.

2.1.1. Metadata

Metadata includes replay-level information such as map name, dimensions (width, height), game length, and player descriptors (anonymized names, races, start locations). The term "map" consistently refers to the in-game battlefield, encompassing terrain and layout. See Appendix A for the data formats and examples.

2.1.2. Entity data

This data records all entities per frame. Recording can occur every frame or at configurable intervals. StarCraft: Brood War replays advance at approximately 24 frames per second (FPS).

Each entry includes the frame number, player info (player name, race), entity name, entity ID, position, bounding box (top/bottom/left/right), and stat fields (HP, shield, energy).

`frame` represents the elapsed game frame number at which the data was recorded, `player` indicates which player the entity belongs to (neutral entities or resources are categorized as `neutral`), `race` indicates the race of the entity, `player color` represents the color the player is displayed as in the game (used for visualization purposes), `name` is the name or type of the entity, and `ID` is the unique identifier for each entity. The `x` and `y` values denote the entity's pixel coordinates on the map. Bounding boxes are reported in pixels as `top`, `bottom`, `left`, and `right` under the spatial convention above. `HP`, `shield`, and `energy` represent the entity's health, shield, and available energy (for entities capable of using abilities), respectively. See Appendix A for the data formats and examples.

2.1.3. Terrain data

Since terrain elevation does not change during the game, it is recorded only once at the start using the `getGroundHeight()` method in the `BWAPI::Game` module. Table 1 shows a brief explanation of the possible values for elevation of the terrain.

2.1.4. Vision data

Player vision is recorded as per-tile binary maps using `isVisible()`. We combine the two players' maps by logical OR (visible if visible to either player). We scan each binary visibility map in row-major order (left-to-right, top-to-bottom) and apply a run-length scheme that stores the initial bit ($b_0 \in \{0,1\}$; 0=unseen, 1=visible) followed by run lengths:

$$\langle \text{frame} \rangle,\ b_0,\ \ell_1,\ \ell_2,\ \ldots$$

Table 1
Possible values for terrain elevation. A *doodad* refers to a decorative terrain tile that is typically impassable or unbuildable, serving primarily as a visual element in map design without directly affecting elevation mechanics.

Value	Description
0	Low ground
1	Low ground doodad
2	High ground
3	High ground doodad
4	Very high ground
5	Very high ground doodad

Table 2
Comparison of data size and compression rates for raw vision data, conventional RLE, and custom RLE.

	Size (byte)		Compression rate (%)	
	avg.	std.	avg.	std.
Raw vision data	16,384			
Conventional RLE	7,263	2,109	44.33	12.87
Custom RLE (proposed)	3,639	1,054	22.21	0.06

For a 4×4 map with $b_0 = 0$, $\langle b_0, \ell \rangle = \langle 0; 2, 1, 3, 3, 2, 3, 1, 1 \rangle$, which corresponds to the same sequence of (value, count) runs. We emit a record only when the visibility map changes from the previous frame, which reduces I/O and storage overhead.

Table 2 summarizes the average storage size and compression ratio for raw vision, conventional RLE, and the proposed scheme. For a 128×128 tile grid, raw (byte-per-tile) storage is 16,384 bytes per frame. On a dataset of 67,372 frames from five replays, conventional RLE averages about 7,263 bytes (44.33% of raw), whereas our scheme averages 3,639 bytes (22.21% of raw). The lower standard deviation further indicates stable performance across frames.

