1. Related work

While previous rendering studies have mainly focused on visual effects which could not be expressed before [WFS22, HMC*22] or improving performance [LGXT17, FHL*18, ZZC*22], in this paper, we will focus on the usability and flexibility of the rendering engines.

1.1. Monolithic Rendering Engines

The most widely used rendering engine in the industry and productions is the monolithic engine. It is an application itself while providing rendering, scene modification and animation. Besides, **Autodesk Maya/3ds Max** [Aut23b, Aut23a] and **Blender** [Ble23a] provide quite a lot of professional tools for modeling, sculpting and texturing. Unlike those engines, **Unreal Engine** [Epi23] and **Unity** [Uni23] are specialized to build game applications.

In order to render a scene using such monolithic engines, we have to use the engine application. It is very difficult for users who already have an application to embed those engines into their applications for rendering. It means that it is not easy for the product to provide features beyond those provided by the engine. But the presented engine can be easily embedded to existing applications as a module.

1.2. Renderer as a Module

In fact, various renderers could be embedded into another applications as well as the renderer itself is a standalone application. Some of them provide rendering libraries as is, others are open-sources and each could become a library with only minor modifications.

PBRT [PJH16] is an well known open-source renderer which is targeted to the education with accompanying books. Due to the renderer's clarity, physical accuracy and usability, several renderers including **LuxRender** are built based on **PBRT**. It is well modularized and provides APIs to be a library for another application, but only **C++** environments can directly import the module. **Mitsuba** [NDVZJ19] focuses on retargetable rendering for research purpose. Recently, it has achieved higher performance based on **Dr.Jit** [JSRV22]. **Mitsuba** is also well modularized to be single library and, moreover, recent version provides the engine as a **Python** module that only **pip** is required for the installation. However, the proposed method supports a lot of environments as well as **Python**. Also, there are other numerous renderers which can be used as a module such as **Google Filament** [Goo23], **Ogre3D** [The23], **Amazon Lumberyard** [Ope23] and **Cycles** [Ble23b], but most of them support only **C++** or few additional languages.

Each of all above renderers could be a *render plugin* very easily. Sec. 2.1.3 shows how to integrate existing renderers and the proposed engine with some examples.

1.3. Foreign Function Interface

A mechanism of calling functions in a programming language from another language is called a foreign function interface (FFI). In many cases, a high-level language or a script language calls **C/C++** functions in their own way. For example, **Java** provides a **Java Native Interface (JNI)** which enables **Java** code to call **C/C++** functions. **Python** provides **ctypes** module that makes communications between **Python** and **C/C++** very easy. The proposed engine also uses each provided FFI to support various programming languages and environments. However, the problem that the rendering has too many APIs to binding still remains.

SWIG [Bea96] automatically generates the binding codes between **C/C++** and various kinds of programming languages. It supports most **C/C++** data types including classes and pointers. By using this tool, above huge number of rendering API bindings might easily be done. However, if any of the APIs is modified or added, the code generation should be performed each time and also redeployed. But with proposed method, it is not necessarily because the System API for binding will not be changed. Also, since the engine manages the APIs directly (managing list of APIs, signatures of parameters/outputs and descriptions) it could implement various applications including automatic documentation, automatic unit test and operations playback. These will be discussed in Sec. **??**.

Another approach for the FFI is using networking protocols such as **REST** or **gRPC** [BZP*22]. Although such protocols can be applied to our problem, they have unnecessary networking overhead even we don't communicate over the network. Rather, the proposed method could be extended to such protocol to support the remote procedure calls. We will see that in Sec. 2.4.

2. Plugins and Renderers

A plugin is a module that can be loaded to the proposed engine in runtime. Any module 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 shapes, materials, 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.

In some cases, a plugin module needs to access some functions of the engine core. For example, if a plugin modifies an engine object such

as a mesh, the plugin needs to refer the engine core since the engine objects are handled by the core. Although we could use the same API, `Command()`, to access the object, however, a far more efficient way might be necessary depending on the access patterns. Recalling the previous example of ray tracing, if a plugin provides a rendering function based on ray tracing, the plugin need to perform ray-intersection tests to the engine objects in numerous times. To address this issue, the engine core provides a direct access to plugins by sharing signatures of functions or classes (e.g., header files in **C/C++**). Accordingly, the plugin has to be implemented in the same language as the engine core to use the direct access.

