DRiVE: An Example of Distributed Rendering in Virtual Environments
Anton Sigitov, Thorsten Roth, Florian Mannuss, Andre Hinkenjann
Institute of Visual Computing, Bonn-Rhein-Sieg University

ABSTRACT
Most Virtual Reality (VR) applications use rendering methods which implement local illumination models, simulating only direct interaction of light with 3D objects. They do not take into account the energy exchange between the objects themselves, making the resulting images look non-optimal. The main reason for this is the simulation of global illumination having a high computational complexity, decreasing the frame rate extremely. As a result this makes for example user interaction quite challenging. One way to decrease the time of image generation using rendering methods which implement global illumination models is to involve additional compute nodes in the process of image creation, distribute the rendering subtasks among these and then collate the results of the subtasks into a single image. Such a strategy is called distributed rendering. In this paper we introduce a software interface which gives a recommendation how the distributed rendering approach may be integrated into VR frameworks to achieve lower generation time of high quality, realistic images. The interface describes a client-server architecture which realizes the communication between visualization and compute nodes including data and rendering subtask distribution and may be used for the implementation of different load-balancing methods. We show an example of the implementation of the proposed interface in the context of realistic rendering of buildings for decisions on interior options.

Keywords: Virtual Environments, Parallel Rendering, Distributed Rendering, Load Balancing, Global Illumination

1 INTRODUCTION
Global illumination methods still remain challenging because of their high computational complexity. Due to vastly increased performance of computer graphics hardware in the last decade and new technologies like NVIDIA’s CUDA [11], it becomes possible for application developers to run their rendering algorithms at higher framerates. Despite the rapid progress there is still a wide range of applications where the desired performance cannot be achieved. Also, latest network technologies with a higher throughput like 10-gigabit Ethernet and low prices of hardware open the opportunity of building efficient rendering farms, consequently making the concept of distributed rendering more meaningful.

In this paper we distinguish between parallel and distributed systems as in [2]. The main difference between both lies in the communication layer. While parallel systems use mainly high-performance and thus very expensive interfaces to communicate with each other, distributed rendering systems utilize more common network technologies such as Ethernet. Both systems have local memory on each node which cannot be shared easily among them. Therefore some kind of managing functionality must be provided, which distributes the data to the compute nodes and assembles the results. This functionality is called load balancing and is a crucial part of any parallel and distributed system.

In this work we concentrate on the creation process of a distributed rendering system targeting a performance increase of the rendering engines. The engines employ local illumination models using ray-based rendering approaches. We also cover the main components of the distributed rendering system and provide the necessary steps for integrating a distributed rendering concept in any existing framework for virtual reality. Finally, we also present our own implementation of the distributed rendering interface and several projects where this interface has already been used. Note that while in this paper we aim for multiple nodes rendering the output for one single display, thus leading to the necessity of transporting the image data over the local network, our approach can also be used for distributed rendering for individual displays, e.g. like in a CAVE-like environment. In this context it would also be possible to attach the displays directly to the respective rendering nodes.

The paper is structured as follows: Section 2 contains information about related work, including information about other distributed rendering frameworks. In section 3 the integration steps of distributed rendering concepts in some framework for virtual environment are described. In section 4 we describe the architecture of our own distributed rendering interface, which is done generically and can be used as a template for any further implementations. In section 5, multiple load balancing approaches are described which may be integrated together with the distributed rendering interface. In section 6 some of our own projects employing our distributed rendering interface will be shortly presented. In section 7, results are concluded and possible future work is presented.

2 RELATED WORK
The idea of distributed rendering is not new. Therefore, there already are several frameworks allowing for the integration of distributed rendering concepts in VR applications. Subsequently, we present a short overview of some of them.

WireGL [4] is Stanford’s system for scalable interactive rendering on a cluster of workstations, which provides the common OpenGL API to distribute the rendering subtasks among the server nodes. These so-called streams are considered to be a sequence of OpenGL commands. The framework makes use of the sort-first rendering approach, as described in [10]. It provides support for different types of output systems like multi-projector wall-sized displays and single monitors. WireGL is limited to rasterization rendering methods because it is based on the OpenGL technology.