2.1.5. Event data

In `BWAPI::AIModule`, key events capture crucial gameplay moments:

The `onUnitCreate()` event marks when a new entity appears, including when a building spawns a unit. It is used to log creation

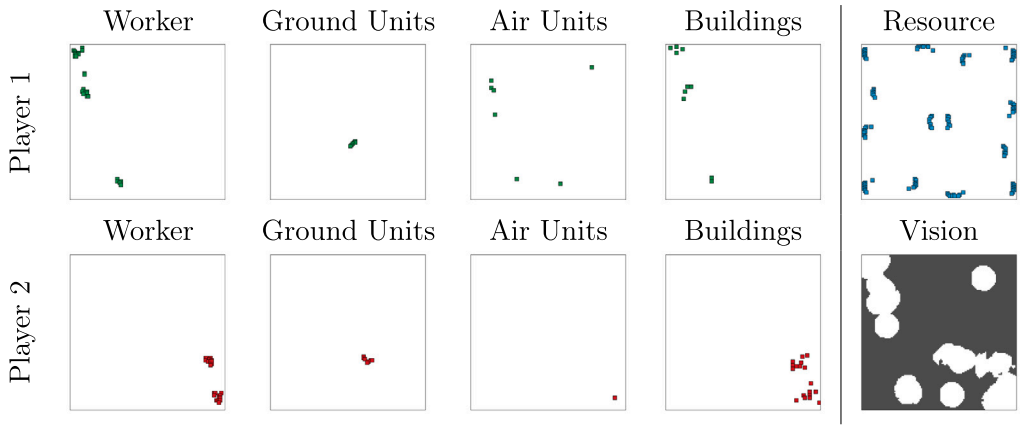

Fig. 3. Example of pre-processed in-game data. Each player's entity data (**left**). Resource data, such as `Mineral Fields` and `Vespene Geyser` (**top right**). Both players' vision data (**bottom right**). For better visibility, the entities are drawn at 10×their actual size.

timing or trigger actions for specific units. The `onUnitMorph()` event tracks entity transformations, such as a worker evolving into a structure, enabling monitoring and task execution upon completion. The `onUnitComplete()` event signifies when an entity becomes fully operational, allowing readiness tracking and combat preparation. The `onUnitDestroy()` event records when entities die in combat or buildings are dismantled, supporting event logging and follow-up actions. The `onUnitRenegade()` event, triggered by ownership changes, allows tracking and strategic adjustments. The `onNukeDetect()` event signals a detected nuclear launch before impact, aiding defensive responses or strategic analysis. Finally, the `onCombat()` event monitors combat phases, enabling real-time engagement tracking, recording unit participation, and adjusting tactics. See Appendix A for the data formats and examples.

2.2. Human observation data collector

In many RTS games, including StarCraft, players cannot simultaneously observe the entire battlefield due to its scale. To aid navigation, these games provide a "minimap", a scaled-down battlefield representation, along with keyboard and mouse controls for selecting the displayed area. The minimap offers an overview, while the currently viewed area is indicated by a "viewport".

`BWAPI::AIModule` allows AI players to access game information from the user's perspective. Using this, we implemented an AI agent that passively records viewport coordinates while a player watches a replay.

The agent logs the observed scene's coordinates at each frame, storing data in the following structured format:

⟨frame⟩ ⟨x⟩ ⟨y⟩

where `frame` represents the frame index, and `x` and `y` represent the viewport's top-left coordinates.

In StarCraft 1.16.1, the viewport is fixed at approximately 640×384 pixels, thus, logging only the top-left corner suffices to determine its position.

2.3. Preprocessor

The preprocessor aggregates the frame-wise data into structured channels. Since storing all entities and building types as individual channels results in excessive sparsity, data are classified into four groups: `Workers`, `Ground entities`, `Air entities`, and `Buildings`.

Table B.1 presents the entity and building classifications by race. This categorization consolidates spatial distributions into four channels.

In addition, the vision and terrain data are converted into separate channels, forming the final structured dataset. Fig. 2 illustrates how raw data are transformed into feature channels.

3. Illustrative examples

3.1. In-game data

Using the **Observer** framework, users can extract in-game data and collect human observation traces. This section provides representative visualizations of the preprocessed outputs.

Fig. 3 shows entity and vision states after preprocessing. For each player, entities are grouped into four channels—`workers`, `ground units`, `air units`, and `buildings`—to reduce sparsity while preserving functional roles. A separate channel encodes player vision, computed on tiles and combined across players by logical OR (visible if visible to either player), and another channel encodes resource locations (minerals, gas).

