Abstract

In the field of visual computing, different technologies like multimedia or physics simulation merge with 3D computer graphics to create more realistic virtual worlds. During this process, 3D scene graph systems have evolved from being pure hierarchical containers of 3D geometry to full application specification frameworks. Through scripting and event routing, for example, the specification of entire (sub-)applications is possible inside a scene graph.

Nonetheless, today’s scene graph renderers are very monolithic: the single renderer has to be aware of every individual node type and its specific implementation. This makes exchange and recombination of different renderers impossible. In contrast to a single-renderer approach, we present SORA, a service-oriented architecture for rendering. SORA enables combining different specialized services, each of which operating only on parts of the same scene graph. We show an extensible and flexible framework for describing and implementing such services, as well as example service implementations interacting with each other.

1. Introduction

The rendering of highly realistic virtual worlds within a real-time graphics system requires more than just a high-performance rendering engine: 3D computer graphics have to be combined seamlessly with multimedia data (audio and video), simulation components for physics and artificial intelligence, and many more.

For describing such virtual worlds in great detail, the scene graph has proven to be the most flexible and powerful data structure to represent hierarchical data and its semantic interrelationships. In order to render a scene described in a scene graph, frameworks like OpenSceneGraph [4] or OpenSG [13] require to know the specifics of all contained nodes of the graph prior to rendering. Although those scene graph systems can be extended arbitrarily (in the form of new graph nodes adding functionality), their monolithic approach to rendering requires a full implementation of all nodes in order to render a frame. Even worse, an additional node that has an effect on its children might require changes in the implementation of the children as well, which leads to an even higher effort in maintenance the larger a node hierarchy gets. An additional problem of this approach is that using a ray tracing engine instead of rasterization would require to rewrite the implementation of almost every node type in the scene graph.

In this work, we propose a different approach based on a service-oriented architecture (SOA) [7]. Characteristic components of SOAs are services, which represent closed entities of data processing, and are loosely coupled to other services across well-defined commu-
communication channels. The second characteristic of a SOA is a registry concept to keep track of available services and perform a mapping from the problem domain to an executing chain of services.

Our approach named SORA – a service-oriented rendering architecture – uses multiple services to operate on the same scene graph. In our context, services are software components that are able to provide an interpretation of a well-defined subset of all nodes of a scene graph data structure. The scene graph itself returns to being just a semantic data structure, without forcing a particular interpretation on the developer. Thus, the SORA architecture is also independent of the underlying scene graph semantics and node types to be used. In our examples, we use X3D [3] as scene representation, because of its standard format and intrinsic extensibility, but our approach would work equally well with a COLLADA [2] scene graph, or anything similar.

The general idea of SORA is to add a layer of abstraction on top of different and partly incompatible visual computing technologies, such that the combination of those technologies works seamlessly. In doing so, each of the connections between services describes on a very high level what is some kind of data flow underneath. For example, pixel data from a service able to interpret multimedia video data might flow into a texture of a visual render service (located above it in the service hierarchy), all of which resulting in the video stream being displayed on a texture within the 3D scene. The important aspect of SORA is now, that those services can be arbitrarily exchanged as long as they have the needed interface in common. This means, you can, for instance, connect the very same multimedia service to a rasterization and a ray tracing service without having to change any implementations, as we will see later in the course of this paper.

In the following Sections, we will derive requirements for a service-oriented rendering architecture (Section 2) and show examples of related work in the field (Section 3). Section 4 then gives an overview of the SORA architecture by presenting essential concepts and components. In Section 5, we show how to realize typical real-world use cases before we conclude the paper in Section 6.

2. Requirements
This Section will derive requirements to be used in motivating the concepts and design of our service-oriented approach to rendering.

2.1 Exchangeability
For being able to have several services operate on the very same scene graph, the latter must not be tightly coupled to the implementation of any service operating on it. Even in the most simple example of a scene graph renderer only visualizing the geometry contained in a graph: if the rendering implementation is done inside the implementation of the graph nodes, rendering is strongly bound to the data representation and thus not exchangeable. So the first requirement (R1) for our architecture is the exchangeability of service components operating on the scene graph.

2.2 Extensibility
The second requirement of extensibility (R2) almost automatically follows from the first: as the scene graph is traversed by multiple services, a rendering system should allow adding new services as well as having an arbitrary number of services operating on the scene graph simultaneously. The actual visual renderer producing the final image – be that a ray tracer, rasterizer, Reyes renderer, or anything else – is just another service operating on the scene graph. The only requirement is its knowledge of the individual nodes to operate on. More details on the connection of services and graph structure will follow in Sections 4 and 5.

