Integrating GPGPU Functionality into Scene Graphs

Jens Orthmann, Maik Keller, Andreas Kolb

Computer Graphics Group, Institute for Vision and Graphics, University of Siegen
Email: {Jens.Orthmann,Maik.Keller,Andreas.Kolb}@uni-siegen.de

Abstract

The concept of scene graphs is widely used in computer graphics to structure graphics-related entities, e.g. geometry, visual attributes as well as abstract data related to certain application requirements like object identifiers or manufacturing details.
This paper presents a new method to incorporate *General Purpose Graphics Programming Unit (GPGPU)*-functionality into scene graph APIs. We define specific scene graph nodes in order to realize a flexible integration of GPU functionality at various levels of granularity without violating the programming paradigm inherent to scene graphs. We focus on current and upcoming *compute APIs* like CUDA, which are designed for GPGPU purposes.
We further present the `osgCompute` framework that implements our concept and is based on the OpenSceneGraph API. CUDA is integrated into `osgCompute` via `osgCuda`. Our method is flexible in the sense that other compute APIs could be used instead. The advantages of our concept and of `osgCuda` are demonstrated by presenting examples with different processing requirements.

1 Introduction

Scene graphs have constituted the platform for many applications in the last two decades. Their ability to employ standard graphics functionality as well as their power to encapsulate complex graphics operations is the reason why scene graphs became an industry standard. Over the years, different improvements like multi-threading have been integrated into scene graph-APIs, making architectures more and more complex. Still, scene graphs provide a good abstraction for developers who are no experts with regard to all technological details. Many hybrid applications like computer games or simulations have incorporated scene graphs in order to split up problems to smaller hierarchical structures.

Apart from scene graphs, General-Purpose Parallel Computing on GPUs has attracted great attention in recent years. The rapid increase of GPUs performance and the parallel architecture designed for streaming computations have convinced many developers to port their algorithms for graphics hardware even though the standard GPU programming paradigm is rather non-intuitive and closely constrained by the underlying graphics pipeline. Currently, so-called *compute APIs* like NVIDIA's Compute Unified Device Architecture (CUDA) [10] have been developed, making the implementation of data parallel algorithms on GPUs much more comfortable.

Both aspects give rise to design a new type of interface for hybrid applications which combines the power of the hierarchical structure of scene graphs with the flexibility offered by new compute APIs like CUDA, OpenCL or DirectX Compute. Incorporating compute APIs into scene graph-APIs has a high potential to leverage GPGPU functionality to new areas, e.g. to industrial applications. A generic integration concept should simplify the programming of parallel algorithms for hierarchically organized data structures. Furthermore, it should follow the intended extension mechanisms of current scene graph APIs and preserve the scene graph's model of abstraction and usability. Our generic concept makes the following contributions:

- We introduce a generic framework concept to integrate data parallel algorithms into current scene graphs. The framework concept minimizes restrictions on both the computation and the rendering API. The approach is integrated seamlessly into the scene graph programming paradigms and allows various levels of granularity regarding the utilization of the GPU.
- We compare our implementation concept to other, already existing, approaches and derive application patterns for our framework concept.

- We present an implementation of our concept by combining the multi-threaded OpenSceneGraph-API with NVidia's CUDA-API. Due to the flexible implementation future compute APIs like OpenCL could easily replace CUDA.
- To show the feasibility of the design we have implemented three example applications which capture some of the most common use cases including a simple ray tracer, a particle simulation and an image processing algorithm.

The structure of this paper is as follows: In Sec. 2, an overview of related concepts and work is presented, while Sec. 3 proposes our sub-traversal approach as the basic concept on an abstract level. In Sec. 4, we describe the implementation of our concept combining OpenSceneGraph and CUDA. Results and sample applications based on our framework are presented in Sec. 5. Finally, in Sec. 6 we conclude with future work.

2 Related Work

This section states the essential concepts of scene graphs (Sec. 2.1) and discusses the existing integrations of compute APIs into scene graphs based on callbacks [8] or scene graph engines [6] (Sec. 2.2 and 2.3). However, both concepts violate the scene graph paradigms to a considerable extent.