In the vision map (Fig. 3, bottom right), highlighted regions denote currently visible tiles. Because strategic decisions depend on available information, the vision channel provides a compact signal of observable state.

Fig. 4 depicts terrain elevation. Terrain is static within a replay; therefore, a single `width` × `height` tile map per game suffices.

Finally, the output is organized as feature channels. In the example, the dataset comprises 11 channels: four per player for entities, plus terrain, vision, and resource locations.[5] Channel resolution follows the tile grid; entity positions originate in pixel coordinates but are aggregated to tiles during channelization (Section 2.1). We report sparsity and storage characteristics of these channels in the results tables.

3.2. Human observation data

Human observation traces are recorded during replay watching and later used as training targets for the automatic observer. Fig. 5 illustrates a reconstructed state derived from the integrated channels in Figs. 3 and 4. The reconstruction does not replicate the original frame exactly but preserves essential entities and spatial relations, providing an analysis-ready abstraction. Colored rectangles indicate viewport positions recorded from observers, highlighting regions that attracted attention.

[5] The number of channels scales with the number of players.

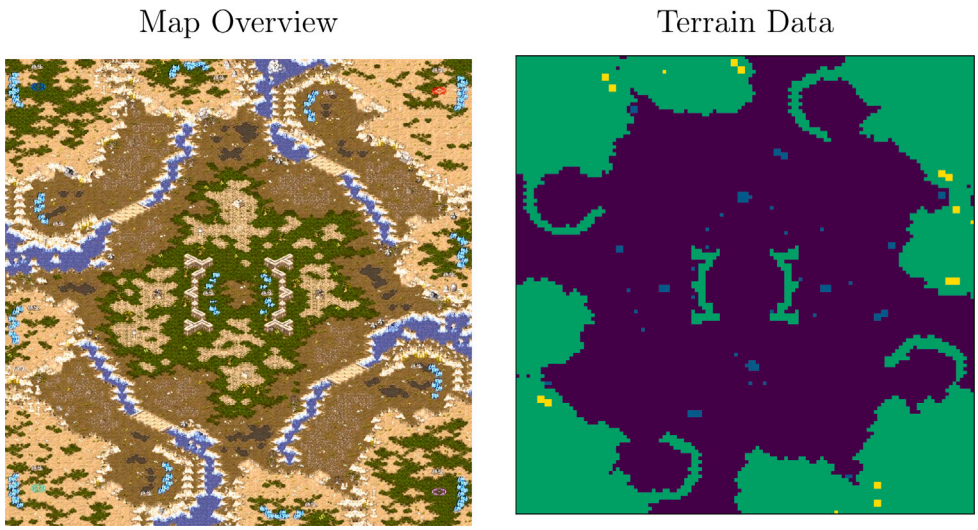

Fig. 4. Example of comparing the actual game map and the terrain data extracted from game map. (Map: `Fighting Spirit 1.3`).

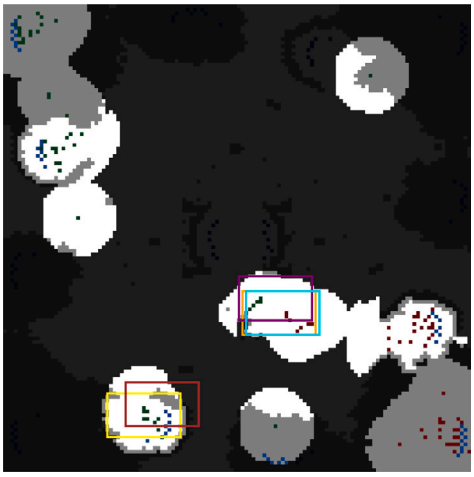

Fig. 5. The game state is reconstructed using feature channels. Each player is represented in a distinct color(red and green), while neutral units (resource fields) are depicted in blue. The terrain is visualized using a gray palette based on elevation, and areas hidden from both players due to the *fog of war* are rendered in black. Each observer's viewport is represented by a colored rectangle. In this figure, a total of five viewports are represented.