In the remainder of this section, we discuss various kinds of plugins that could be implemented by proposed method.

2.1. Render Plugins

Rendering process can be simplified to a single command, `render(scene, camera)`, for example. It takes a target scene to be rendered and a camera as a viewer. The output of the command is an image storing the rendering result. Some renderers [PJH16, NDVZJ19] let the camera have an image as a property (usually named as a film or a sensor) to simulate physical photography. Based on the definition, it can be interpreted to an User API as follows:

- **Description**: Render an image of a scene.
- **Command**: *render*
- **Parameters**
  - *scene*: Target scene object.
  - *camera*: A camera object representing the view point.
  - *options*: Miscellaneous options depending on renderers.
- **Output**
  - *image*: Rendered image object.

Above User API can be invoked by the System API, `Command()` as follows:

```
Command(renderPlugInID, "render" +
    " -scene " + sceneID +
    " -camera " + cameraID, ...);
```

Since this kind of procedure is quite trivial, it would not be discussed anymore in the rest part of this paper.

From now on, several references of rendering plugins which implement the *render* API are demonstrated. Because they share the same API, the users can perform various renderer in same manner and switch very easily; just load another plugin. Most of the reference implementations are straightforward, so it seems that further explanations might not be necessarily. However, we would like to clarify that the proposed method is well suited to the most rendering scenarios in practice.

2.1.1. Renderer using graphics libraries and shaders

A graphics library (e.g., **OpenGL**, **Vulkan**, **Direct X**, or **Metal**) is a set of APIs which provides access to GPUs for the raster-based rendering. As the first rendering plugin, we will look at an implementation using one of the graphics library. In this case, it is relatively simple to implement the *render* API; 1. get all geometries from the given scene, 2. convert them to meshes and feed to a GPU, 3. draw all meshes and finally, 4. copy the framebuffer to an image object. For realtime engines, it might be better option that render directly to a device context given by the operating system rather than copy the framebuffer to host memory. If a material is assigned to a mesh, then the corresponding shader might be set right before drawing the mesh. The overhead of proposed System API calls (including parsing the command string) is not concerned because usually it occurs only once per a frame.

2.1.2. Path tracer

In order to implement a path tracer, it is required that develops an *integrator* which computes radiance along ray paths between lights and the camera. A plugin for the path tracer could handle such an integrator. One of the major things to be considered is that there should be a way to handle intersections between geometries and rays. If each type of geometry could perform the intersection test between rays and its own, the plugin might exploit the routine during the integration.

For interactive engines, per frame rendering may not be appropriate unless the number of samples per pixel is small. To provide immediate response, the *render* API implementation launches a thread and performs the path tracing in the background. For each *render* API call, it update the output image object instead of rendering from scratch unless the scene or the camera is not changed. In other worlds, it supports the progressive rendering using only the single command.

2.1.3. Using other renderers

Although a plugin can have its own rendering algorithm, it would be very useful if we are able to adopt existing renderers. Following well modularized renderers are good examples to be embedded. 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 to convert scene objects such as geometries, cameras, lights and materials. Of course, if we import a scene with the renderer's own format, the objects conversion is not necessary. Among the various types of conversions, the most cumbersome task is converting or supporting a number of materials. Since that is not the scope of this paper, only a part of Disney's principled material [MHH*12] is used, but extending to support other types are trivial.

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

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

Cycles: Since the **Cycles** renderer was originally developed for **Blender**, there are some limitations in handling scene objects in detail. The main issue is that it is not easy to fully utilize the rendering ability with its own scene file format (**.xml**) in current version of **Cycles Standalone**. The best way to take full advantages of the renderer's features is importing the **Blender**'s file format (**.blend**). However, the **Blender** is not designed to be as a module directly nor does not provide any SDK to import/export the file format. But fortunately, it provides a **Python** API, named **bpy**, which makes the whole engine as a **Python** module. We can run any **Python** script on the proposed engine if we have a **Python** runtime plugin which will be discussed later. Therefore, the **Blender**'s file format could be imported indirectly via the **Python** runtime plugin and the **bpy**.