2.1 Scene Graphs and Compute APIs

The scene graph concept was first introduced by Strauss et al. [14]. Widely used scene graph APIs are Performer [13], OpenInventor [16], OpenSG [12], NVSG [11], and OpenSceneGraph [3]. Scene graphs generally organize rendering objects in a hierarchical way, usually in a directed acyclic graph (DAG), by using a composition of nodes. Scene graphs are evaluated by a traversal sequence. During these traversals each node is visited and either an action is performed on the objects or each object is inspected. Doellner et al. [4] propose a generic rendering system which decouples the scene graphs structure from the underlying rendering API. As in most scene graph APIs, they use a pre-order traversal method, which they denote as *evaluation traversal*. We follow this concept in our design by employing additional design patterns from Gamma et al.[5].

It has to be noted that multi-threading concepts strongly differ, making an abstraction of multi-threaded scene graph APIs de facto impossible.

CUDA [10] and OpenCL [7] are the most prevalent compute APIs; new APIs like Microsoft's DirectX compute [1] will be launched in near future. NVSG [11] is a new scene graph API from NVIDIA, which is a closed source project. NVSG is built on top of shader-based graphics APIs. Current developments are also touching upon the usage of compute APIs like CUDA [11] in order to improve ray tracing techniques. However, the switch to NVSG requires the complete replacement of the scene graph API for existing applications.

2.2 Callback Approach

One way to integrate a compute API, as presented by Mercury Systems [8] in the context of their VolumeViz project, is to encapsulate data parallel algorithms within callbacks which are attached to a specialized type of leaf node. Callbacks are designed to redirect the execution to some user-defined function during traversal of the graph. They are usually self-contained structures which should gain access only to their attached nodes. In their simplest form they interact with one dedicated node of the scene graph only and they are completely invisible to the other parts of the graph. In the graph structure presented in Fig. 1 (A) two CUDA-based computation functions generate two procedural texture resources. The resulting scene is shown in Fig. 8. Even this simple graph structure introduces fixed references between the callback functions and their resources. The computation resources located in the graph must be directly addressed by the application. This requires additional application dependent logics in order to resolve dependencies. Computations cannot be arranged hierachically or cannot share data easily without intransparent software indirections which connect parts of a scene in order to exchange resources.

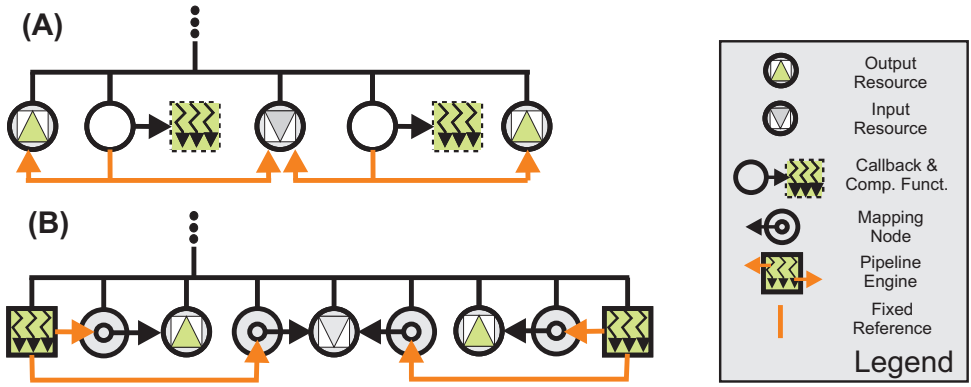

Figure 1: The graph structure introduced by the callback approach (A) and the engine approach (B) representing the scene shown in Fig. 8. Both approaches require four fixed references to resolve their resource dependencies.

