Capsule: Efficient Player Isolation for Datacenters

Z. Du, N. Davari, L. Li†, W. S. Loi, N. Kodirov

Huawei Technologies, Vancouver, BC, Canada

Figure 1: *An example application for a museum digital twin: (a) no Capsule, one player per engine, (b) with Capsule, nine players per engine, (c) compares datacenter resource utilization with and without Capsule. Capsule provides lightweight and efficient player-isolation.*

Abstract

Cloud gaming is increasingly popular. A challenge for cloud providers is to keep datacenter utilization high: a non-trivial task due to the diversity of hardware and applications. We introduce Capsule, a mechanism to seamlessly share datacenter resources across multiple players. We implemented Capsule in the Open 3D Engine (O3DE). Our evaluations show that Capsule increases datacenter resource utilization by accommodating up to 2.25x more players, without degrading user experience. This is the product of Capsule using up to 1.43x less GPU, 3.11x less VRAM, 3.7x less CPU, and 3.87x less RAM compared to the baseline. Capsule is also application-agnostic, i.e., no changes were required to run applications on the Capsule-based O3DE. Our experiences with four applications, three servers with different hardware specifications, including one with four GPUs, and a multi-server cluster shows that the Capsule design can be adopted by other game engines to increase datacenter utilization.

CCS Concepts

• ***Computer systems organization*** → *Cloud computing; Real-time system architecture;* • ***Computing methodologies*** → ***Graphics systems and interfaces;***

1. Introduction

Cloud gaming is attractive for both players and cloud providers. For players, it alleviates deployment cost. They no longer need to own the latest hardware (e.g., GPU) to play games in high quality. The cloud already hosts the latest hardware, sometimes before it becomes publicly available [PGD23]. For providers, the goal is to generate revenue while delivering the highest gaming quality. More players lead to higher datacenter utilization and, consequently, higher revenue.

However, it is challenging to achieve high datacenter utilization with gaming applications. Part of this challenge arises from games having diverse *shapes* and *sizes*. Shapes correspond to the diverse resources that games consume, such as CUDA cores, RT cores, and Tensor cores in GPUs, in addition to the host CPU and RAM. Sizes correspond to the differing amounts of these resources that games consume, e.g., a graphics-intensive game consumes the entire GPU, while a graphics-light game consumes only a fraction of that GPU. Another challenge arises from the diverse hardware found in datacenters. Datacenters often house servers with several generations of CPUs and GPUs [PGR*23]. For example, a server with an older GPU might accommodate only one player, while a server with the latest GPU accommodates dozens. Thus, cloud providers need a mechanism to share GPUs, as well as other resources, across multiple players.

† Work done while at Huawei

DOI: 10.2312/egs.20261027

Table 1: *Existing work: academic papers and real deployments.*

	Console	O3DE	Capsule
Remote	x	✓	✓
Compute & Mem. Sharing	✓	x	✓

We propose player-level multiplexing. A player in a graphics-heavy application will continue consuming the entire GPU. However, when a GPU has sufficient capacity to accommodate two or more players in a multiplayer game, its resources will be multiplexed across these players. We designed, implemented, and evaluated Capsule: an in-game-engine player isolation mechanism for multiplayer games. Capsule also allows cross-player *sharing*. For example, when two players enter a room and have a shared game asset in their view, we can reuse the asset geometry across these two players without players noticing. Sharing allows amortization, i.e., sublinear growth in datacenter utilization for a linear increase in player count: a phenomena we call a *sublinear resource footprint*.

We implemented Capsule in O3DE [O3D26b]. It satisfies four practical requirements for player-isolation in the cloud:

- **R1: Transparent**: Players should be unaware of other players sharing cloud resources. A player's experience—such as input latency, output streaming quality, and frames-per-second (FPS)—should not degrade due to other players.
- **R2: Compatible**: Player isolation should not require significant changes to run existing applications; ideally, no application changes are required. The workflow for developing a new application should also remain nearly identical, if not exactly identical.
- **R3: Lightweight**: The isolation mechanism itself should not consume significant system resources, such as CPU and RAM.
- **R4: Efficient**: Maximize cross-player sharing. For example, the resource footprint (e.g., CPU) of the second player should be less than that of the first player, because the second player can reuse parts of the computation results from the first player.

