SPEAR: A Simulator for Photorealistic Embodied AI Research Supplementary Material

Mike Roberts[1,2], Renhan Wang[3], Rushikesh Zawar[4], Rachith Dey-Prakash[2], Quentin Leboutet[2], Stephan R. Richter[2], Matthias Müller[2], German Ros[5], Rui Tang[3], Stefan Leutenegger[6,7], Yannick Hold-Geoffroy[1], Kalyan Sunkavalli[1], and Vladlen Koltun[2]

[1]Adobe Research [2]Intel Labs [3]Manycore Tech Inc [4]Adobe [5]NVIDIA
[6]ETH Zurich [7]Imperial College London

https://github.com/spear-sim/spear

1 Open-Source Commitment

Our entire codebase is publicly available under an MIT license, and includes: our UE plugins; our Python client; a sample UE project; our collection of 36 example applications demonstrating the capabilities of SPEAR; a collection of scripts we use internally to assist with UE development; and the apartment scene shown throughout the paper. The apartment scene is released under a Creative Commons license. All example applications shown in this paper and supplementary material are available on our public code repository.

Fig. 1: In this application, we control UE's default offroad vehicle using SPEAR.

Fig. 2: In this application, we import meshes from the Stanford 3D Scanning Repository [13] into UE, and we spawn them dynamically at runtime using SPEAR.

2 Example Applications

We demonstrate several additional example applications in Figures 1, 2, 3, 4, and 5. We include videos in our supplementary ZIP file. Unless otherwise noted, we recorded all of our videos on a Windows 11 laptop with an NVIDIA GeForce RTX 4090 laptop GPU, a 2.3 GHz Intel Ultra 9 processor with 16 physical cores, and 32 GB of memory. We recorded our agentic scene editing

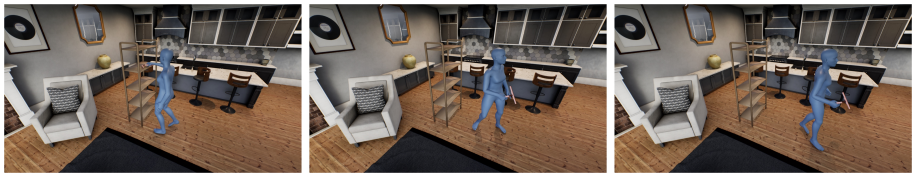

Fig. 3: In this application, we import a human character animation from the Humoto dataset [7] into UE, and we spawn it dynamically at runtime using SPEAR.

and CityGenerator videos on a macOS Sequoia laptop with an Apple M3 Max integrated GPU and CPU with 40 physical cores and 128 GB of memory.

In Tables 1 and 2, we summarize the unique functionality in each of our example applications that would not be possible to implement in any existing UE-based simulator. We consider our implementations of `find_actor_by_class` and `find_actors_by_class` to be unique because they respect the UE type hierarchy; `inject_input_for_actor` because it supports injecting continuous vector-valued input actions, and it supports routing input actions to specific actors; `load_class` and `spawn_actor` because they support loading and spawning Blueprint types; and `read_pixels` because the SPEAR camera sensor supports image modalities that aren't available in any existing UE-based simulator (e.g., a non-diffuse intrinsic image decomposition, physically-based shading parameter images, and material ID images).

3 The SPEAR Camera Sensor

Implementing our Camera Sensor as an Unreal Component Perhaps surprisingly, our default camera sensor is not a fundamental primitive in our system. Instead, it is implemented as a lightweight UE component [2] that derives from UE's default render-to-texture component, and defines an SpFunction named `read_pixels` for returning rendered image data from the GPU. We demonstrate some additional ground truth image modalities provided by our camera sensor in Fig. 6.

Implementing our camera sensor as a lightweight UE component has three distinct advantages over existing UE-based simulators (e.g., [5, 8–10, 12, 14]). First, our cameras can be created, manipulated, and destroyed using the same code path as any other UE object, and therefore the rest of our system architecture can remain completely decoupled from our camera implementation (e.g., we do not need to implement any hand-crafted server entry points specifically for cameras). Second, multiple cameras can be attached to multiple UE objects in any spatial configuration, either dynamically at runtime using the general-purpose functionality we expose through our programming model, or offline using any of the Unreal Editor's existing visual editing tools and scripting functionality. In our implementation, we include several pre-configured camera rigs for convenience (including several multi-view rigs), but it is straightforward to create new ones or customize our existing ones using any of the aforementioned

Fig. 4: In this application, we render images using UE's real-time deferred rasterizer (left) and offline reference path tracer (right).

methods. Third, over 600 camera parameters and rendering settings are already exposed as reflection-visible variables in UE's render-to-texture component, are inherited implicitly by our camera sensor component, and can be configured through our programming model as though they were native Python attributes, again without requiring any hand-crafted code in the rest of our system.