2.2. I/O Plugins

An I/O plugin is a module that implements both `load()` and `save()` User APIs. They take a destination (or a source) engine object and a file path as the arguments. The I/O plugin either loads a scene, meshes, or images from file into engine objects and vise versa. This kind of plugins make the engine much more practical. For example, if there is a plugin that use **OpenImageIO** to read and write images, then the engine can handle most image files simply by loading the plugin. Recalling that some rendering engines often give up to support various types of image I/O because of cumbersomeness of linking external image libraries to their project, we can see how useful that the proposed methods solve the issue easily.

Please note that some plugins using other renderers that we discussed earlier have routines for reading and writing their own scene format, already. Therefore, a rendering plugin also can be a I/O plugin which makes a scene from a renderer is able to used in another renderer; load a **PBRT** scene by using the **PBRTv3** plugin then render the scene with **Mitsuba2** renderer.

2.3. Script Runtime Plugins

Widely used renderers support some kinds of scripting to control or modify scene objects at runtime, especially in game engines for their game logics. In our engines, the scripting can be easily supported by making script runtime plugins. The plugin implements `run()` User API which takes a script string of a specific language. To verify the claim, two kinds of script runtime plugins have been implemented. The first one accepts **Javascript** sources by using the **Google V8** engine which is widely used in a number of applications including **Google Chrome**, **Microsoft Edge** and **Node.JS**. Another plugin accepts **Python** codes by using the **CPython** library.

Supporting script languages is useful not only for the control logics but also to enhance the adaptability and the flexibility of the engine. We already have seen that the **Python** runtime plugin is useful to handle the **Blender** scenes. Moreover, the **Python** runtime plugin makes it possible to run numerous AI algorithms within the engine because, currently, major AI applications are written in the **Python** language. Also, we will see how important the **Javascript** runtime plugin is to GUI integration. Please note that supporting a script language is different from supporting the engine in the script runtime. For example, the engine can be run on a **Node.JS** which is a **Javascript** runtime while the engine can interpret **Javascript** scripts.

2.4. Remote Plugins

On top of all kinds of plugins, an interesting extension, called *remote plugin*, could be applied. When an user try to load a plugin with specifying a remote URL, the engine core set a bridge between the user and the remote server rather than loading the plugin. The bridge passes any request from the user to the remote server, and passes back the results to the user as well. This method is useful when the plugin is not able to run in the local machine because of the OS mismatch or absence of required hardware, but the remote server.

Please not that it is not necessary to limit the request target to a remote server (or servers). Same mechanism can be applied to launch a separate process for a plugin. In this case, an IPC (Inter-Process Communication) might be better than a socket which is usually chosen for

remote communications. This launching a separate process for each plugin is useful when it must run in parallel while it requires private address space and resources.

3. Adaptors

The main paper contains only a Javascript example of adapters, but here a wider variety of examples are presented.

3.1. Python

Binding of the System APIs is can be done with **ctypes** library. **Python** supports the dynamic properties by using a set of **magic methods** including `__getattr__()` and `__setattr__()`. Here is a code example that uses the magic method.

```
class Object:
    def __getattr__(self, property):
        def func(args):
            args['command'] = property
            commandString = json.dumps(args)

            # Invoke a System API.
            return Command(self.ID, commandString)

        return func
```

Converting pointer values can be easily done using the **ctypes**, too.

3.2. C#

Thanks to close relationship between **C#** and **C++**, the API binding is relatively easy using `DllImport()` and *data marshaling*. Also, the concept of dynamic properties almost corresponds to **DynamicObject** class of **C#**. Here is a code example that uses the **DynamicObject** class.

```
public class Object : DynamicObject {
  public override bool TryGetMember(
    GetMemberBinder binder, out dynamic result) {
    dynamic func(dynamic args) {
      args.command = binder.name;
      string commandString = ToJSON(args);

      // Invoke a System API.
      return Command(this.ID, commandString)
    }
    result = func;
    return true;
  }}
```

3.3. Javascript and Web

Unlike **Python**, **Javascript** does not have the standard runtime so that each target runtime requires System APIs bindings separately. In the case of **Node.JS**, the binding could be done by building a native addon using the **Node-API** (also none as **N-API**). For web browsers, there might be two options. One is to compile engine and all plugins into **WebAssembly** so that the browsers can directly load the complied binaries. In order to use this method, not only the engine and plugins but also external library the plugins use must be compiled into **WebAssembly**, too. Another options is to operate a server which runs the engine and communicate with the browsers using HTTP requests or sockets. Please note that our string based, unified System API is quite suitable for the web communications.