3.3. Learning method and reference implementation

Using Sections 3.1 and 3.2, we provide a *reference training recipe* (detection-style objective, data loaders, and evaluation scripts) to demonstrate end-to-end use of the pipeline. This paper *does not* report newly trained model accuracy; any accuracy numbers cited in Section 4.2 are *reproduced from prior work* for context only.

4. Impact

4.1. Framework: standardized pipeline and measurable system properties

Observer standardizes replay ingestion, feature-channel construction, and synchronization with human observation traces, enabling reproducible downstream studies. We instrument the pipeline to report *system-level* metrics under *offline replay processing*: processing throughput (FPS), per-frame preprocessing latency, parsing coverage/failure rates, and storage throughput (MiB per minute). These measurements proxy operational feasibility and remain independent of downstream model choices.

Table 3
System-level pipeline metrics under *offline replay processing* on a reference workstation. Metrics are computed over five randomly selected replays; values are means unless noted. *Stage scope:* "Extraction FPS" is measured during the *replay data extraction* stage, whereas "Preprocessing latency" is measured in the *preprocessor* stage; they do not represent the same stage. Processing FPS was measured on an Intel Core i7-6800K (64 GB RAM); other metrics on a dual-socket Intel Xeon E5-2630 v4 (64 GB RAM).

Metric	Value	Unit
Extraction FPS (offline)	5.39	frame/s
Preprocessing latency	7.71	ms/frame
Coverage	100	%
Storage I/O	10 703.85	MiB/min

Operational snapshot. Table 3 summarizes, on a reference machine, **extraction throughput** (FPS), **preprocessing latency**, parsing coverage on corrupted or partial replays, and net storage throughput (MiB per minute) after compression, alongside per-frame expansion across stages (Table 4).

Modularity and extensibility. Game-specific logic is isolated behind small interfaces (data-source adapter, entity mapping, coordinate transform, channelizer, label sink). Porting to another RTS (e.g., StarCraft II) primarily requires a new data-source adapter and entity taxonomy; downstream channelization and training code remain unchanged.

Emulated real-time assessment. We approximate real-time constraints by pacing offline processing to the nominal frame rate and reporting FPS and latency distributions (Table 3). Live integration remains future work.

4.2. Evaluation: System metrics (new) and reference results (external)

This section presents (i) *new* system-level evaluation of the software (preprocessing throughput, determinism, resource usage) and (ii) *external* reference results (prior model accuracy, human–human agreement) for context only. Unless stated otherwise, accuracy numbers are *reused or cited from prior work* and were *not* newly measured here. We mark reproduced values in captions and notes.

4.2.1. System metrics

We evaluate the pipeline on a standard workstation and report (a) **extraction throughput** (FPS, measured during replay *data extraction*), (b) **per-frame preprocessing latency** (measured in the *preprocessor*

Table 4
Per-frame compression ratio and expansion across processing stages (O = Original `.rep`, E = Extracted, P = Preprocessed).

	O→E	E→P	O→P
Compression ratio (from/to)	0.000697	0.003760	0.00000262
Compression (%)	0.0697%	0.3760%	0.000262%
Expansion (to/from)	×1,434.26	×265.93	×381,417.33

stage), (c) coverage, and (d) storage I/O and footprint. See Table 3 for a consolidated offline summary.

Storage footprint and per-frame expansion. We quantify the per-frame transformation from archival logs to model-ready tensors. Table 4 reports compression ratios and expansion factors across stages.

