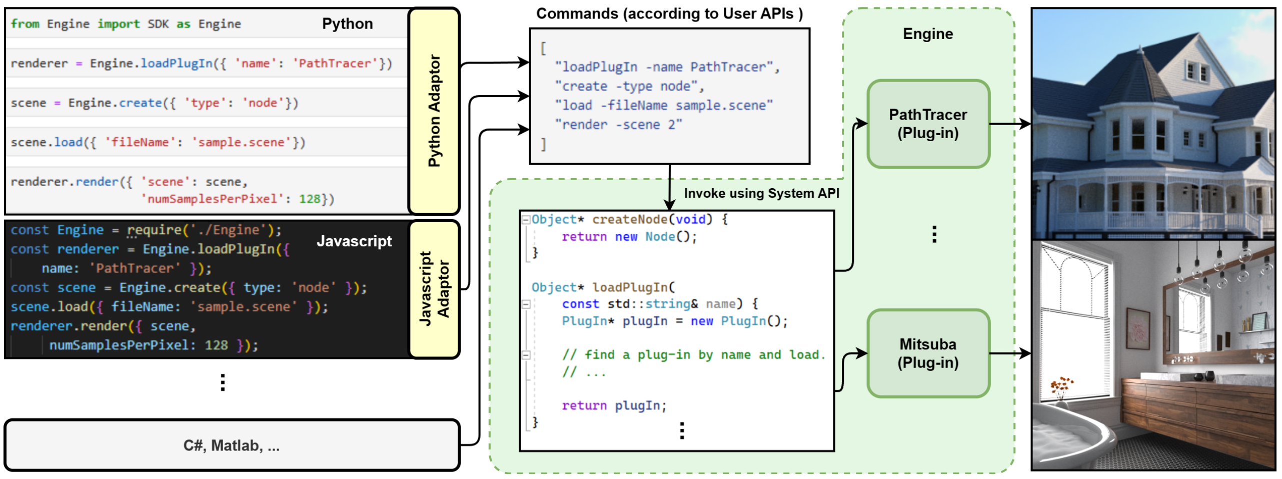
# A Highly Adaptable and Flexible Rendering Engine by Minimum API Bindings

Taejoon Kim

Electronics and Telecommunications Research Institute (ETRI), South Korea



```python
from Engine import SDK as Engine

renderer = Engine.loadPlugIn({ 'name': 'PathTracer'})

scene = Engine.create({ 'type': 'node'})

scene.load({ 'fileName': 'sample.scene'})

renderer.render({ 'scene': scene,
                  'numSamplesPerPixel': 128})
```

```javascript
const Engine = require('./Engine');
const renderer = Engine.loadPlugIn({
    name: 'PathTracer' });
const scene = Engine.create({ type: 'node' });
scene.load({ fileName: 'sample.scene' });
renderer.render({ scene,
     numSamplesPerPixel: 128 });
```

```
[
  "loadPlugIn -name PathTracer",
  "create -type node",
  "load -fileName sample.scene"
  "render -scene 2"
]
```

```cpp
Object* createNode(void) {
    return new Node();
}

Object* loadPlugIn(
    const std::string& name) {
    PlugIn* plugIn = new PlugIn();

    // find a plug-in by name and load.
    // ...

    return plugIn;
}
```

**Figure 1:** *The proposed rendering engine is designed to be easily embedded in various development environments. An adaptor for each environment converts undefined functions into corresponding command strings. The engine then parses the command strings and executes proper procedure using the System API. The upper right image shows the rendering result using a simple path tracer, while the bottom right image shows the rendering result of another scene using* Mitsuba *render plugin.*