Support for dynamic properties using *Proxy* appears in the main article. Converting pointer values is slightly more complicated than **Python** or **C#** because **Javascript** does not allow to use pointer values directly, or, pointer values of remote servers are meaningless. For methods of embedding the engine such as **Node.JS** and **WebAssembly**, native supports (e.g., **N-API**) for handling pointers can be used. For the case of using a remote server, some implementations to synchronize the data are necessary.

3.4. Matlab

Matlab uses two builtin functions, `loadlibrary()` and `calllib()`, to invoke **C** functions. In order to specify the function signatures, additional a header file is necessary. Similar to **Javascript** on **Node.JS**, handling pointer values could be done by native supports; building **C** source code for accessing pointers with **C Matrix API**. Dynamic properties could be achieved by customizing object indexing. When it overloads `subsref()` we could handle operators of a class object. But this method is somewhat tricky because dots as well as parentheses and braces are needed to be handled. Here is a code example that uses the customizing object indexing.

```
classdef Object
  function varargout = subsref(obj, s)
    switch s(1).type
      case '.'
        if length(s) >= 2 && strcmp(s(2).type,'()')
          args = s(2).subs{1};
          args.command = s(1).subs
          commandString = jsonencode(args)

          % Invoke a System API.
          result = Command(self.ID, commandString)
          [varargout{1:nargout}] = result
        end
```

4. More Applications and Results

This section presents other applications and results not presented in the main paper.

4.1. Remote rendering

By using the remote plugin which is introduced earlier, the remote rendering could be achieved without any additional work; just load a rendering plugin as a remote plugin. But for interactive remote rendering such as cloud gaming services, some tough developments are required. The major challenge is transmitting each frame from the remote server to local machine in realtime because the size of the raw image data is too large to transmit; it takes around 200 MB per a second for 30 FPS, non HDR, full HD frames. Therefore, it is necessary to encode/decode the frames to lower the network overhead.

For efficient realtime remote rendering, two plugins are introduced: one is an I/O plugin for handling videos and the other is a plugin which implements a streaming service using the video plugin. Of course, the video plugin can be used for other purposes, for example, supporting video sources as inputs or simply generating a video file from some rendered frames. The plugin for streaming service is able to run either as a server mode or a client mode. In server mode, it encodes a specific image object periodically to the target frame rate in a separate thread. Similarly, in client mode, it receives the encoded data from the remote server then decodes the data into a specific image object periodically in the background. Accessing that image object yields the most recently rendered result on the remote server.

4.2. Distributed rendering

Unlike the remote rendering, the network overhead for transmitting frames is no longer an issue with a distributed rendering because the rendering time for each frame is expected to be much longer. Therefore, a distributed rendering could be achieved easily by using the remote plugin. The only thing to consider is how to distribute the frames (or tiles in a frame) to multiple rendering servers.

4.3. Operations playback

Another advantage of managing APIs directly is that all executed operations can be replayed later in same order. If the engine writes the command target and string for each command to a file, then it is able to read the file and execute command by command in later runtime. This method helps to reproduce the errors that have been occurred previously. Not only for the errors from the engine developers but also the errors from the end users could be reproduced. However, if any operation or series of operations have any racing condition, or, the outputs for same command are nondeterministic, the proposed method might not be usable for that purpose.

4.4. Attaching and debugging

If the engine opens a kind of IPC channel at startup, a debugging application can attach to the application that the engine runs. When the channel receives any request in User API form, the engine process the request then send back the result via the IPC channel. The debugger

simply connects to the channel then can send any request at any time. Please note that this asynchronous processing might cause some hazards depending on situations. In this case, additional producer-consumer implementation might be necessary. After all, such seemingly difficult task can be easily implemented thanks to the proposed method.

4.5. Universal applications

If the engine supports at least one script runtime via a plugin and there is a repository server for downloading libraries and resources, a kind of *Universal applications* could be created. An universal application consists of the engine core as an executable, a script runtime plugin, a plugin for network communications and a script file. The executable interprets the script file and download plugins and/or resources if necessary.