Chromium [5] is a successor of the WireGL system and was developed to extend the concept of it. So for example it may be used not only for creation of the sort-first, but also sort-last parallel graphics. The main novel feature of Chromium is the possibility of developing and integrating different types of the stream filters, where WireGL provides only three of them: sorting stream into tiles, dispatching of the streams to server nodes and reading back a frame buffer from render nodes. The framework allows for building of parallel rendering algorithms either on top of or integrated in it due to its generic implementation. Chromium is also limited to rasterization rendering approaches just as WireGL.

The Real-Time SceneGraph framework [14] is not really designed for distributed rendering approaches, but provides a good...
separation of the implemented scene graph and specific renderer instance. Such a separation provides the opportunity to use this framework with any type of render-engine, utilizing both rasterization or ray-based methods. By adding network functionality, the framework may be converted into a distributed rendering system.

Equilizer [3] is a toolkit for scalable distributed and parallel rendering which can work on cluster or in a shared-memory system as well. Based on OpenGL, it focuses on the rasterization rendering approach but it could be used with a ray-based renderer as well. The disadvantage of Equalizer is the fixed pipeline, which prohibits the integration of flexible post-processing routines for rendered images.

The last framework for distributed rendering presented in this section is called DRONE [12], developed at the Computer Graphics Lab of the Saarland University. This framework covers a wide range of scenarios which could be used together with distributed rendering approaches, such as single-screen rendering, multi-screen rendering, remote rendering and collaborative rendering. The framework is built on top of the Network-Integrated Multimedia Middleware [7] that makes it possible to create scalable and flexible systems for distributed rendering. DRONE specifies a number of different nodes, like render node, manager node, assembly node etc. which provide different functionalities and must be considered during system composition. The framework implements a fixed dynamic task scheduling method by dividing the render area into tiles and treating them as rendering subtasks which are assigned to the compute nodes. After a rendering node has processed its subtask it will be assigned a new one until all the tiles are rendered and submitted to the main node. In contrast to this, our approach is able to handle multiple different load balancing strategies (see section 5).

We propose a generic approach for the flexible and extendable integration of a distributed rendering and load balancing system into various applications with easily exchangeable components rather than a concrete software-framework.

3 INTEGRATION STEPS

The first step which is usually done before the integration of new components is an evaluation of the actual state. In this case the framework for virtual environments has to be analyzed. The result of this evaluation in most cases provides an analysis to the question if there is a clear separation between the framework and the render-engine. If the result is negative, detachment of the render-engine from the framework must be done before proceeding with step two. In most cases however, such a separation is already present. The possible initial state of the framework and the result of the first step is shown in figure 1.

At the beginning of step two, the VE framework is detachable from its render-engine. The render-engine may be fully or partially independent. Fully independent means that the render-engine is not built upon any data type implementations of the framework and does not use any utility functions of it. Partially independency means that some of the data types and utility functions might be used. Both cases may be supported by now. In this step a small interface including a client and server part of the future distributed rendering interface have to be implemented and integrated between the framework and its render-engine. This can be achieved by creating two classes, consequently called client and server. Both classes will define an equal set of methods, which provide the possibility of putting the scene elements of the VE framework into the render-engine. The set of methods to be implemented depends largely on the defined requirements. After implementing aforementioned classes, the pipeline setting will look as follows:

1. The virtual environment framework forwards the desired data of the 3D scene to the client instance using one of its member methods.
2. The client instance transfers the data to the server instance.
3. The server instance provides the received data to the render-engine.
4. The rendered image is sent back to the VE framework.

The result of the second step is shown in figure 2.

In the next step the serialization and deserialization functionality of data types is implemented. This is necessary because the single nodes of the system do not have any shared memory per definition and therefore transmit the data through the network layer. Serialization allows for conversion of the data type instances into a format ready for network transport, while deserialization is responsible for restoring these on the recipient side. At the end of this step the update or newly-implmented uniform data types are integrated into the existing data transmission pipeline. By now the VE framework should work as before, with some overhead caused by the interface between it and render-engine.

In the fifth step the VE framework and the render-engine are distributed to the different network nodes. This step necessarily
implicates the binding of network library to the system. Afterwards, the data transport pipeline looks as follows: The client instance serializes the data and forwards it to the server instance using the network functionality. Because the methods of the server instance cannot be called directly anymore, it must be reported which of them should be invoked, e.g., by enumerating these methods and transmitting the corresponding identifiers together with the other data. The server instance then uses the received method identifier to forward the data to the proper function. The other parts of the pipeline should stay as before. The resulting system is shown in figure 4.