**Abstract**

*This paper presents a method for embedding a rendering engine into different development environments with minimal API bindings. The method separates the engine interfaces into two levels: System APIs and User APIs. System APIs are the low-level functions that enable communication between the engine and the user environment, while User APIs are the high-level functions that provide rendering and beyond rendering functionalities to the user. By minimizing the number of System APIs, the method simplifies the adaptation of the engine to various languages and platforms. Its applicability and flexibility are demonstrated by the successful embedding the engine in multiple environments, including C/C++, C#, Python, Javascript, and Matlab. It also demonstrates its versatility in diverse forms such as CLI renderers, Web GUI framework-based renderers, remote renderers, physical simulations, and more, while also enabling the easy adoption of other rendering algorithms to the engine.*

**CCS Concepts**

• ***Computing methodologies*** $\rightarrow$ ***Rendering;***

## 1. Introduction

Advances in rendering engines and their APIs enable users to visualize or render scenes easily without requiring a high level of rendering expertise. Recently, some monolithic engines such as Unreal Engine, Unity, Blender, and Autodesk Maya/3ds Max have been widely used in games, films, and animation. Each provides ways for users to describe scenes so that the users can work within the engines' proprietary software, typically called an *Editor*. Another type of engines [PJH16, NDVZJ19] offers APIs that allow users to embed the engine into their own applications.

DOI: 10.2312/egs.20241027

Despite the diversity of advances in development environments, developers are often forced to use only a few, such as C++ or Python, by the most API-based engines due to the rendering nature; rendering requires numerous APIs. If developers want to support a specific environment or language that differs from the engine's implementation, they must undergo a tedious process known as *binding*, which links each API between two environments. For example, assume there's an API implemented in C++, `createMesh(...)`. To call this API from Python, one needs to create a corresponding function in Python and then somehow *bind* this Python function to the C++ implementation. Moreover, the function parameters should be converted to a format that C++ can handle, and the return value should be converted properly as well. Depending on the level of API detail, tens to hundreds of bindings for various APIs (e.g., `getMeshPosition(...)`, `setImageWidth(...)`) are typically required for just a single environment. To support another environment, such as Node.JS (Javascript), an additional extensive binding effort is again required.

**Previous Work** has shown that, for these reasons, many engines including Unreal Engine and PBRT [PJH16] only support C/C++, while some others such as Mitsuba [NDVZJ19] and Blender additionally support a few scripting languages for convenient development. This kind of trend (restrictions in using a rendering engine) tends to divide the way of using rendering engines into only two manners; users either operate within the provided engine software (the *Editor*) or develop within a limited range of environments or languages. Recently, Khronos ANARI [SGA*22] was released, making it easy to render scenes with a few dozen high-level APIs. However, it is not yet ready to be used in a wide range of development environments because it requires support from the respective communities. The proposed method allows ANARI to be used in a variety of environments with a single integration. Also, the use of Foreign Function Interface (FFI) related tools might be considered. SWIG [Bea96] automatically generates binding codes between C/C++ and various kinds of programming languages. However, the code generation and redistribution might be necessary if any of the APIs are changed or added. Networking protocols such as REST or gRPC [BZP*22] are also considerable but they have unnecessary network overhead even if we don't communicate over the network.

## 2. Method

This paper proposes an approach to easily support various development environments by reducing the number of API bindings. The main idea involves organizing the APIs into two levels; *System APIs* and *User APIs*.

System APIs are low-level functions that facilitate communication between different environments. This means that it is necessary to implement as many bindings as the product of the number of System APIs and the number of environments. User APIs, on the other hand, are high-level functions that provide rendering and various functionalities. By integrating these two API levels while minimizing the number of System APIs, an efficient process can be established. This process involves configuring all functionalities using User APIs, which are then requested to the engine via a single System API, called a *command*. The following is an abstract of the command:

$$Result \leftarrow \mathsf{Command}(target, command\_string)$$

An User API with its parameters are converted to a string and given as the *command_string*. The use of strings has several advantages: 1) they are accepted by almost all environments, 2) a string itself can contain various data types, 3) strings are highly readable, and 4) there are many tools available for generating or parsing strings.