2.3 Engine Approach

Another option for the integration of CUDA has been introduced by Giden et al. [6]. They have implemented computations as OpenInventor engines. Engines are designed to constrain one part of the scene to another and provide inputs and outputs of a fixed type [16]. Internally, the so-called pipeline engines are set up by one or more modules which can be arranged in a complex recursive pipeline structure. Each of these nodes implements a single and isolated data parallel algorithm.
Giden et.al. [6] introduced additional node classes to map data between the rendering and the computation context, which can be connected to the engine. In their case, the interface comprises the engine and the mapping nodes. In addition, they have to introduce a new kind of context which executes the computations as they do not apply any traversal mechanism which usually defines a context. Different engines can share the same computation context. The parts of the scene graph affected by the engine cannot be identified by observing the graphs structure only. Even worse, the engine nodes can be connected to arbitrary parts of the scene graph and thus do not reflect any hierachies. The graph of the scene related to Fig. 8 is outlined in Fig. 1 (B).

3 Subtraversal Approach

The two design variants presented in Sec. 2 offer well defined interfaces for specialized application domains. However, they both do not fit into the abstract object-oriented design patterns of scene graphs. The major motivation for our design is to overcome their limitations. Thus, we introduce *computation nodes* which traverse their own subgraph. The idea is similar to the concept of camera nodes: Camera nodes are usually connected to a render target. A camera defines the parameters for rendering and provides the result of the rendering in form of render targets to its ancestors. For each geometry object found in their subgraph the rendering pipeline is triggered.
In contrast to camera nodes, computation nodes gather information from their sub-graph during traversal, execute data parallel computations and offer the result to their parent nodes. Fig. 2 (C) shows the graph structure to build up the simple example already realized for the callback and engine approach. The input and output resources are collected from the subgraph. This approach maintains most of the programming paradigms of scene graphs. To be precise, we aim at the following three designs goals:

- Computation nodes minimize the number of fixed references to scene nodes because all necessary resources for the computation are located in their subgraph. In contrast to the

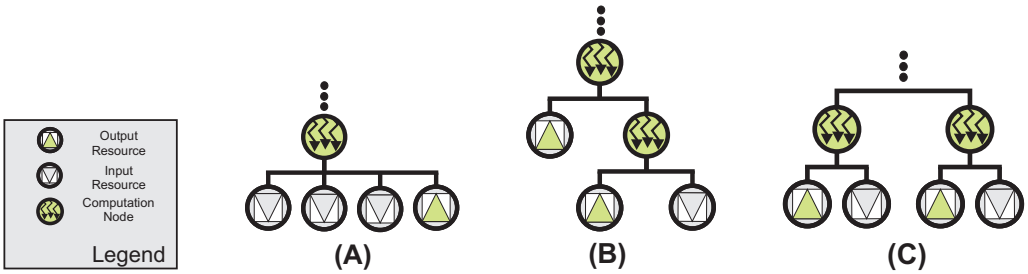

Figure 2: (A): A single computation node with an arbitrary sub-graph. (B): Multiple computation nodes in a pipeline structure. (C): Multiple disjoint computation nodes. The graphs do not show the structures required for rendering.

callback or engine approach, resources and computations are detached and modular objects which can be easily exchanged. This provides a seamless abstraction for developers.

- The common scene graph traversal mechanism is utilized in order to select resources for the computation. Resources may be directly applied to the respective computation node as well. The subgraph of a computation node can be replaced or rescaled dynamically. This is an advantage over the previous approaches, since they need to introduce new logics in order to make a dynamical exchange of resources possible.

- Hierarchies between computation nodes are described by the graph's structure and do not introduce any invisible dependencies. This also allows developers to interlace computations with camera nodes. However, recursive structures are not directly supported as scene graphs are acyclic by nature.

The subtraversal approach requires *computation contexts*, which manage the resources for one or more computation nodes even in multi-threaded environments. One or more computation contexts manage computation resources, e.g. input and output buffers or modules of one computation graph.
Since computation nodes are designed as a specialized group node, they demand their own traversal schemes to keep resources up-to-date.
One can utilize computation nodes to construct several structures in the scene graph as discussed in the following Section.