Rendering Fine-Grained IDs In standalone applications, UE only supports rendering 8-bit integer IDs by default [4], which is insufficient for representing fine-grained IDs in complex environments. To address this limitation, we implement a custom proxy geometry system that supports rendering fine-grained 24-bit integer IDs.

Roughly speaking, our system works by duplicating every mesh in a scene dynamically at runtime, and assigning a custom emissive material whose color encodes a unique ID to each duplicated mesh. We refer to these duplicated meshes as *proxy meshes*. We route each proxy mesh through a custom ID rendering pass that disables UE's many lighting and post-processing effects, and we take care to ensure the final rendered pixels exactly match the previously assigned emissive materials. We also take care to disable our proxy meshes in all other rendering passes. We assign a unique ID to every component-material pair, and therefore our rendered IDs can used for instance segmentation tasks (e.g., by grouping all IDs that refer to the same UE object), as well as material segmentation tasks (e.g., by grouping all IDs that refer to the same material). We assign each proxy mesh to be a child of its non-proxy counterpart, which guarantees that the poses of corresponding proxy and non-proxy meshes remain synchronized, even during complex skeletal animations. Additionally, we assign each proxy mesh to be owned by a single manager object, which simplifies the task of routing the proxy meshes to our custom ID rendering pass and disabling them in our other passes.

Creating and rendering proxy meshes is relatively lightweight, since each set of mesh vertices only needs to be stored on the GPU once and can be referenced by both a proxy and its non-proxy counterpart. However, since there is no global callback mechanism in UE for tracking spawned or destroyed objects, our system must instead keep track of scene geometry by iterating over every object every frame. Our system is disabled by default, and a user only needs to enable it if they want to render fine-grained ID images. For the scene in Figure 6, enabling our system increases overall frame time by 1-2 milliseconds per frame.

We implement our proxy geometry system as a spawnable UE object with a public interface of reflection-visible functions. Therefore, our system can be controlled using the same code path as any other UE object, and the rest of our system architecture can remain completely decoupled from it.

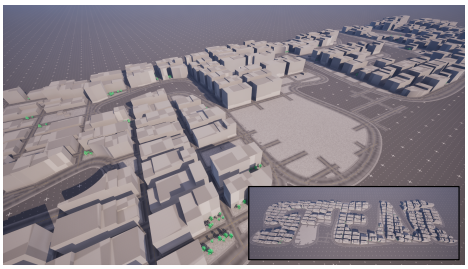

Fig. 5: In this application, we programmatically manipulate the CityGenerator sample map from Epic Games, which was newly released in UE 5.8. We control the shape of the main procedural content generation (PCG) entity in the scene to spell SPEAR. We include this application to demonstrates SPEAR's compatibility with UE 5.8, which is the latest UE version at the time of publication.

Rendering a Non-Diffuse Intrinsic Image Decomposition We provide a non-diffuse intrinsic image decomposition similar to the one found in the Hypersim dataset [11] (see Figure 2 in the main paper) by leveraging two custom rendering passes. In our first custom pass, we globally replace every material with a perfectly diffuse grey material, and we use the output from this pass to estimate a *diffuse illumination* image. In our second custom pass, we globally disable the specular component of every material to obtain a diffuse-only approximation of the final image. By subtracting our true final image from this diffuse-only approximation, we obtain the specular-only *residual* image in our intrinsic decomposition. We take care to perform all of these operations using 16-bit linear high-dynamic range images before UE has applied any tone-mapping or post-processing.

Exchanging Latency for Throughput Our camera sensor supports user-configurable rendering latencies of {0, 1, 2} frames, and in exchange for increased latency, our implementation can provide increased rendering throughput (see Table 2 in the main paper). The default behavior of our camera sensor is to incur 0 frames of latency, meaning that if the user visually modifies the scene in a `begin_frame` context, then the modifications are guaranteed to be visible in the pixel data returned by `read_pixels` in the immediately following `end_frame` context. Internally, we invoke separate subroutines that guarantee {0, 1, 2} frames of rendering latency by {single, double, triple} buffering the various intermediate steps involved in GPU-to-CPU data copying. See the implementation of our camera sensor component in our code repository for details.