Followings are some examples of creating a mesh object that satisfy the User API.

```
/* JSON */ "{ \"command\": \"create\", \"type\": \"mesh\" }";
/* CLI  */ "create -type mesh";
```

Based on the above approach, the proposed rendering engine system consists as shown in Fig. 2. The engine basically manages the engine objects such as meshes, materials, cameras, etc. In addition, the engine can be extended and customized by a set of plugins. A plugin is a dynamically loadable module (i.e., a shared library) that implements its own functions such as rendering frames, importing/exporting 3D models, handling scripting languages, deforming geometries, and so on. More and detailed examples of the plugins are discussed in Sec.2.3.

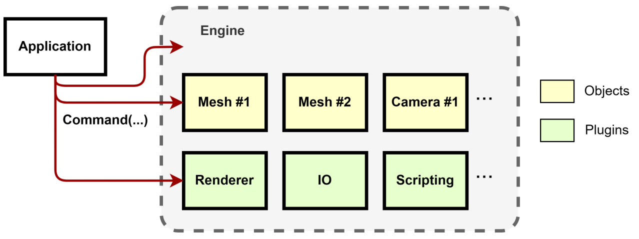


**Figure 2:** *This figure illustrates how the application commands are applied to the engine. The yellow boxes represent engine objects and the green boxes represent plugins. Commands are applied directly to the engine or passed to engine objects or plugins, depending on the specified target.*

Although the system proposed so far may make the rendering engine appear flexible, there are significant issues when it comes to applying it to actual applications. These issues can be categorized into problems from the user's perspective and those from the developer's perspective. From a user perspective, it can be quite cumbersome to create strings for the desired functionality. For example, instead of using a command with a string to change a property of a camera object, as shown below

```
Command(camera, "setAspect -aspect 1.5", ...);
```

users would prefer to call it in the form of a class such as:

```
camera.setAspect({ aspect: 1.5 });
```

that is more productive. To accomplish this, the presented adaptors for each environment provide methods by using *dynamic properties* (Sec.2.1). On the other hand, from a developer's point of view, the extra effort is required to interpret or parse the requested strings and map them to the corresponding functions. To solve this problem, this paper proposes to use an automation tool (Sec.2.2).

In summary, a user sets up a request using a User API, and an adaptor then converts it to a string before forwarding it to the engine through single System API, `Command()`. The engine then parses the command strings and executes proper procedure (Fig. 1).

### 2.1. Adaptors

Fortunately, many modern high-level languages support some form of dynamic properties in their own way. A dynamic property intercepts accesses to any property or invocations of any method, and then allows you to process additional actions. By using these features, an adaptor automatically converts each property access and method invocations to command strings. Here is a Javascript code example that implements this using *Proxy*. Other examples (Python, C#, Matlab) can be found in the supplementary material.

```
const object = new Proxy(instance, {
  get: function (target, property) {
    return function (args) {
      const commandString = JSON.stringify(Object.assign({
        command: property}, args));
      // Invoke a System API.
      return Command(target.ID, commandString);
    }}});
```

In addition to supporting dynamic properties, an adaptor converts pointer values into usable forms. Specifically, a pointer of an array in the engine is converted to typed array in the user's environment. Also, a function pointer is able to be converted to a callback function or a lambda function.

Because of the simplicity of the System APIs, developing an adaptor might take less than a week in practice if he or she has enough experience with the environment. In the case of Python, the source code of an unoptimized adaptor has around 100 lines for core implementations and less than 500 lines total; most of the lines are for typical type mappings.

### 2.2. API Designer

In practice, defining an User API involves some systematic tasks. We need to write a function that implements an User API, make a parser that identifies whether a given command string is for the API or not and parses other parameters, and write a code to call the working function inside the `Command()` function. To reduce these cumbersome processes, this paper proposes an automated tool named *API Designer*.

When an engine developer or a plugin developer defines an API name, parameters, and descriptions for the API using API Designer, the tool generates a skeleton code for the API implementation with documentation comments, and codes to link the System API (Please see the accompanying video).