3.1 Utilization

Computation nodes can be inserted at any place within the scene graph. Regarding the requirements for the context handling, three application types can be distinguished, which are examined in the following paragraphs. Corresponding examples are presented in Sec. 5.

3.1.1 Single Computation Context

In this case only one computation node is present. This is the simplest possible scene graph structure containing computation nodes. A single computation context is necessary which does not need to be shared between different computation nodes. The traversal only has to collect and filter required resources from the subgraph. Note that the subgraph of the computation node can be of arbitrary structure containing any standard scene graph nodes (see Fig. 2 (A)). This heterogeneous structure requires mapping of memory between the computation and the rendering context. For example a ray tracer may use this kind of structure which does not require the introduction of a hierarchy. However, more complex approaches may utilize ray tracing in distinct parts of the scene graph. This would require one or more additional contexts.

3.1.2 Shared Computation Context

If a scene graph has a hierarchy of two or more computation nodes, the computation context is shared among these nodes, which requires an optimized

mapping of resources. Additionally, hierarchical computation nodes and camera nodes affect the traversal mechanism. Nodes may be *linked*, i.e. input and output resources get connected, if the output resource of successive computation nodes is an input of the current computation node. The topmost computation node starts an update traversal and receives resources from its successor nodes (see Sec. 3.3.1 for details on resource handling). Similar to the single context case, resources from rendering and computation contexts need to be mapped. This typically introduces a computational overhead. Afterwards the structure is executed in a bottom-up approach, as explained in Sec. 3.3.2. Nodes are organized hierarchically in a directed acyclic structure (see Fig. 2 (B)), thus more complex structures like recursions need to be realized within a single computation node (see also Giden et al. [6]).

3.1.3 Multiple Computation Contexts

Placing several computation nodes in a scene graph in a non-hierarchical manner, i.e. in different disjoint subtrees, introduces a new level of complexity since several computation contexts exist. Here, resources may have to be attached to one or more computation contexts, providing resources to different computation nodes. Similar to multi-threading, this might require multiple and synchronized instances of shared resources, i.e. a specific *resource mapping*. Fig. 2 (C) gives an example of a graph layout employing multiple computation contexts.

3.2 Computation Contexts

In compute APIs like CUDA, developers have to deal only with two things, i.e. the kernels or programs which implement the algorithms written in the language of the respective compute API and the required memory. All resources and actions are assigned to a *computation context*. Consequently, a computation context must be active in order to execute data parallel code accessing context dependent memory [7, 9, 10]. In scene graphs, on the other hand, a rendering context [4] or rendering state [2, 13, 14] provides resources which are currently active according to the rendering traversal state. Both context types handle resources potentially required by specific computation nodes.

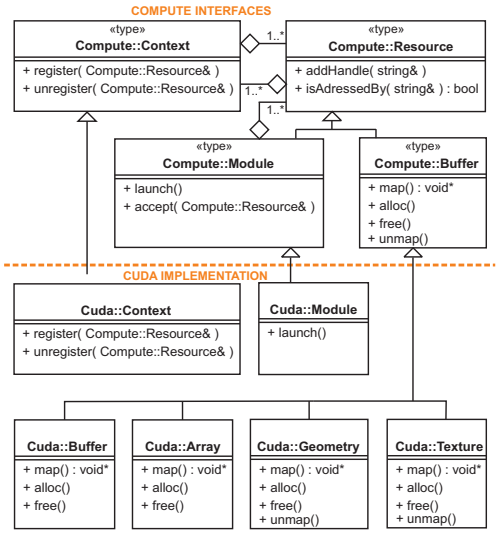

Figure 3: The class diagram showing the major components for the abstract `Compute` interface and their `Cuda` implementation, focusing on computation contexts and resource handling.

In our concept, the *computation context*, (see Fig. 3 class `Compute::Context`) manages all resources required by any data parallel algorithm. We utilize a generic common interface class (class `Compute::Resource`) to handle modules and buffers (data streams or arrays). This interface allows developers to address computation resources within the graph on an abstract level. Such an interface based class structure is frequently used by current scene graph APIs in order to abstract from rendering API specific implementations [3]. An alternative concept would be a 2D handler table to make the implementation more generic and independent from the respective functionality and resources [4].