In summary, we make the following contributions: **(1)** We outline practical requirements (R1-R4) for player isolation, which we believe are common across cloud providers; **(2)** we propose a system, Capsule, that implements these requirements. Capsule is a novel in-engine player isolation mechanism. It decouples players' local and global states to maximize cross-player sharing. **(3)** We offer a thorough evaluation of Capsule using applications with diverse characteristics, servers with different hardware specifications, and a multi-server cluster.

2. Related Work

Table 1 broadly categorizes existing work into three groups across two dimensions. The *Console* group represents deployments with user hardware, such as the PS5, Xbox, and PC. Here, frames are generated on the user hardware, i.e., no heavy computation occurs in the cloud. The user console's compute and memory resources are shared across players (e.g., [Haz21]). On the other hand, in the second group, labelled *O3DE*, frames are generated in the cloud, but compute and memory resources in the cloud cannot be shared

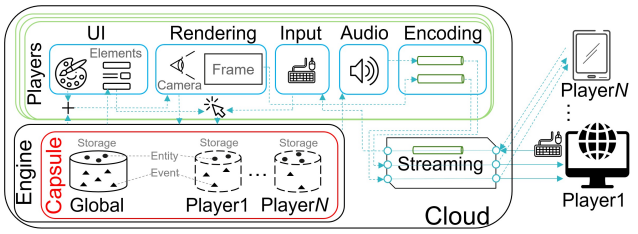

Figure 2: *Capsule-based cloud architecture. Capsule Storage manages player state, which is a key part for player-isolation.*

across players. This is the case for any game engine running in the cloud (e.g., Unreal Engine and Unity), although we label the group *O3DE*. Capsule (i.e., O3DE with Capsule) enables cross-player resource sharing in the cloud.

A large body of literature, such as CloudLight [CLM*15] and Weinrauch et al. [WTS*23], also belongs to the Capsule group. They generate frames in the cloud, and they share the compute and memory resource used to generate those frames for multiple players. However, these works only demonstrate the feasibility of performing (part of the) rendering computation in the cloud, without addressing the practical requirements (R1-R4).

3. Design and Implementation

Figure 2 shows the Capsule architecture, along with other essential modules in a cloud deployment. Capsule is a new module in O3DE that communicates with different system components, such as the audio system, input system, rendering system (which includes both audio and video rendering), and game logic (event system). Capsule leverages the Entity–Component–System (ECS) architecture, which is widely adopted by modern game engines (e.g., [Uni26]), including O3DE. ECS makes it convenient to represent game world objects. An ECS-based game engine contains *entities* that have data *components* and *systems* to operate on those components.

As shown in Figure 2, players connect to the cloud over a wide area network (e.g., the Internet). The players' entry point is the Streaming module, which creates a separate game session for each player, isolating their inputs (e.g., keyboard and mouse), and outputs (e.g., video streams). The Streaming module passes the player-specific input to a separate game session, which includes player-specific UI, Rendering, Input, Audio, and Encoding states. These states are reflected in the game but are managed inside the engine. In other words, the application will perceive all player-specific states as being isolated from each other at the engine level; i.e., as if each player (or, in fact, each game instance) has an engine of its own. In Capsule-based O3DE, all of these players share one engine.

Capsule distinguishes different players' inputs, outputs, and behaviours by using Capsule Storage. Capsule Storage manages two constructs: entities and events. Entities are game objects, such as a car, a light, or an avatar. Events, or gameplay events, include in-game events such as running, jumping, or exploding. They are pre-designed by the game developer and are written in the game script.

Table 2: *Workstations used for Capsule evaluation. All workstations use NVIDIA GPUs. (*Exact GPU model is not disclosed. VRAM "24+" means it has more VRAM than the above two.* FORTIS *is our custom label, meaning* ***strong*** *in Latin.)*

	SINGLEGPU	DUALGPU	QUADGPU
CPU	AMD Ryzen 7 5800X	Intel Core i7-13700K	Threadripper PRO 5975WX
# of Cores	8	16	32
RAM	32 GB	64 GB	256 GB
GPU	GeForce RTX 4090	GeForce RTX 3090	FORTIS *
VRAM	24 GB	24 GB	24+
# of GPUs	1	2	4

Entities and events determine the behavior of each player and of the global environment. They also determine the final rendered frame.