### 2.3. Plugins

A plugin is a module that can be loaded into the proposed engine at runtime. Any module that implements the `Command()` and exports the API so that the rendering engine can get the address of the function (using *dlsym* on POSIX, *GetProcAddress* on Windows) could be a plugin. Unlike plugins in other systems that are tightly bound to the main engine, the proposed method is much more open to the plugin form. Mitsuba2 [NDVZJ19], for example, has a plugin-based architecture and each of the shapes, materials, and integrators is a plugin module. In order to add another shape as a plugin, it should implement every interfaces with exact parameter types defined by Mitsuba2. Moreover, it is not straightforward that how to support other kinds of plugins.

As discussed earlier, interpreting a request, i.e., parsing a command string has a performance overhead. To see the amount of overhead, an experiment that invokes the simplest function, getting the object ID, had been performed. When a plugin exports a separate function (such as a System API) to get an object ID, it takes less than 1 ms for 100,000 times invocations on an Intel i9 CPU @ 3.60GHz machine. But it takes about 120 ms for 100,000 times invocations via the `Command()` API. Most of the time is spent for parsing the result string. As we can see, it might not be a good choice to make plugins for each engine object which is what Mitsuba2 does. Instead, we could focus on high-level functions for a plugin and make some custom objects (if necessary) inside the plugin module. The ray-intersection test for a mesh in ray tracing has to be performed incredibly often which might not be appropriate for our API, but for the rendering process which is done usually only once per frame.

## 3. Applications and Results

Due to high flexibility based on unified command interface, the proposed engine significantly increases the productivity of developing various types of applications. In this paper, the utility of the proposed method is evaluated by directly applying it to various applications (Please see the accompanying video and supplementary material which has more diverse applications.).

### 3.1. GUI using web frameworks

Recently, one of the most powerful and popular development method of GUI is to use web frameworks. Front-end frameworks such as React and Vue.JS and corresponding component libraries (UI elements such as buttons, text fields, ...) allow developers to develop high-quality GUIs with high productivity. These frameworks can be used not only in web browsers but also in desktop applications using web-to-desktop-frameworks such as Electron (combination of the Chromium browser engine and Node.JS). Since the proposed engine works with Javascript which is a scripting language in most web environments, these advantages can be actively utilized. Please note that the commands of the engine run natively, the performance within single command is not a concern.

### 3.2. Embedding other Renderers

Although a rendering plugin can have its own rendering algorithm, it would be very useful if we are able to adopt existing renderers.

The following well modularized renderers are good examples to embed. In practice, it only takes around or less than a week to make a renderer as a plugin if it is already familiar with. Most of the time is spent converting scene objects such as geometries, cameras, lights and materials. For materials, since it is beyond the scope of this paper, only a part of Disney's principled material [MHH*12] is used, but extending it to support other types is trivial.

**PBRTv3**: The PBRT rendering pipeline automatically saves the rendered film to only a file after all tasks have been completed, but this workflow is not suitable for our API. To address this issue, the original source code is modified to save the film into a memory buffer. Also, in order to directly retrieve scene objects from an imported scene, additional code is required because the PBRT is not designed for this purpose.

**Mitsuba2**: Thanks to its well-defined software architecture, the original source is used without any modification. Obtaining scene objects from an imported scene or objects attributes can be done using `traverse()` or `to_string()`.

## 3.3. Remote and distributed rendering

By designing a special plugin, named a *remote plugin*, the remote rendering could be achieved without any additional work; just load a rendering plugin as a remote plugin. The remote plugin acts as a bridge, forwarding any request from the user to the remote server, and also passing the results back to the user. This method is useful when the plugin is not able to run on the local machine due to OS mismatch or lack of required hardware, but can run on the remote server. This feature enables the distributed rendering easy. The only thing to consider is how to distribute the frames (or tiles in a frame) to multiple rendering servers.