The main advantage of this interface is the transparent mapping of resources, especially between different contexts. The topmost computation node will create a computation context object which is shared via traversal with further computation nodes that are possibly located in its subtree.

3.2.1 Modules

A module executes a kernel function and is directly attached to a computation node. This allows developers to vary the algorithm of a computation node in order to provide different behaviors for the same subgraph. Thus, the communication between the computation node and the respective module is defined by a so-called strategy pattern which is described in Gamma et al. [5].
Modules accept resources of interest which are collected by the gathering traversal. These resources are passed through the computation node to the module. Prior to program execution each module maps its buffers each of which returns a pointer to its context dependent memory. A module has to specify a regular array of threads in order to execute a program in parallel on the GPU [1, 7, 10]. However, the organization of the array is crucial for achieving high performance. It depends on the processed resources and can therefore not be generalized. For each module the layout of the so-called thread-blocks must be explicitly specified in order to expose enough fine-grained parallelism to exploit the massively multi-threaded GPU hardware.

3.2.2 Buffers

Buffers abstract memory which can be allocated in different contexts. Memory is allocated lazily when the first mapping of a buffer is requested. At this point a buffer is registered in the active computation context. The mapping function then returns a handle to the related memory. Whenever a computation context is removed, all registered buffers are notified to release their memory. Finally, the memory is freed when the reference count reaches zero.
Buffers which also exist in the rendering context, so-called interoperability buffers, need to map the internal rendering objects to the respective compute context before launching a module. Usually, this mapping causes some computational overhead [10, 1]. After a module has been executed the `unmap function` of each buffer is called which maps the memory back to the rendering context in order to be available for rendering purposes. Buffers which only exist in the compute context do not implement this function.
In general, two types of rendering resources are of interest: textures and geometry objects (providing vertex buffers). The scene graph API must grant developers access to the underlying graphics API of those objects in order to make mapping for such buffers possible. In our design, classes like `Cuda::Geometry` and `Cuda::Texture` act as mediators [5] between the different contexts: A buffer as well as other resources can be attached to several contexts at the same time and so need to replicate their data as long as the contexts do not share the same memory space. I.e. multiple CPU threads evaluate independent computation nodes in parallel. Systems which utilize only a single memory space for computations do not require such replication mechanisms. In this case the attachment to contexts is unnecessary and each buffer can handle its memory allocation and deallocation internally.
The API dependent implementation of the allocation and mapping functionalities are defined in the classes which implement the interface, e.g. in `Cuda::Texture`, and thus is completely transparent for modules.

3.3 Graph Evaluation

The evaluation of the scene graph requires two traversals. The gathering process which collects the required resources whenever the subgraph has changed and the execution traversal which launches the computation nodes in bottom-up order, since input resources may be output resources of a successor node.
Both traversals are implemented in a pre-order-like scheme [12], i.e. first nodes, starting from the root of the subgraph, are processed before their successors are visited. In the context of the generic rendering systems this is called *evaluation traversal* (see Doellner et.al. [4]) which is the most common traversal type found in scene graphs and is available in OpenSceneGraph [3], OpenInventor [16], OpenSG [12], Performer [13] and NVSG [11].

3.3.1 The Gathering Traversal

Since computation nodes have flexible input and output resources which solely depend on the implemented algorithm, the standard fixed-function state