Capsule has two types of Storage: global and local. There is only one global storage in the entire engine. All players share the entities and events inside the global storage. There are one or more local Capsule Storages: one for each player. Entities inside the local storage are visible only to the storage-owner player. Capsule isolates player-specific tasks at runtime by directing player-specific entities and events to the respective player-owned local storage. (Appendix A in the Supplementary Materials describes player entity tracking using global and local storage.)

We ported four applications to the Capsule-based O3DE to validate our design. All four applications fit within our entity and event tracking systems. Figure 1 shows only one of them for brevity. (The Appendix in Supplementary Materials shows the other three applications.) Our experience with these four applications shows that Capsule-based O3DE is fully compatible (**R2**) with standard non-Capsule O3DE.

4. Evaluation

We evaluated Capsule on diverse datacenter hardware. We used three different workstations: one with a single GPU, one with two GPUs, and one with four GPUs, as described in Table 2. All workstations run the Windows OS to faithfully reproduce our production environment. We fixed the application FPS to 30, a common minimum threshold. We read the system-wide utilization levels of the GPU, VRAM, CPU, and RAM resources every second. The value for each second is the average utilization during that one-second interval, which is consistent with Windows performance counters [Mic25].

We compare the cloud server resource consumption of Capsule against the baseline. For the baseline, we implemented process-level isolation, (i.e., a separate game engine process for each player). In the baseline, we launched the game server [LPM06] and then launched game clients one by one, measuring the server utilization as players were added. The client-side evaluation is identical between Capsule and the baseline, but on the server side, after the first player, we kept adding players to the same client process (running in the cloud server), rather than creating a separate process

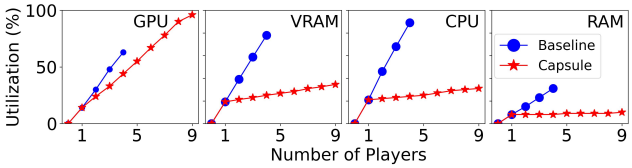

Figure 3: *Scalability of Capsule as we increase the number of players. Capsule and the baseline host up to 9 and 4 players, respectively, on the* EXHIBITION *application. Capsule accommodates up to 2.25x more players thanks to the sublinear resource increase per added player.*

per player (in the cloud server). This is consistent with the Capsule design in Figure 2. (We were unable to use production-level alternative virtualization techniques, such as the NVIDIA RTX Virtual Workstation (vWS), as the baseline due to licensing restrictions [NVI25].)

We evaluated Capsule's applicability to diverse datacenter workloads by running three different applications–PARIS OPERA HOUSE, O3DE MULTIPLAYER SAMPLE [O3D26a], and EXHIBITION–on the SINGLEGPU workstation. For brevity, Figure 1 shows results for only the EXHIBITION application on the SINGLEGPU workstation, which has the strongest GPU. We monitored resource utilization for 40 seconds of gameplay time. (Appendix B in the Supplementary Materials describes our evaluation methodology. Gameplay footage is also included in the Supplementary Materials.)

Figure 1(a) and Figure 1(b) show the game server view of the EXHIBITION application with a single player (1p) and nine players (9p), respectively. Figure 1(c) shows resource utilization for these two environments. GPU utilization with a single player is around 20%. VRAM and CPU utilization are also around 20% while RAM utilization is around 24%. Capsule takes advantage of the remaining capacity to allocate eight more players. The GPU becomes a bottleneck with the 10th player, causing the player FPS to drop below the threshold (30). At this point, we stopped allocating more players. As Figure 1(c) shows, GPU utilization increases from 18% with one player to almost 99% with nine players.