4 Programming Model Details

Controlling the Unreal Editor In the Unreal Editor, there are effectively two UE *worlds* that exist side-by-side: a persistent *editor world* that is an editable representation of the currently loaded environment; and an ephemeral *game world* that is intended for previewing gameplay when the user launches a live simulation within the editor [1]. In our programming model, it is possible to

Fig. 6: From left to right: The SPEAR camera sensor can render *material IDs*, as well as various parameters from UE's physically-based shading model, e.g., *material ambient occlusion* (sometimes referred to as *cavity*), *roughness*, and *metallic* (see [6]).

control both worlds in the same Python program at the same time. For example, in Figure 3 in the main paper, we call `get_game` to obtain an object that represents the game world, but we can also call `get_editor` to obtain an object with an identical interface, except it represents the editor world and has some extensions for controlling the editor itself. When our Python client is connected to a standalone application, it is only possible to call `get_game`, and attempting to call `get_editor` results in an error.

Controlling the Unreal Path Tracer UE's offline path tracer [3] can be accessed via the reflection system, and can therefore be invoked using our programming model, both in standalone applications and in the editor (see Fig. 4).

5 System Architecture Details

An Ergonomic Interface for Unreal Objects In order to provide an ergonomic high-level interface to represent UE objects in Python, we implement a lightweight wrapper type that builds on our low-level interfaces for accessing the UE reflection system and calling SpFunctions. In our wrapper type, we override Python's default attribute resolution mechanism, which enables us to identify when a user's Python code is attempting to access a function or attribute that doesn't exist natively on the wrapper object, and dispatch to our own server entry points. This approach enables user Python code to call any reflection-visible UE function and any SpFunction as though it was a native Python function, and to access any reflection-visible UE variable as though it was a native Python attribute.

Implementing SpFunctions In Figure 8, we provide a complete code example showing how to implement a simple SpFunction, and how to call it from Python. Our code repository includes the complete implementation of our camera sensor component, which defines a more complex SpFunction that uses shared memory to transfer pixel data from the GPU to user Python code.

6 Evaluation Details

Programmability Comparison The UnrealCV+ [10, 14] code repository describes functionality to *"call any Blueprint function from Python"* using their `vbp` command, which is conceptually similar to our approach for calling UE functions from Python. However, we tried using the `vbp` command to get and set the location of a known object, and we got error messages in all of our attempts. We used the exact command formulation provided in the UnrealCV+

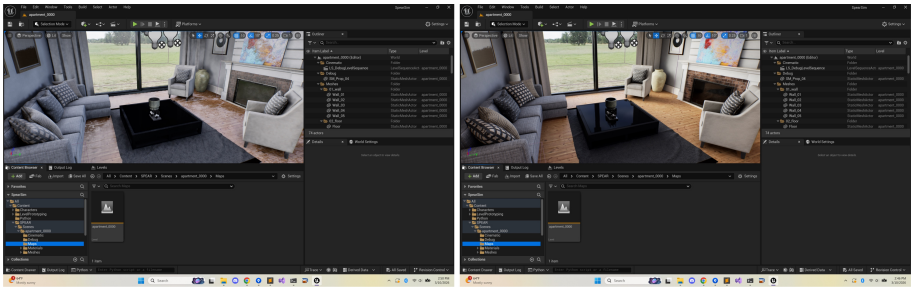

Fig. 7: Screenshots from the Unreal Editor with global illumination disabled (left) and enabled (right). By comparing these screenshots to Table 2 in the main paper, we can see that the SPEAR camera enables global illumination by default, whereas UnrealCV+ [10, 14] disables it by default.

documentation, and we tried with several different objects, using each object's name exactly as it was reported by UnrealCV+ in our command strings. Based on these findings, we concluded that exposing Blueprint functions remains a work-in-progress in UnrealCV+, and therefore we excluded Blueprint functions in our count of UE functions that are accessible in UnrealCV+.

Photorealism Comparison In Figure 7, we show screenshots of the Unreal Editor with global illumination disabled and enabled via the editor GUI. When comparing these screenshots to Table 2 in the main paper, we find that SPEAR supports more photorealistic rendering than UnrealCV+ [10, 14], which disables global illumination by default and offers no documented way to re-enable it.

Benchmarking our MuJoCo Co-Simulation Application To evaluate system performance in a setting that is more representative of typical embodied AI workloads, we benchmark our MuJoCo co-simulation application, implemented using SPEAR and UnrealCV+, while varying the number of dynamic objects and rendered views (see Table 3). During each simulation step, we apply a fixed set of forces to every dynamic object, and we render 1 or 2 RGB views at 1920×1080 resolution from a moving camera rig on a fixed path. To ensure fair comparisons, we use batched commands where possible in UnrealCV+, we use the asynchronous API in SPEAR, and we configure the SPEAR camera sensor to render with 0 frames of latency. We measure the total end-to-end frame time required to obtain rendered images in a user NumPy array, including all MuJoCo simulation time, averaged across 1000 frames.