Any chosen approach should not only be able to add new services, but also to extend and specialize existing services by using standard methods from software engineering like inheritance and composition. Both vertical and horizontal extensibility are important aspects of the target framework.

2.3 Specialization
In heterogeneous environments, services should be able to utilize all the processing power they can get. This involves, for example, services being able to run on different hardware (like the CPU, GPU) and different operating systems. In making this possible, the goal should not be a single service that must support as many platforms as possible (although it very well might), but that the framework allows to interoperate between all kinds of specialized implementations.
This specialization, being the third requirement ($R3$) for our rendering system, is however not restricted to a single host: if we imagine a virtual world with highly realistic visualization and several simulation services (e.g., crowd, traffic or weather simulation) on top of that, we quickly have to cross the border of a single machine and have to consider multiple computers in a network for the tasks at hand. Here, the rendering system should be able to specialize in handling different networking technologies (e.g., Ethernet, or InfiniBand) seamlessly in a heterogeneous setup.

2.4 Interoperability

Services in the rendering architecture have to be linked together as the traversal of the scene graph proceeds. Interoperability (requirement $R4$) must be assured by agreeing on a well-defined interface through which the services are connected, and are enabled to exchange data during the process of rendering. Take for example the mentioned multimedia service that is to insert a video stream as texture into a different service responsible for the actual rendering: on a high level, both services have to know where they send and receive the pixel data, respectively, a task mapping down to method calls on a lower level.

Besides the pure interface connection, services must provide a higher-level description of what inputs and outputs they offer. A multitude of meta-data should be contained in this description (e.g., ‘image resolution’ of a multimedia service, or ‘level of detail’ for a mesh skinning service), as well as information about which hardware processing unit the respective service provides or receives its connection (e.g., CPU or GPU) to be able to optimize the connections. All these descriptions need to be formulated in a common service description language, which the system must be able to process.

2.5 Automation

Being able to have a multitude of services operating on the same scene graph also entails having a variety of options how to to map available services onto a given graph. Here, the rendering system should provide a full automation of the mapping process, which constitutes the fifth requirement ($R5$) for our framework.

Additionally, all the connections and configuration details of different services should be accessible and changeable for the application using the framework. Here, the implementation should follow the guidelines of white box design, which allows to provide the automatic configuration that “just works”, but also a manual reconfiguration to optimize the inner workings. By contrast, a black box design would lack the scalable transparency; that is, the ability to manually adapt the configuration of services and the resulting service graph, which is highly undesirable.

3. Related Work

Our requirement $R1$ has already been covered by numerous publications. In [6], the authors propose a generalized rendering system which decouples the scene graph from the interpretation done by the renderers. To achieve this, rendering functionality is encapsulated in engines. Each engine uses scene-node-specific techniques and handlers to interpret the nodes. This approach facilitates the introduction of a new rendering engine, because one must only wrap it into the given interfaces without requiring any changes to the scene graph. However, this is only true as long as there is no communication required between the separate components. Also, the scene graph traversal model with techniques and handlers implies some assumptions about the inner workings of the rendering engine. Some rendering algorithms like ray tracing may be difficult to fit into this concept.

Interaction between rendering modules is addressed in [9], where the rendering pipeline is assembled from a set of smaller components. Here, a separate data flow graph is introduced in addition to the scene graph. This graph defines the order of processing of the scene nodes, allowing the developer to visually compose the rendering pipeline from small building blocks. Even though this reduces the need for reimplementing some functionality when introducing new rendering engines, the system is designed strictly for OpenGL and will therefore not work with other rendering techniques.

In [1], the rendering pipeline is assembled from several components by connecting their inputs and outputs. A difference is made between 2D and 3D inputs and outputs, but apart from that inputs of the same dimensionality are assumed to be using the same data format overall, which makes it difficult to incorporate existing solutions into the framework. Run-time performance was not a critical concern to the designers. While it is possible to easily define the processing of the complete scene using this system, the fixed number
of inputs and outputs on a component make it difficult to use it for displaying multiple video textures.

The RTSG scene graph [15] fully separates the implementation of the rendering algorithm from the representation of the scene graph by letting each renderer module create its own representation of the scene as required. This allows highly efficient implementations of any rendering algorithm while keeping the scene loading and event handling separated. Because there is no interaction between rendering modules in RTSG, every renderer has to support every single scene node type.