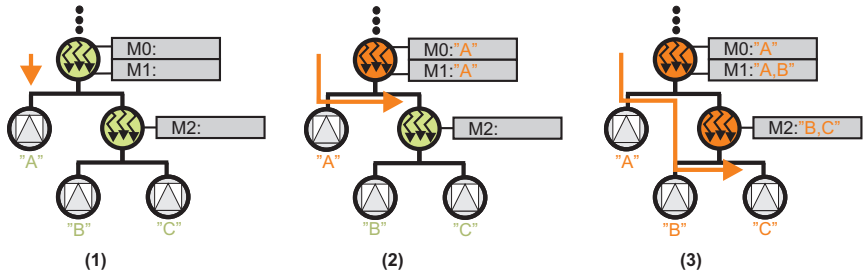

Figure 4: The gathering traversal. The topmost computation node creates a computation context and starts the traversal (1). The buffer 'A' located in the subgraph is forwarded to the modules M0 and M1 (2). The traversal hands over the context to the child computation which requests its resources 'B, C' (3). The buffer labeled with 'B' is accepted by modules of both computations.

approach used for rendering scene graphs is insufficient. The gathering traversal identifies the required resources in the subgraph of each computation node and establishes a link between hierarchically organized computations. This is necessary since data parallel computations on the GPU require input and output streams in a very flexible way, which means that the required resources cannot be easily identified.

Our approach uses name-identifiers to locate respective resources in the subtree. Collected resources are inspected by the modules and bound to the respective computation node. Alternatively, a module may request all resources of a specific type, e.g. all geometry resources. Each computation node located in the subtree of a topmost computation node will share this context with the topmost node. Thus, if the gathering traversal visits a child computation, the context is handed over. Fig. 4 explains the gathering process for the pipeline structure which has already been introduced.

Note that moving computation nodes possibly changes the node's computation context. This might force an expensive reallocation of all memory if memory spaces differ among computation contexts. After the traversal the resources are ready for being utilized by the parallel programs during the execution traversal.

3.3.2 The Execution Traversal

The execution of the computation nodes is straight forward. In case of a single computation node, each attached module maps all the relevant buffers, executes the parallel programs and unmaps the buffers. Calling the mapping functions in this order is absolutely necessary since interoperability buffers might be rendered after or before the computation.

For a shared computation context this procedure implies several context switches. Resource data has to be mapped more often between the computation context and the rendering context. Fig. 5(1) shows

```
C.map();    B.map();
M2.launch();
C.unmap(); B.unmap();

A.map();
M0.launch();
A.unmap();

A.map();    B.map();
M1.launch();
A.unmap(); B.unmap();

C.render();

A.render();
```

(1)

```
C.map(); B.map();
M2.launch();

A.map();
M0.launch();

A.map(); B.map();
M1.launch();

C.unmap();
C.render();

A.unmap();
A.render();
```

(2)

Figure 5: Pseudo code showing the function calls during the computation of the graph outlined in Fig. 4. 'A' and 'C' are interoperability buffers whereas 'B' is accessed only within the computation context. If the interoperability buffers are mapped back to the render context directly after the modules are launched (1) then more expensive context switches (orange) are necessary as if the buffers are mapped back during the rendering operation (2).

the involved function calls for our pipeline example. The modules of a computation are launched after the respective subgraph has been traversed, because the outputs of successor nodes might be used during execution. This also is the case for interlaced rendering operations. If it is possible to call the unmap function directly before the rendering function of an interoperability buffer the number of mappings is reduced as shown in Fig. 5 (2). Note that this does not influence the rendering action for any interlaced non-computation node. If required, intermediate results are still available through interoperability buffers. Similar to the approach of Giden et al, one can introduce more complex and recursive structures into a single computation node.

4 Implementation Aspects

By implementing the subtraversal approach within OpenSceneGraph we have to deal with the following API specific aspects: First, computation resources providing interoperability between the computation API and the rendering API have to fit seamlessly into the rendering state concept of the respective scene graph. Second, resources need to support the provided threading model.