7 Limitations

Our programming model allows users to execute custom work at the beginning and end of a UE frame, but it does not allow users to execute custom work in the middle of a UE frame (e.g., after physics has been simulated but before rendering has begun). Including mid-frame extension points in our programming model would occasionally be useful, and could be implemented in our system architecture, but we have intentionally omitted them to keep the common case

as simple as possible. The workaround is to implement mid-frame workloads in C++, enable them from Python at the beginning of a frame, and collect their results from Python at the end of a frame.

Additionally, our choice to directly expose the Unreal Engine's entire reflectable interface has some costs, in addition to its unique benefits. One such cost is that users must express their programs directly in terms of UE abstractions, and therefore user code must be aware of and respect the subtle invariants that govern the behavior of the engine. For example, in a SPEAR program, a user must ensure that a component's mobility is set to `Movable` in order for subsequent calls to `SetRelativeLocation` to have any effect. In other simulators, this kind of detail would likely be handled in a hand-crafted function provided by the simulator. Although SPEAR removes the need to write C++ code in most situations, it does not remove the need to understand the engine. The power to call any function is also the responsibility to call it correctly.

References

1. In-editor testing (play and simulate). Epic Games Developer Documentation 4
2. An overview of components. Epic Games Developer Documentation 2
3. Path tracer. Epic Games Developer Documentation 5
4. Using custom stencil to selectively bypass postprocessing. Epic Games Developer Documentation 3
5. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: CARLA: An open urban driving simulator. In: CoRL 2017 2
6. Karis, B.: Real shading in Unreal Engine 4. In: SIGGRAPH 2013 Course on Physically Based Shading in Theory and Practice 5
7. Lu, J., Huang, C.H.P., Bhattacharya, U., Huang, Q., Zhou, Y.: HUMOTO: A 4D dataset of mocap human-object interactions. In: ICCV 2025 2
8. Martinez-Gonzalez, P., Oprea, S., Castro-Vargas, J.A., Garcia-Garcia, A., Orts-Escolano, S., Garcia-Rodriguez, J., Vincze, M.: UnrealROX+: An improved tool for acquiring synthetic data from virtual 3D environments. In: IJCNN 2021 2
9. Martinez-Gonzalez, P., Oprea, S., Garcia-Garcia, A., Jover-Alvarez, A., Orts-Escolano, S., Garcia-Rodriguez, J.: UnrealROX: An extremely photorealistic virtual reality environment for robotics simulations and synthetic data generation. Virtual Reality **24** (2019) 2
10. Qiu, W., Zhong, F., Zhang, Y., Qiao, S., Xiao, Z., Kim, T.S., Wang, Y.: UnrealCV: Virtual worlds for computer vision. In: Multimedia 2017 2, 5, 6
11. Roberts, M., Ramapuram, J., Ranjan, A., Kumar, A., Bautista, M.A., Paczan, N., Webb, R., Susskind, J.M.: Hypersim: A photorealistic synthetic dataset for holistic indoor scene understanding. In: ICCV 2021 4
12. Shah, S., Dey, D., Lovett, C., Kapoor, A.: AirSim: High-fidelity visual and physical simulation for autonomous vehicles. In: Field and Service Robotics 2017 2
13. Stanford Computer Graphics Laboratory: The Stanford 3D scanning repository (1996) 1
14. Zhong, F., Wu, K., Wang, C., Chen, H., Ci, H., Li, Z., Wang, Y.: UnrealZoo: Enriching photo-realistic virtual worlds for embodied AI. In: ICCV 2025 2, 5, 6

Example application	Appears in	Unique functionality
`control_car`	Fig. 1 supp Supp. video	`find_actor_by_class` `inject_input_for_actor`

`control_city_generator`	Fig. 5 supp
	Supp. video
		`AddSplinePoint`
		`ClearSplinePoints`
		`Generate`
		`SetControlRotation`
		`SetMobility`

`control_city_sample`	Fig. 1
	Supp. video
		`FindNearestLaneLocationByName`
		`GetPlayerController`
		`K2_GetPawn`
		`Possess`

`control_cropout`	Fig. 1
	Supp. video
		`call` (Blueprint-defined functions)
		`GetNavDataForAgentName`
		`GetNavigationSystem`
		`GetOverlappingActors`

`control_electric_dreams`	Fig. 6 Supp. video	`FlushCache` `Generate`
`control_game_animation`	Fig. 1 Supp. video	`Possess`

`control_game_animation_58`	Fig. 1
	Supp. video
		`GetGameMode`
		`GetPlayerController`
		`Possess`
		`SwapPlayerControllers`

`control_hillside_sample`	Tab. 3
	Supp. video
		`GetPlayerController`
		`K2_GetPawn`
		`LoadStreamLevel`
		`SetControlRotation`

Table 1: Non-exhaustive summary of unique functionality used in our example applications that is not available in any existing UE-based simulator.