Figure 3 shows the average resource utilization increase as players join the EXHIBITION application in Figure 1(c). Figure 3 shows that overall GPU utilization increases by around 10 percentage points per additional player, after the first one. Other figures show a similar trend for VRAM, CPU, and RAM. Unlike in Figure 1, in Figure 3 we kept adding players in the baseline, (i.e., adding another game-engine process for each player). The baseline sharply dropped to single-digit FPS after 4 players due to CPU contention, as shown in Figure 3. At the peak of the baseline (4 players), Capsule used **1.43x less GPU**, **3.11x less VRAM**, **3.7x less CPU**, and **3.87x less RAM** compared to the baseline. Thanks to these savings, Capsule accommodates more players, beyond the baseline. As expected, the baseline's resource consumption increases linearly with the number of players. However, the increase in Capsule is sublinear thanks to cross-player sharing, i.e., Capsule requires a sublinear amount of extra VRAM, CPU, and RAM past the first player. The GPU utilization benefit of Capsule could be further improved if

the application uses more shareable rendering techniques, such as shadowmaps, global illumination, and cross-view diffuse and effect sharing [WTS*23].

Figure 1(c) contrasts the utilization levels for the baseline (1p) and Capsule (9p), which we call *delta*. The delta is the highest for the GPU resource. This delta is smaller but is more **significant** in other resources (VRAM, CPU, and RAM). If the delta were the smallest (i.e., is zero), the 1p and 9p lines would overlap. This would mean that the 9p utilization of that resource is identical to that of 1p; i.e., the additional players came for free (for that resource). In Figure 1(c), this is almost the case for VRAM, which is also reflected in Figure 3. Figure 3 shows that RAM consumption stays relatively flat as more players are added. It also shows that the angle of the sublinearity line differs between resources: it is highest for the GPU and lowest for RAM. This is because Capsule is able to achieve a high degree of cross-player RAM sharing in the EXHIBITION application.

We evaluated Capsule with two other applications and diverse hardware. We summarize the results for brevity. One of the two other applications is from our production, called PARIS OPERA HOUSE: a digital twin of the Paris Opera House. The second application is an open-source multiplayer shooting game, called O3DE MULTIPLAYER SAMPLE [O3D26a]. Both of these applications are more graphics-intensive than EXHIBITION. In PARIS OPERA HOUSE, Capsule accommodates only two players (before hitting the FPS threshold) and only four players in O3DE MULTIPLAYER SAMPLE. Overall, Capsule's sublinearity benefits hold in these applications as well, e.g., an ≈10 percentage points lower GPU utilization per-player with Capsule in the most graphics-intensive PARIS OPERA HOUSE application.

In our hardware diversity experiments, we evaluated EXHIBITION on the three workstations in Table 2 as well as on a two-server cluster that connects DUALGPU and QUADGPU workstations over the network. Figure 1 and Figure 3 already show results for the SINGLEGPU. The sublinearity trends also holds for the DUALGPU (accommodates up to 8 players) and QUADGPU (accommodates up to 16 players) workstations, where the GPU resource has the highest delta, followed by the CPU and then VRAM, with the RAM resource having the lowest delta. These results also hold for two-server cluster, which accommodates up to 24 players in aggregate. (Appendix C in the Supplementary Materials elaborates on application variety and hardware diversity experiments.)

5. Limitations and Discussion

Capsule has three major limitations: (1) a CPU bottleneck, (2) performance isolation, and (3) fate-sharing. The current implementation of Capsule focuses on multiplexing GPU resources because GPUs are expensive and are the main source of bottlenecks in our production. However, for some applications, the bottleneck shifts to the CPU or other resources. In PARIS OPERA HOUSE, for example, Capsule accommodates only two players because the third player drops the FPS below the threshold (30) due to a CPU bottleneck. In the current implementation, Capsule has one main thread for all players, which runs on a single CPU core. That thread becomes the bottleneck. We can extend Capsule to use different CPU cores for different players. This will widen Capsule's applicability to diverse applications.

Capsule currently offers functional isolation; e.g., one player jumping does not interfere with another player jumping (even when both players share the same GPU). However, Capsule does not offer performance isolation, which is required when players on the same GPU contend for the same resource. Capsule also introduces player fate sharing by colocating multiple players on the same GPU. Thus, if one GPU fails, multiple players suffer. Our future work can leverage existing isolation and fault-tolerance techniques to alleviate these limitations. (Appendix D in the Supplementary Materials further discusses these techniques.)