## 3.4. Physical simulation

A physical simulation can be achieved by using a plugin that provides the simulation functionalities. The plugin is responsible for creating/managing the physical objects, handling the physical parameters, handling the interactions between the user and the physical objects and deforming the geometries over time. Then the main engine simply calls some User APIs to the plugin to perform the simulation. Similar to the rendering plugins, any kind of implementation of the physical simulation could be applied and the user could perform the simulation in the same manner if all plugins share same APIs. In this paper, a particle-based simulation [MMCK14] has been successfully tested.

## 3.5. Automatic unit test

One of the advantages of managing APIs directly is that it makes easy to automate unit tests. Moreover, since the engine can run in a variety of environments, the testing tools also have a wide variety of choices. In this paper, Mocha, a Javascript testing framework running on Node.JS, has been chosen. The test framework reads predefined test cases (arguments and expected outputs) for each User API and then checks whether the running outputs match to the expected outputs or not.

## 4. Conclusion and Future Work

In this paper, an approach is proposed to easily embed the rendering engine in various environments by separating the engine interfaces into user APIs (providing semantic functions to users) and system APIs (for syntactic communication between different environments where the engine is implemented and the engine will be used) and then minimizing the number of system APIs. This is much more productive than the bindings of other rendering engines which have usually hundreds of APIs. By applying the proposed approach to plugins as well, we are able to develop a plugin system with virtually unlimited functionality, thereby significantly enhancing the engine's adaptability and flexibility. As a result, many kinds of applications could be easily created.

The source code or web sandbox interface associated with this work is planned to be released in the future. This will ensure that the code is comprehensive, well documented, and thoroughly tested, to best serve the research community. In addition, a detailed performance analysis with techniques related to FFI, such as SWIG [Bea96] and gRPC [BZP*22], will be conducted.

## Acknowledgment

3D scenes of the Victorian style house, bathroom, hotel in San Miguel (in the accompanying video) are courtesy of MrChimp2313, Mareck and Guillermo M. Leal Llaguno respectively. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2021-0-00193, Development of photorealistic digital human creation and 30fps realistic rendering technology)

## References

[Bea96] BEAZLEY D. M.: SWIG: an easy to use tool for integrating scripting languages with C and C++. In *Fourth Annual USENIX Tcl/Tk Workshop 1996, Monterey, California, USA, July 10-13, 1996* (1996), Diekhans M., Roseman M., (Eds.), USENIX Association. 2, 4

[BZP*22] BOLANOWSKI M., ZAK K., PASZKIEWICZ A., GANZHA M., PAPRZYCKI M., SOWINSKI P., LACALLE I., PALAU C. E.: Eficiency of REST and grpc realizing communication tasks in microservice-based ecosystems. *CoRR abs/2208.00682* (2022). `arXiv:2208.00682`. 2, 4

[MHH*12] MCAULEY S., HILL S., HOFFMAN N., GOTANDA Y., SMITS B. E., BURLEY B., MARTINEZ A.: Practical physically-based shading in film and game production. In *International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2012, Los Angeles, California, USA, August 5-9, 2012, Courses* (2012), ACM, pp. 10:1–10:7. 4

[MMCK14] MACKLIN M., MÜLLER M., CHENTANEZ N., KIM T.: Unified particle physics for real-time applications. *ACM Trans. Graph. 33*, 4 (2014), 153:1–153:12. 4

[NDVZJ19] NIMIER-DAVID M., VICINI D., ZELTNER T., JAKOB W.: Mitsuba 2: A retargetable forward and inverse renderer. *Transactions on Graphics (Proceedings of SIGGRAPH Asia) 38*, 6 (Dec. 2019). 1, 2, 3

[PJH16] PHARR M., JAKOB W., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*, 3rd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016. 1, 2

[SGA*22] STONE J. E., GRIFFIN K. S., AMSTUTZ J., DEMARLE D. E., SHERMAN W. R., GUNTHER J.: Anari: A 3-d rendering api standard. *Computing in Science & Engineering 24*, 02 (mar 2022), 7–18. 2