4.1 Interoperability

OpenSceneGraph provides full access to the internal OpenGL structures of texture and geometry resources. For each of these rendering resources we designed a resource class, e.g. `osgCuda::Texture`, which on the one hand inherits functionality from the respective OpenSceneGraph class, e.g. `osg::Texture`, and on the other hand implements the interface class, e.g. `Compute::Buffer` (see Fig. 3). This allows us to address the related memory in both contexts by implementing CUDA related mapping functions within a single specialized class. However, doing so introduces multiple inheritance into our framework design which is a commonly used design principle in OpenSceneGraph.
Another CUDA related aspect is the different handling of the interoperability functions concerning texture objects. Texture objects currently cause some additional synchronization workload as texture memory internally requires to be copied into or from an OpenGL pixel buffer object in order to make mapping to CUDA possible [10]. Since mapping between contexts always comes with some performance penalty, especially for large textures, we only map resources on demand by employing the optimized mapping strategy shown in Fig. 5(2).

4.2 Thread Synchronization

OpenSceneGraph utilizes multiple threads to render a single representation of a scene to multiple displays e.g. for virtual reality applications. Burns et al. have implemented a fixed sequence of subsequent traversals (update-cull-render) of which each might run in its own thread. Burns et al. define which visitor traverses the graph at which point in time and so are able to clearly synchronize traversals. However, it is still possible that two threads are rendering the same scene for multiple displays as it is not an unusual case for virtual reality applications.
With multi-threading the problem of synchronization between multiple reader-writer threads is introduced. One solution is to replicate data of scene objects for each active thread in order to avoid synchronization calls [15]. Because we perform the computations within the rendering traversal, each computation resource must be aware of multiple threads accessing it during rendering. OpenSceneGraph avoids conflicts in such situations by replicating the private GPU memory of each rendering resource and constraining each rendering thread to render only within a single rendering context. Resources have to manage one copy for each active compute context. Additionally we constrain one CPU thread to operate only on a single compute context in order to be thread-safe in the sense of OpenSceneGraph. The selection of the respective data is encapsulated within the mapping function and so is transparent for modules.

5 Results

We have presented a concept and implementation for the integration of modern compute APIs into existing scene graph APIs. Our approach ensures a high flexibility, i.e. there are no severe restrictions

inserted into any of the APIs. Furthermore, our approach establishes methodologies to integrate computation nodes into scene graphs based on extended scene graph concepts, i.e. an additional identification of computational resources is introduced. Our method makes different levels of granularity possible. The concept does not enforce restrictions on how computational tasks are integrated in the scene graphs. A general and transparent interface model is designed for resource handling with scene graphs. We have implemented three simple example applications in order to show the feasibility of our framework. Each example utilizes one of the tree structures discussed in Sec. 3.1.

5.1 Ray Tracing

Fig. 6 shows a ray tracing example which utilizes the atomic graph structure of Fig. 2 (A). Three sphere geometries of which each has a different level of detail are located in the subtree of the computation node launching a ray tracing program. The left spheres are accepted as inputs for the computation. Whereas the last sphere node is not labeled and is omitted by the ray tracing module but is rendered using the raster-based graphics pipeline. The ray traced results are written into a `Cuda::Texture` resource which afterwards is blended with the rendered image containing the rightmost sphere.

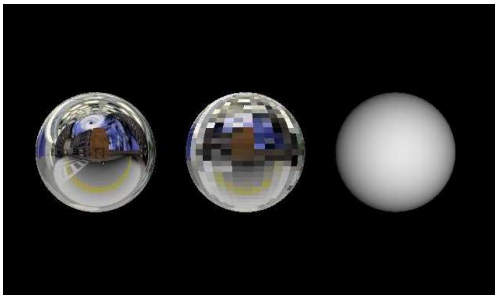

Figure 6: Ray Tracing Example. The scene is rendered in two steps. The two spheres on the left are ray traced while the last one is rendered using the rendering pipeline. The scene graph is organized similar to Fig. 2 (A)

5.2 Particle Simulation

The graph used to render the particle simulation example (see Fig. 7) is arranged by two computation nodes: the topmost node computes the particle movement and emits particles at specific seed points. The child computation recomputes these seed points whenever the box is moved. Seed points (`Cuda::Buffer`) as well as the box geometry (`Cuda::Geometry`) are located in the sub-graph of the child computation whereas the particle geometry (`Cuda::Geometry`) is attached to its parent computation. In each frame the particles are moved towards a specific direction and are reseeded when they leave the box.