Our need for loose coupling, automatic configuration and retrieval is largely covered by Service Oriented Architectures. According to the W3C, "A Service-Oriented Architecture is a set of components which can be published and discovered" [16]. However, previous work on service oriented architectures usually covers web services, which have performance issues that make them unsuitable for real-time rendering.

For web services, the WSDL standard defines a service description language which allows the service to announce the methods it supports and their syntax to any interested third parties. It consists of two sections: The abstract section and the concrete section. The abstract section contains a description of the functionality offered by the service, while the concrete section contains information on how to access this functionality, e.g. which method invocation protocols should be used. While the method invocation protocol is not predefined by WSDL, its format indirectly implies stateless communication via method calls and is therefore unacceptable for a rendering architecture.

4. Architecture of SORA

The general idea of our component-based architecture is depicted in Figure 1. Multiple services are responsible for processing different nodes of the scene graph. Each service provides a specialized functionality which is able to process a specific subset of nodes of the scene graph. Using a service-oriented approach entirely separated from scene graph operations thus automatically fulfills our requirement R1 (exchangeability).

A service in SORA can run on any system in the network, as all SORA components are build on top of the Network-Integrated Multimedia Middleware (NMM) [10], which provides network-transparent access to all components. Since a service in general should be able to manipulate the scene graph itself, either a distributed scene graph has to be used or the service itself is responsible to synchronize its internal representation of the scene graph with the scene graph of the application and other services. The usage of a generic distributed scene graph together with SORA is out of the scope of this paper but is a topic in our future work. In the following Sections we will show that even local application scenarios greatly benefit from our service-oriented approach.

The internal service architecture of SORA is based on a three layer approach, which is depicted in Figure 2, using NMM and the Object-Oriented Graphics Rendering Engine (OGRE) [12] as examples for multimedia and 3D rendering. In the following, we will describe all major components that make up a service and components designed specifically for data transmission.

4.1 Processing Layer

The processing layer is the lowest layer of our architecture and includes all components that have to be combined for performing concrete operations on the scene graph. In Figure 2, we show the combination of the visual renderer OGRE and the multimedia framework NMM. Both are chosen out of convenience, but could be replaced by other similar frameworks at will. Here, NMM is responsible for receiving images from arbitrary sources (e.g., a file, a camera, or TV board), then sending raw images to the rendering engine as textures.

OGRE provides external access to texture memory; at least two buffers should be used to prevent tearing artifacts when displaying a texture that is incompletely written. These buffers are used by NMM to store de-
Services in SORA are composed of three layers. The processing layer performs the actual rendering work, while the component layer decouples the processing from the scene graph. The description layer decouples services from each other and allows simple combination and exchange of services.

coded images in, before OGRE may flip buffers for presentation of the current frame, which thus requires no redundant copy operation. This is the general approach for the communication between arbitrary services: the receiving service provides memory areas that are requested by a sending service to store processed data in. This unified approach allows to use arbitrary communication protocols for transporting the buffer, for example simple pointer forwarding if both services run in the same address space, or TCP in the case of a network connection between the two services.

4.2 Component Layer

The second layer, called component layer, encapsulates all operations performed in the lowest layer within three types of components: the traversal component, the processing component, and the transport component. They abstract from the underlying technology and provide unified interfaces so that any technology can be used within services in a unified way.

A traversal component encapsulates all specific aspects of a used scene graph and is responsible for the communication between the processing component and the scene graph. This may include, for instance, traversing the nodes of the scene graph and mapping them to a specific representation for the processing component, or synchronizing the internal representation with the scene graph. Since this component is tightly coupled to the scene graph itself, we apply the Strategy design pattern [8] within a service, such that only this component has to be reimplemented for a different scene graph, while the rest of the service can be reused.

The central component of a service is the processing component, which is responsible for the actual processing based on the scene graph structure. The processing component is the one component defining the characteristic nature of a service, such as a service for visual rendering, or mesh-based physics. Within SORA, each processing component has a unified interface for receiving data from or sending data to other services. A processing component can provide interfaces for sending data, receiving data, both, or neither.

A transport component is responsible for connecting two services by providing a link between their processing components. To support arbitrary transmission technologies, our transport component is based on unified interfaces. The processing component at the receiving end can provide a set of buffers to the transport component. It is then then used by the transport component to either copy data to the next service or directly store it at the location provided by the successor.