Depending on what is written in the script file, this program works completely differently from other universal applications. For example, following script makes the application work as a 3D converter.

```
Engine.loadPlugIn({name: 'FBX', global: true});
Engine.loadPlugIn({name: 'OBJ', global: true});
Engine.loadPlugIn({name: 'PLY', global: true});
const node = Engine.create({type: 'node'});
node.load({fileName: args.input});
node.save({fileName: args.output || 'output.fbx'});
```

Assuming that there are multiple I/O plugins to support various types of 3D formats, above scripts load them in advance and make them global to one of them be automatically selected by the extension of given file name. Then load a file into a *node* object and, finally, save to another file. Input and output file names follow the command arguments of the executable while output file name is *'output.fbx'* if it is not given.

As can be seen, each universal application becomes a different application with its own functions depending on the contents of the script file. And all files except the script file are shared by all universal applications. Therefore, just sharing a small-sized script file can have the effect of sharing an application. This feature is seems to be similar to well-known package management systems such as **Node Package Manager(NPM)** for **Node.JS** or **pip** for **Python**. But it can be easily achieved by proposed methods.

References

[Aut23a] AUTODESK: 3ds max, 2023. URL: http://autodesk.com/3ds-max. 1

[Aut23b] AUTODESK: Maya, 2023. URL: http://autodesk.com/maya. 1

[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. 1

[Ble23a] BLENDER FOUNDATION: Blender, 2023. URL: https://www.blender.org/. 1

[Ble23b] BLENDER FOUNDATION: Cycles, 2023. URL: https://www.cycles-renderer.org/. 1

[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. 1

[Epi23] EPIC GAMES: Unreal engine, 2023. URL: https://www.unrealengine.com/. 1

[FHL*18] FASCIONE L., HANIKA J., LEONE M., DROSKE M., SCHWARZHAUPT J., DAVIDOVIC T., WEIDLICH A., MENG J.: Manuka: A batch-shading architecture for spectral path tracing in movie production. *ACM Trans. Graph. 37*, 3 (2018), 31. doi:10.1145/3182161. 1

[Goo23] GOOGLE: Filament, 2023. URL: https://google.github.io/filament/. 1

[HMC*22] HUANG W., MERZBACH S., CALLENBERG C., STAVENGA D., HULLIN M. B.: Rendering iridescent rock dove neck feathers. In *SIGGRAPH '22: Special Interest Group on Computer Graphics and Interactive Techniques Conference, Vancouver, BC, Canada, August 7 - 11, 2022* (2022), Nandigjav M., Mitra N. J., Hertzmann A., (Eds.), ACM, pp. 43:1–43:8. doi:10.1145/3528233.3530749. 1

[JSRV22] JAKOB W., SPEIERER S., ROUSSEL N., VICINI D.: Dr.jit: A just-in-time compiler for differentiable rendering. *Transactions on Graphics (Proceedings of SIGGRAPH) 41*, 4 (July 2022). 1

[LGXT17] LEE M., GREEN B., XIE F., TABELLION E.: Vectorized production path tracing. In *Proceedings of High Performance Graphics, HPG 2017, Los Angeles, CA, USA, July 28 - 30, 2017* (2017), ACM, pp. 10:1–10:11. doi:10.1145/3105762.3105768. 1

[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. 3

[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

[Ope23] OPEN 3D FOUNDATION: Open 3d engine (successor of amazon lumberyard), 2023. URL: https://www.o3de.org/. 1

[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

[The23] THE OGRE TEAM: Object-oriented graphics rendering engine (ogre), 2023. URL: https://www.ogre3d.org/. 1

[Uni23] UNITY TECHNOLOGIES: Unity, 2023. URL: https://unity.com/. 1

[WFS22] WRETBORN J., FLYNN S., STOMAKHIN A.: Guided bubbles and wet foam for realistic whitewater simulation. *ACM Trans. Graph. 41*, 4 (2022), 117:1–117:16. doi:10.1145/3528223.3530059. 1

[ZZC*22] ZHENG S., ZHOU Z., CHEN X., YAN D., ZHANG C., GENG Y., GU Y., XU K.: Luisarender: A high-performance rendering framework with layered and unified interfaces on stream architectures. *ACM Trans. Graph. 41*, 6 (nov 2022). doi:10.1145/3550454.3555463. 1