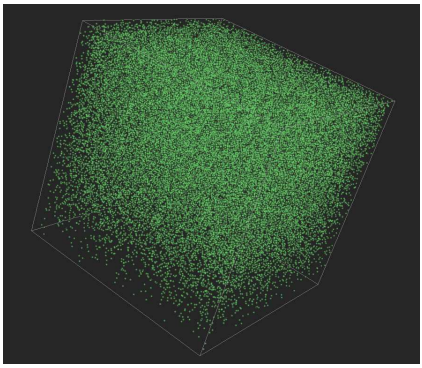

Figure 7: The particle simulation example. The scene graph is composed of two hierachically organized computation nodes. The computation of the particle movement and the update of the seed points are consecutive operations described by the graph shown in Fig. 2 (B).

5.3 Image Processing

The graph of Fig. 2 (C) employs two independent computation nodes to realize the scene shown in Fig. 8. Each node performs a different operation on the same input resource of type `Cuda::Texture` which is added to both subgraphs. The result of each CUDA program is written into an output texture (`Cuda::Texture`) of which each is also located in the respective sub-graph. Both nodes create their own computation context which share the same memory space.

Figure 8: Two textures, each of which is generated by a CUDA program and projected onto a quad.

6 Future Work

With regard to future work we would like to improve the resource identification which currently induces a rather hard-link between computation nodes and resources. Furthermore, we will extend our implementation to support distributed rendering and computations across several rendering nodes in a cluster.

Acknowledgments

This work is partially funded by the Siegener Graduate School “Development of Integral Heterosensor Architectures for the n-Dimensional (Bio)chemical Analysis”.

References

[1] Chas Boyd. Direct3D 11 compute shader – more generality for advanced techniques. Presented at Gamefest, Microsoft Game Technology Conference, 2008.

[2] Grigore Burdea. Introduction to VR technology. *Proc. IEEE Virtual Reality Conference (Tutorial)*, page 265, 2004.

[3] Don Burns and Robert Osfield. Open Scene Graph: Introduction (part A), examples and applications (part B). *Proc. IEEE Virtual Reality Conference*, page 265, 2004.

[4] J. Döllner and K. Hinrichs. A generic rendering system. *IEEE Transactions on Visualization and Computer Graphics*, 8(2):99–118, 2002.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[6] V. Giden, T. Moeller, P. Ljung, and G. Paladini. Scene graph-based construction of CUDA kernel pipelines for XIP. In *Proc. High-Performance Medical Image Computing and Computer Aided Intervention*, 2008.

[7] Khronos. The OpenCL specification. December 2008.

[8] Mercury Computer Systems. Mercury computer systems, visualization sciences group, and NVIDIA to provide combined high-performance computing and visualization to make oil and gas exploration more efficient. Mercury Computer Systems Press Release, 2008.

[9] Microsoft Corp. Introduction to the Direct3D 11 graphics pipeline. 2008.

[10] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[11] NVIDIA. NVSG – nvidia scene graph. Presented at NVISION – The World of Visual Computing, 2008.

[12] D. Reiners. *OpenSG: A scene graph system for flexible and efficient realtime rendering for virtual and augmented reality applications*. PhD thesis, Fraunhofer IGD, Germany, 2002.

[13] John Rohlf and James Helman. IRIS performer: a high performance multiprocessing toolkit for real-time 3D graphics. In *Proc. SIGGRAPH*, volume 28, pages 381–394, 1994.

[14] Paul S. Strauss and Rikk Carey. An object-oriented 3D graphics toolkit. *Proc. SIGGRAPH*, 26(2):341–349, 1992.

[15] G. Voß, J. Behr, D. Reiners, and M. Roth. A multi-thread safe foundation for scene graphs and its extension to clusters. In *Proc Eurographics Workshop on Parallel Graphics and Visualization (EGPGV)*, pages 33–37, 2002.

[16] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*. Addison-Wesley, 1994.