As the processing component at the sending end requests buffers from the transport component to store processed data, this allows the local optimization that the sender directly copies data into an internal buffer of the receiver, in case both processing components are located on the same host. The mechanism still works if these components are distributed in the network. In a distributed setup, the transport component provides a set of buffers to the sender but still uses the given buffer of the receiver to directly store the buffer read from the network. As soon as the receiver no longer needs any buffer, it is released for further use.

Through strict separation of functionality for traversal, processing and transport we define clear interfaces for developers to extend entities and thus master requirement \( R2 \) (extensibility). In addition, the extensibility allows us to provide specialized implementations for specific platforms or processing units within services. At the same time, specialized transport components enable efficient communication between any specialized processing components while still achieving a maximum of reusability, all of which fulfilling \( R3 \) (specialization).

4.3 Description Layer

The uppermost layer of SORA is the description layer which includes all components representing informa-
tion that is required for traversing the scene graph as well as connecting, and configuring used services. The functionality of this layer is required to ensure interoperability (R4).

We use control interfaces to allow application-specific access to and control of a service; for example, to configure the output image size of a renderer. Those interfaces are used only for those features independent of other services; inter-service functionality is negotiated between two services, and included in a service description of a service; for example the resolution of a texture in-between multimedia and renderer service.

Within SORA we distinguish between two kinds of those service descriptions: service offers and service requests. As the names suggest, the former describe what service connections a service has to offer, and the latter describe the need for a certain input in order to work correctly. Each service contains at least one service offer, and an arbitrary number of service requests. We will give specifics of when and where these descriptions are used in Section 5. Service descriptions always consist of the following information:

1. Root nodes: a list of nodes the service can start a traversal from (service offer), or the node a service cannot handle anymore (service request).
2. Data format: This optional information describes the date format for the connection as key-value pairs; for example, image resolution values if a service offers sending imagery to other services.
3. Location information: This optional information describes onto which hosts the result of the processing can be delivered to, in case produced data is sent to a different service. The location information contains all required connection information for realizing and optimizing data transfer, such as IP address, network port, or processing core (CPU, GPU).

During setup of a service, all its service offers are created using the information provided by the traversal and processing components. Service requests, on the other hand, are created by the traversal component as soon as it finds a node it cannot interpret, as will be detailed on in the following Section.

5. Implementation

For implementing all components of SORA we again build on NMM; this time not at all for its multimedia functionality, but for its powerful communication framework.

5.1 Automatic Service Setup

Even though a manual configuration of the service graph is possible in SORA, we need an automatic mechanism to find and setup a service mapping to handle the increasing possibilities of processing any single scene graph. In addition, the mechanism of mapping a graph to services should consider newly added services and thus overall fulfill requirement R5 (automation).

As a first component to manage any number of services, we need the concept of a service registry. In contrast to other approaches like UDDI [11], we do not use one central registry that is responsible for managing the available services, but have each host that provides SORA services run its own service registry. Using a decentralized approach for our service registries allows both using a single registry as well as realizing a peer-to-peer approach, which then provides better scalability and reliability in large environments. Each service registry stores all service offers of available services to be able to answer queries for their capabilities. An application either explicitly specifies participating hosts or uses a broadcast approach to find registries available in the network. As soon as a service description of type service request (cf. Section 4.3) is received at a service registry, it tries to find suitable matches:

In a first step, we compare the root node of the service request with the service offers existing in the current registry for matchmaking. As mentioned before, the first node of a scene graph a requesting service encounters, but does not understand anymore, becomes root node for the respective service request. In addition to the root node, an optionally specified data format is also compared and has to be supported for a full match.

In a second step, our matchmaker checks if a transport component exists that is able to connect the requesting service with any service offer provided by the services found during the first step. Available transport components are also provided by each service registry. Inside our transport components we use NMM transport strategies, which allow data transport between (a) remote hosts using various protocols (e.g., UDP, TCP, RTP), and (b) different processing architectures (e.g., GPU, CPU) using various communication technologies (e.g., pointer forwarding, DMA transfer, shared memory). The binding mechanism of NMM, which is also
used in SORA, then automatically finds the best combination of transport strategies to enable the data transport between any two hosts, architectures and technologies [14]. As a straightforward solution, our implemented matchmaker uses the first valid matching combination of services, even though more sophisticated approaches can be integrated at will because of our usage of the Strategy design pattern [8].