By now the distributed memory system has been created, but it can utilize only one rendering node because of missing task distribution functionality. To make use of more than one compute node the load balancing component has to be added. The implementation of this component is done in the last step. The load balancing component runs on the visualization node, where the client component of the distributed rendering system will forward all rendering requests from the main application. After receiving the rendering requests, the load balancer will generate a number of subtasks depending on its implementation and distribute these among the connected server nodes. The compute nodes will provide the load balancing component with computed partial images, which can in turn be assembled into the output image. Note that other scenarios are also possible, e.g., rendering for different outputs in a CAVE-like environment and connecting the rendering nodes directly to the respective outputs. Different load balancing approaches are presented in section 5 of this paper. A detailed taxonomy of the load balancing methods for different parallel and distributed rendering approaches may be found in [2]. The architecture of the final system for distributed rendering is shown in figure 5.

4 INTERFACE ARCHITECTURE

In this section we describe our own implementation of the distributed rendering interface that was created and integrated into multiple frameworks based on the steps described before. The architecture of the interface consists of five main components: Client, Server, Load balancing, Uniform data types, and Network library component. The client and server API is implemented in a way that makes it possible to use the interface in retained-mode [9] as well as immediate-mode. In retained-mode the entire scene of VR application will be transmitted to the server nodes before the first rendering request takes place. After that the VR application informs the compute nodes only about updates such as transformation of the virtual objects or camera movements. In immediate-mode the whole scene is transported to all server nodes before rendering each frame. In most cases the retained-mode is preferred because of the immediate-mode’s huge performance hit. The choice of the desired mode refers only to the distributed rendering interface and does not affect how the render-engine works. This solely depends on its specific implementation. The class diagram and the collaboration diagram of the interface are shown in figures 4 and 7.

Figure 4: Result of the fifth step

Figure 5: Architecture of the system (result of the sixth step)

4.1 Client component

The client component is responsible for the serialization and transfer of control and data packets to the compute nodes. Control packets contain a small amount of data like update informations of scene elements. They are transmitted shortly before a new rendering request takes place. The data packets usually contain a large amount of data which describes the objects itself, e.g., meshes. These big packets should be transferred during the initialization step before the first rendering request to avoid a network bottleneck.

The client component also redirects the rendering request from the client application to the initialized load balancing module. Therefore the visualization node (running the client component) is also treated as a management node. Such distributed rendering systems with one central management node are called centralized [2]. The client component is initialized within a client application by passing the network configuration in text form and an instance of the desired load balancing component. It offers a wide range of methods for the application, which may be used for the transfer of single elements of any 3D scene to server nodes and also for updating and deletion of these. It also provides methods for putting the rendering requests and retrieving rendering results.

Figure 6: Class diagram of the distributed rendering interface
4.2 Server component
The server component is responsible for receiving the control and data packets from the visualization node as well as their deserialization and passing of the restored objects to a render-engine. The render-engine should therefore provide a set of methods for forwarding the received data. Another responsibility of the server component is the transfer of rendering results back to the visualization node. The initialization of the server component is done by passing the identification number of the compute node (which should be unique in the entire network), a network configuration in text form and an instance of the render-engine. Our implementation of the server component consists of the ServerStub and CommandObserver elements.

The ServerStub element is the place where the raw data comes in. This data is then converted to the specific instances of uniform data types or subtasks using deserialization methods. After the conversions, the data is passed to the CommandObserver element.

The CommandObserver element represents the connection point of the server component and the utilized render-engine. In its pure form it is only a set of the methods, which are then used by the specific implementation to get the data from the ServerStub element and put it into the render-engine. If necessary, further conversions for the render-engine are also done here. Within the render method, the rendering results are put in an appropriate buffer, which is then transmitted back to the visualization node.