Rationale. The replay (`.rep`) format encodes sparse, event-driven logs with domain-specific, variable-length compression optimized for storage and transmission. Extraction and preprocessing materialize dense, multi-channel tensors of full game state at fixed frame intervals, removing temporal and spatial sparsity. For learning and analysis, integer/bitfield data become floating-point tensors (e.g., `float32`) and are augmented with one-hot channels, temporal stacks, and reproducibility metadata—intentionally increasing volume. General-purpose formats (e.g., `.npy`) prioritize portability and I/O over maximal entropy coding, yielding lower compression than specialized replay logs. Thus the very low compression ratios (and large expansions) are an expected, goal-aligned consequence of producing model-ready dense representations.

4.2.2. Reference results (external, not measured here)

Provenance note. The figures in this subsection are *reproduced from prior work* and were *not* measured here; they are included solely to contextualize achievable bounds for imitation-style observers. For example, the learned observer achieves 56.9% on the Viewport-Overlap Benchmark [16], and human–human viewport overlap is around 50%–55% (e.g., 52.3% with SD=4.36) [16].

4.3. Practical and academic impact

Practical. By explicitly exposing latency, throughput, and footprint—and decoupling data engineering from policy learning—**Observer** enables planning for live or near–real-time use in small events where expert observers are impractical. Standardized outputs also ease integration with broadcast tools (e.g., rule-based fallbacks or hybrid controllers). *Subsampling for live use.* In StarCraft: Brood War (24 FPS), per-frame execution is often unnecessary: the system can trigger extraction–preprocessing–inference at N Hz, i.e., every $k = \lceil 24/N \rceil$ frames (e.g., N=4 Hz ⇒ k=6; N=6 Hz ⇒ k=4), reducing compute and I/O while preserving a broadcast-friendly cadence; simple hold-last or interpolation smooths camera motion.

Academic. A shared representation and reference pipeline reduce variance across projects and make comparisons more credible. The framework complements existing RTS tools [8,11,18,19] by providing a measured, reproducible path from replays to learning-ready tensors.

4.4. Limitations and future work

API dependence. The extractor relies on game APIs; titles without APIs require screen-based extraction [20] or community tooling.

Observer-trace quality. Performance varies with who generates observation traces (expertise, intent). Increasing trace diversity and modeling annotator variance are natural next steps.

Live integration. The current release operates on replays only; online integration for live 1v1 matches has yet to be implemented. All system-level metrics were collected offline and should be interpreted as feasibility indicators rather than deployment results.

5. Conclusions

We presented **Observer**, a software framework that standardizes replay ingestion, feature-channel construction, and synchronization with human observation traces for StarCraft: Brood War. The framework produces a structured, learning-ready representation and records observer viewports that can be used as training targets for automatic camera control. It also exposes system-level measurements (processing FPS, per-frame preprocessing latency, coverage, and compression) under offline replay processing, enabling comparative assessment of operational feasibility independent of model choice.

Using the standardized channels as inputs and aggregated human viewports as targets, prior work reports consistent improvements over rule-based baselines under a modified IoU metric (see Section 4.2) Our software contribution enables such studies by providing a measured, reproducible data path.

Practically, a measured and reproducible data path may reduce the engineering burden of building observer models and can inform planning for small events. Academically, **Observer** provides a common basis for replication and fair comparison across studies. The current release operates on replays; live integration for 1v1 matches is left for future work.

CRediT authorship contribution statement

Cheong-mok Bae: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Data curation, Conceptualization. **Ho-Taek Joo:** Writing – original draft, Validation, Methodology. **SungHa Lee:** Writing – original draft, Validation, Methodology. **Kyung-Joong Kim:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Cheong-mok Bae reports financial support was provided by Korea Creative Content Agency. Ho-Taek Joo reports financial support was provided by Korea Creative Content Agency. SungHa Lee reports financial support was provided by Korea Creative Content Agency. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was supported by Culture, Sports and Tourism R&D Program through the Korea Creative Content Agency grant funded by the Ministry of Culture, Sports and Tourism in 2024 (Project Name: Development of generative AI-based esports service automation platform technology to improve esports operation efficiency, Project Number: RS-2024-00441523).