5.2 Scene Graph Traversal

To initially start the traversal of a given scene graph, any service that “understands” the root node of the scene graph may start the traversal. This is usually a visual rendering service, but could as well be a generic format converter service, for instance. As soon as the traversal component of a found service arrives at a graph node it cannot interpret, it creates a service request for that particular unknown node type. The SORA registry then tries to find a suitable counterpart component that is able to interpret this very node type as a root node and connects the services. Step by step, the scene graph is mapped to a corresponding service graph covering the interpretation of all of its nodes and describing the flow of information in-between self-contained services (cf. Figure 1).

After the initial mapping is done successfully, the actual processing of the scene graph takes place in the processing components of each service, for which we define a two-stage execution model: During the update stage, first the scene graph is configured to a new frame time by sending this time to all services. Then all operations of services are executed that manipulate the scene graph, e.g., events and adding or removing nodes of the scene graph. Since the processing order of nodes of a scene graph is tightly coupled to the scene graph semantics itself, we cannot provide a generic solution that suits all needs. Instead we apply the Strategy pattern that allows to configure an update strategy specific to the scene graph in use. By default, we use a straightforward implementation that invokes services in the order of their instantiation.

In the second execution stage, the processing stage, those operations are executed that only need read access to the scene graph; that is, all graph-altering changes have been made in the previous stage already. During this stage the execution order is defined by the data flow of the service graph starting at the leaves and going up to the root service. This intrinsically allows parallelization of service execution within the same depth of the service graph, as no changes can ever happen during this stage.

5.3 Use Cases

For demonstrating the capabilities of SORA, we implemented a set of services for realizing different use cases. First, we integrated two rendering engines into our system: OGRE and OpenRT [5], representing a rasterization and a ray tracing approach, respectively. Both engines are able to load images as textures but are not able to use moving pictures even if the texture format of the images is supported in general. Both could be extended to support moving pictures, but any extension could not be reused for a different engine.

Using SORA, we then integrated the multimedia functionality of NMM into a separate service, such that its results can be used within both mentioned rendering engines. The results are shown in figure 5.3 and figure 5.3 respectively. Due to the capabilities of NMM we are not limited to the playback of local movie files. Moreover, arbitrary local and remote video sources (e.g., cameras, TV cards, or streaming videos) can now be played back inside the 3D scene without changing a single line in the implementation of our rendering engines. This nicely shows the benefit of extensibility and exchangeability within the design of SORA.

As a further extension, we want to apply video filters which run on the GPU using the CUDA, all of which provided by NMM. Again we want to use the results...
in rendering services afterwards, and want to apply as few copy operations as possible. This is where our specialized transport components come into play: For connecting this new CUDA filter service to the OGRE service, we employ a CUDA-to-OpenGL transport component. OGRE exposes its OpenGL texture buffers to this transport component, which in turn registers them as CUDA buffers. This way, the CUDA filter service can directly write the image into the OpenGL texture in OGRE without a single unnecessary copy operation. And we can use the same CUDA filter service in combination with the OpenRT ray tracing service running on the CPU: In this case a CUDA-to-CPU transport component is used, which copies the CUDA buffer to the respective texture buffer of OpenRT in main memory. Being able to suit both services in such a heterogeneous memory environment not only shows the benefit of but also the need for specialization to specific technologies.

6. Conclusions and Future Work

We have presented SORA, a service-oriented rendering architecture that enables the flexible combination of several services on the same scene graph. Services in that respect may include classical visual rendering (e.g., by ray tracing), but also simulation services like traffic or game AI simulation. Services process the scene graph as far as they understand the respective nodes, and export unknown nodes within so-called service requests. Other services registered with SORA can then fill in the blank by offering the requested service and form a connection to the first service. The overall result is a service graph that maps an input scene graph entirely to its interpretation within SORA.

In general, SORA does not affect the real-time capabilities of a resulting system due to an optimized communication between the processing components. However, we only evaluated SORA in a small network environment with a maximum of five services such that the initial setup time for finding and assembling a suitable service graph is not relevant for the user experience. In the future, we need to test SORA within larger network setups, though, involving more hosts and more services to evaluate scaling and performance; for example in the realm of massive multiplayer online games.

Future work on SORA also includes the implementation of even more services and interfaces to examine the suitability of our approach for even more complex challenges like crowd simulation and graph-altering mesh operations. Additionally, we try to break up existing services into smaller units that, for example, only handle parts of the rasterization pipeline, but do that more efficiently on specific hardware.

References