Capsule's sharing benefits can be further improved. Capsule's cross-player sharing is inspired by cross-VM (Virtual Machine) compute and memory sharing, present since the early days of the cloud (e.g., by Waldspurger [Wal03] to share memory at the OS page level). Difference Engine [GLV*10] further increases the sharing degree via sub-page-level sharing. EndRE [AAA*10] applies network-level redundancy-elimination techniques to reduce bandwidth consumption between cloud endpoints. Capsule is complementary to these lines of work. In fact, similar optimizations already exist in our cloud software stack. At the same time, Capsule can be further improved by applying these techniques at the game-engine level (e.g., sub-asset-level sharing, as in to Difference Engine, and redundancy-elimination in cross-player network streams, as in EndRE). Put differently, Capsule intends to bring already-successful optimizations in other parts of cloud systems (e.g., OSes, hypervisors, and storage systems) to the game-engine level.

6. Conclusion

We designed, implemented, and evaluated Capsule: an efficient in-game-engine player isolation mechanism. It satisfies all four player-isolation requirements in the cloud. Our implementation in a popular open-source game engine, O3DE, shows that Capsule is application-agnostic. We ported four existing applications to the Capsule-based O3DE without application changes. Our experiments with these applications, three servers with different hardware specifications, including multi-GPU servers and a multi-server cluster, show that Capsule can increase datacenter resource utilizations—GPU, VRAM, CPU, and RAM—by accommodating up to 2.25x more players. This is the product of Capsule using up to 1.43x less GPU, 3.11x less VRAM, 3.7x less CPU, and 3.87x less RAM compared to the baseline. The Capsule design can be adopted by other game engines to increase datacenter utilization across cloud providers.

References

[AAA*10] AGGARWAL B., AKELLA A., ANAND A., BALACHANDRAN A., CHITNIS P., MUTHUKRISHNAN C., RAMJEE R., VARGHESE G.: EndRE: an end-system redundancy elimination service for enterprises. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (2010), NSDI'10. 4

[CLM*15] CRASSIN C., LUEBKE D., MARA M., MCGUIRE M., OSTER B., SHIRLEY P., SLOAN P.-P., WYMAN C.: CloudLight: A system for amortizing indirect lighting in real-time rendering. *Journal of Computer Graphics Techniques (JCGT)* (2015). 2

[GLV*10] GUPTA D., LEE S., VRABLE M., SAVAGE S., SNOEREN A. C., VARGHESE G., VOELKER G. M., VAHDAT A.: Difference engine: harnessing memory redundancy in virtual machines. *Commun. ACM* (2010). 4

[Haz21] HAZELIGHT STUDIOS: It Takes Two. Video Game, 2021. 2

[LPM06] LU F., PARKIN S., MORGAN G.: Load balancing for massively multiplayer online games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games* (2006). 3

[Mic25] MICROSOFT: About performance counters, 2025. URL: https://learn.microsoft.com/en-us/windows/win32/perfctrs/about-performance-counters. 3

[NVI25] NVIDIA: Nvidia vgpu software (rtx vws, vpc, vapps), 2025. URL: https://www.nvidia.com/en-us/drivers/vgpu-software-driver. 3

[O3D26a] O3DE: MultiplayerSample Project, 2026. URL: https://github.com/o3de/o3de-multiplayersample. 3, 4

[O3D26b] O3DE: Open 3D Engine, 2026. URL: https://github.com/o3de/o3de. 2

[PGD23] PETTY H., GOLDWASSER I., DESALE P.: One Giant Superchip for LLMs, Recommenders, and GNNs, 2023. 1

[PGR*23] PATEL P., GONG Z., RIZVI S., CHOUKSE E., MISRA P., ANDERSON T., SRIRAMAN A.: Towards improved power management in cloud gpus. *IEEE Computer Architecture Letters* (2023). 1

[Uni26] UNITY: ECS for Unity, 2026. URL: https://unity.com/ecs. 2

[Wal03] WALDSPURGER C. A.: Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.* (2003). 4

[WTS*23] WEINRAUCH A., TATZGERN W., STADLBAUER P., CRICKX A., HLADKY J., COOMANS A., WINTER M., MUELLER J. H., STEINBERGER M.: Effect-based multi-viewer caching for cloud-native rendering. *ACM Trans. Graph. 42*, 4 (July 2023). 2, 4