Appendix A. Data format and example

Metadata

```
map_name,width,height,game_length,players_data
fighting_spirit_1.3,128,128,27996,[anonymous_1
    ;Zerg;(7 6)][anonymous_2;Terran;(117 117)]
```

Table B.1
Lists all the entity types for each race in StarCraft, categorized into workers, ground entities, air entities, and buildings.

Race	Category	Type
Terran	Workers	SCV
	Ground entities	Firebat, Ghost, Goliath, Marine, Medic, Siege_Tank_Siege_Mode, Siege_Tank_Tank_Mode, Vulture, Vulture_Spider_Mine
	Air entities	Battlecruiser, Dropship, Nuclear_Missile, Science_Vessel, Valkyrie, Wraith
	Buildings	Academy, Armory, Barracks, Bunker, Command_Center, Engineering_Bay, Factory, Missile_Turret, Refinery, Science_Facility, Starport, Supply_Depot, Comsat_Station, Control_Tower, Covert_Ops, Machine_Shop, Nuclear_Silo, Physics_Lab
Protoss	Workers	Probe
	Ground entities	Archon, Dark_Archon, Dark_Templar, Dragoon, High_Templar, Reaver, Scarab, Zealot
	Air entities	Arbiter, Carrier, Corsair, Interceptor, Observer, Scout, Shuttle
	Buildings	Arbiter_Tribunal, Assimilator, Citadel_of_Adun, Cybernetics_Core, Fleet_Beacon, Forge, Gateway, Nexus, Observatory, Photon_Cannon, Pylon, Robotics_Facility, Robotics_Support_Bay, Shield_Battery, Stargate, Templar_Archives
Zerg	Workers	Drone
	Ground entities	Broodling, Defiler, Drone, Egg, Hydralisk, Infested_Terran, Larva, Lurker, Lurker_Egg, Ultralisk, Zergling
	Air entities	Cocoon, Devourer, Guardian, Mutalisk, Overlord, Queen, Scourge
	Buildings	Creep_Colony, Defiler_Mound, Evolution_Chamber, Extractor, Greater_Spire, Hatchery, Hive, Hydralisk_Den, Infested_Command_Center, Lair, Nydus_Canal, Queens_Nest, Spawning_Pool, Spire, Spore_Colony, Sunken_Colony, Ultralisk_Cavern

Entity data

```
frame,player,race,player_color,name,ID,x,y,top
    ,bottom,left,right,HP,max_HP,shield,
    max_shield,energy,max_energy
0,anonymous_1,Zerg,Red,Zerg_Drone,1,312,288,
    277,299,301,323,40,40,0,0,0,0
0,anonymous_2,Terran,Yellow,Terran_SCV,17,3832
    ,3848,3837,3859,3821,3843,60,60,0,0,0,0
0,Neutral,Zerg, ,Resource_Mineral_Field,77,
    3776,2960,2944,2975,3744,3807,100000,
    100000,0,0,0,0
...
```

Event data

```
frame,x,y,player,entity,event
0,96,336,Neutral,Resource_Mineral_Field,Create
0,96,336,Neutral,Resource_Mineral_Field,
    Complete
0,312,288,anonymous_1,Zerg_Drone,Create
0,312,288,anonymous_1,Zerg_Drone,Complete
...
```

Appendix B. StarCraft entity categories

See Table B.1.

References

[1] Block S, Haack F. Esports: a new industry. In: SHS web of conferences, vol. 92, EDP Sciences; 2021, p. 04002.

[2] Newman JI, Xue H, Watanabe NM, Yan G, McLeod CM. Gaming gone viral: An analysis of the emerging esports narrative economy. Commun Sport 2022;10(2):241–70.

[3] Oh I-S, Cho H, Kim K-J. Playing real-time strategy games by imitating human players' micromanagement skills based on spatial analysis. Expert Syst Appl 2017;71:192–205. http://dx.doi.org/10.1016/j.eswa.2016.11.026, URL https://www.sciencedirect.com/science/article/pii/S0957417416306613.