Example application	Appears in	Unique functionality

`control_metahumans`	Fig. 8	`get_components_by_class_as_dict`
		`load_class`
		`spawn_actor`

`control_stackobot`	Fig. 1 Supp. video	`GetPlayerController` `K2_GetPawn`
`getting_started`	Fig. 3	`K2_GetComponentLocation`
`import_humoto_dataset`	Fig. 3 supp	`load_class` `spawn_actor`
`import_stanford_dataset`	Fig. 2 supp	`load_class` `spawn_actor`

`movie_render_queue`	Fig. 4 supp	`AllocateJob`
		`IsRendering`
		`RenderJob`
		`SetConfiguration`

`mujoco_interop`	Fig. 7
	Supp. video
		`GetPlayerController`
		`K2_GetPawn`
		`SetControlRotation`

`render_image_hypersim`	Fig. 2
	Fig. 6 supp
		`get_component_by_name`
		`load_class`
		`read_pixels`
		`spawn_actor`

`run_mcp_server`	Fig. 9
	Supp. video
		`get_component_by_name`
		`load_class`
		`read_pixels`
		`spawn_actor`

Table 2: Non-exhaustive summary of unique functionality used in our example applications that is not available in any existing UE-based simulator (continued).

Simulator	Num. dynamic objects (1 view)			Num. dynamic objects (2 views)		
	4	16	64	4	16	64
UnrealCV+	279.4	293.7	288.8	528.5	521.4	541.6
SPEAR (ours)	**19.2**	**20.2**	**30.9**	**30.8**	**31.7**	**34.7**

Table 3: End-to-end frame times (milliseconds, lower is better) in our MuJoCo co-simulation application. SPEAR is between 9× and 17× faster than UnrealCV+ in all experimental conditions.

Implementing an SpFunction in C++

```cpp
UCLASS()
class AMyActor : public AActor {
    GENERATED_BODY()
public:
    AMyActor() {
        // create USpFuncComponent and attach to this actor
        SpFuncComponent = SpUtils::createComponent<USpFuncComponent>(this, "my_sp_func_component");
    }

    UFUNCTION(BlueprintCallable)
    void Initialize() {
        SpFuncComponent->registerFunc("my_sp_func", [this](SpFuncDataBundle& args) -> SpFuncDataBundle {

            // unpack argument
            SpArray<double> input("input");
            SpUtils::moveFromPackedArrays({input.getPtr()}, std::move(args.packed_arrays_));

            // process argument
            const std::span<double>& view = input.getView();
            SP_LOG("Input is [", view[0], ", ", view[1], ", ", view[2], "]");

            // prepare return value
            SpArray<double> output("output");
            output.setDataSource({4.0, 5.0, 6.0});

            // pack return value
            SpFuncDataBundle return_values;
            return_values.packed_arrays_ = SpUtils::moveToPackedArrays({output.getPtr()});
            return return_values;
        });
    }

    UFUNCTION(BlueprintCallable)
    void Terminate() {
        SpFuncComponent->unregisterFunc("my_sp_func");
    }

    UPROPERTY(VisibleAnywhere)
    USpFuncComponent* SpFuncComponent = nullptr;
};
```

Calling an SpFunction from Python

```python
my_actor = game.spawn_actor(uclass="AMyActor")
my_actor.Initialize()

# returns [4.0, 5.0, 6.0] as a numpy array named "output"
return_values = my_actor.my_sp_func(arrays={"input": np.array([1.0, 2.0, 3.0], dtype=np.float64)})
output = return_values["arrays"]["output"]

my_actor.Terminate()
game.destroy_actor(actor=my_actor)
```

Fig. 8: Code example demonstrating how to implement an SpFunction in C++, and how to call it from Python. We highlight the boilerplate code required to define an SpFunction in green, and we highlight the names of the function and its arguments and return values in purple. Defining an SpFunction requires a similar amount of effort as exposing a function to UE's reflection system, but with the added benefit that NumPy arrays can be passed back and forth efficiently from Python as arguments and return values.