[4] Justesen N, Risi S. Learning macromanagement in starcraft from replays using deep learning. In: 2017 IEEE conference on computational intelligence and games. 2017, http://dx.doi.org/10.1109/CIG.2017.8080430.

[5] Vinyals O, Ewalds T, Bartunov S, Georgiev P, Vezhnevets AS, Yeo M, Makhzani A, Küttler H, Agapiou J, Schrittwieser J, et al. Starcraft ii: A new challenge for reinforcement learning. 2017, arXiv preprint arXiv:1708.04782.

[6] Glass BD, Maddox WT, Love BC. Real-time strategy game training: emergence of a cognitive flexibility trait. PLoS One 2013;8(8):e70350.

[7] Ontanón S, Synnaeve G, Uriarte A, Richoux F, Churchill D, Preuss M. A survey of real-time strategy game AI research and competition in StarCraft. IEEE Trans Comput Intell AI Games 2013;5(4):293–311.

[8] Tang Z, Shao K, Zhu Y, Li D, Zhao D, Huang T. A review of computational intelligence for StarCraft AI. In: 2018 IEEE symposium series on computational intelligence. IEEE; 2018, p. 1167–73.

[9] Dale G, Shawn Green C. The changing face of video games and video gamers: Future directions in the scientific study of video game play and cognitive performance. J Cogn Enhanc 2017;1:280–94.

[10] Nagorsky E, Wiemeyer J. The structure of performance and training in esports. PLoS One 2020;15(8):e0237584.

[11] Synnaeve G, Nardelli N, Auvolat A, Chintala S, Lacroix T, Lin Z, Richoux F, Usunier N. Torchcraft: a library for machine learning research on real-time strategy games. 2016, arXiv preprint arXiv:1611.00625.

[12] Lie H, Lukas D, Liebig J, Nayak R. A novel learning-to-rank method for automated camera movement control in e-sports spectating. Commun Comput Inf Sci 2019;996:149–60. http://dx.doi.org/10.1007/978-981-13-6661-1_12/COVER/, URL https://link.springer.com/chapter/10.1007/978-981-13-6661-1_12.

[13] Tot M, Conserva M, Chitayat AP, Kokkinakis A, Patra S, Demediuk S, Munoz AC, Olarewaju O, Ursu M, Kirmann B, Hook J, Block F, Drachen A, Perez-Liebana D. What are you looking at? Team fight prediction through player camera. In: IEEE conference on computatonal intelligence and games, vol. 2021-August, 2021, http://dx.doi.org/10.1109/CoG52621.2021.9619038.

[14] Phang DW-S. Intelligent camera control in game replays [Ms thesis], Lehigh University; 2014.

[15] Mattsson BP, Vajda T, Čertickỳ M. Automatic observer script for starcraft: Brood war bot games (technical report). 2015, arXiv preprint arXiv:1505.00278.

[16] Joo H-T, Lee S-H, Bae C-m, Kim K-J. Learning to automatically spectate games for esports using object detection mechanism. Expert Syst Appl 2023;213:118979.

[17] Šustr M, Malý J, Čertický M. Multi-platform version of starcraft: brood war in a docker container: technical report. 2018, arXiv:arXiv:1801.02193.

[18] Lin Z, Gehring J, Khalidov V, Synnaeve G. Stardata: A starcraft ai research dataset. In: Proceedings of the AAAI conference on artificial intelligence and interactive digital entertainment, vol. 13, 2017, p. 50–6.

[19] Kim M-J, Kim K-J, Kim S, Dey AK. Evaluation of starcraft artificial intelligence competition bots by experienced human players. In: Proceedings of the 2016 CHI conference extended abstracts on human factors in computing systems. 2016, p. 1915–21.

[20] Kim D-W, Park S-Y, Yang S-I, Lee S-K. Real-time player tracking framework on MOBA game video through object detection. IEEE Trans Games 2024.