4.3 Uniform data types
The uniform data types represent a specification of the scene elements, such as meshes, cameras, lights, materials and textures, which are shared between the visualization node and compute nodes. Such an approach is necessary because a framework for virtual environments and a render-engine can implement and handle the data structures of virtual objects in different ways. The uniform data types also provide methods for serialization and deserialization. Serialization takes place on the client node and is used for conversion of the attributes of the specific data type into a byte stream which will then be passed through the network layer using the underlying network library. After a server node receives those byte streams, it will use the deserialization method for the specific data type to restore the transferred object. The uniform data types provide the possibility to pair different render-engines with any VE framework or vice versa, independently of how they implement and handle the data types internally.

4.4 Load balancing component
The load balancing component is responsible for preparing, serializing, restoring and distributing the rendering subtasks. It also implements the routines for combining, saving and optionally for post-processing of the rendering results. Our implementation of this component contains two elements: The generic task object and the abstract task provider.

The generic task object describes the data of the rendering subtasks. The task objects are generic because their implementation does not provide any instructions on how the task should be defined. The data transferred to compute nodes consists of the following parameters:

- one integer describing the load balancing method currently used, so the render node can correctly interpret how to handle the data
- A set of custom string, float and integer parameters. In our path tracing application, we use these to configure the rendering server on-the-fly regarding the environmental conditions and sampler behaviour.

The generic task object also provides methods for serialization and deserialization of the data it contains.

The abstract task provider implements a set of methods which should be overridden by the specific implementation of the load balancing approach. The implemented logic in these methods describes how the specific sub tasks will be allocated and how the received partial results will be assembled into the output image. For this reason with each rendering request the task provider gets access to the image buffer, which is stored in the client component. This is where the resulting image is put.

4.5 Network component
The network component is an obligatory part of any distributed rendering implementation by definition. Our current implementation builds on the gulNetwork library, which has been developed at the Bonn-Rhein-Sieg University and utilizes basic POSIX sockets with the TCP protocol. However, the interface is generally not bound to any specific network technology or network library, which is why these should be treated as freely exchangeable components. The choice of the network component has to be made wisely and such aspects as data packaging and utilized protocol should be considered due to unnecessary overhead they can cause on the entire system.

Figure 7: Collaboration diagram of the provided system
5 Example implementations of the load balancing methods

The first and the simplest load balancing method is based on blocks of fixed size, which are dynamically distributed among the compute nodes. Using this method the whole render area is divided. After the rendering request has been put by the VE application, all server nodes receive one rendering subtask, which contains the description of the render area. Each server completes its task and delivers the results. The client saves the received information and checks if there are any blocks left to be rendered. If this is the case, the client creates a new rendering subtask for this server. If there are no more empty blocks, the compute node stays idle until a new rendering request is received.

The second approach uses dynamically adjusted blocks instead of blocks of fixed size. The main difference to the first method lies in the calculation of block sizes for each server node. The decisions about how the rendering area should be split is based on rendering statistics from previously computed frames. If the rendering process is not interactive but rather generates the images only for some varying points of view, then it should be divided into two steps: statistical gathering and rendering. At the beginning of the first step the render area is split into segments of equal size. The number of segments is set equal to the number of compute nodes. During the statistical gathering steps the blocks are assigned to the server nodes which in turn do their computation. After each completed rendering request the bounds of blocks will be recalculated. Therefore the blocks converge to an optimal size with an increasing number of segments. In the rendering step the computed blocks are used for task subdivision. The subtasks are then assigned to the respective server nodes, resulting in approximately equal rendering times for all nodes. The process of this method is shown in figure 8.

![Figure 8: Load balancing with dynamically adjusted blocks](image-url)

Another approach to decrease the image calculation time using distributed rendering is splitting the complexity of computation rather than dividing the rendering area. This we will use a path tracing approach as example here.

Path tracing is a rendering method which generates physically correct images of high quality. It is based on statistical methods from the area of Monte-Carlo-Simulation. The main idea of this method is to shoot some amount of rays into the scene generating random paths considering material properties of intersected objects. A small number of rays results in a high amount of noise in the image while a larger number of rays makes it possible to produce high-quality, photorealistic images, but increases computation time vastly. Based on the nature of path tracing, a different parallelization strategy can be suggested, which lets the compute nodes render the whole image with a specific number of samples. The images will then be merged by computing the mean value for each pixel. In this method two cases have to be distinguished depending on the architecture of the underlying distributed rendering system. If the system consists only of equally equipped compute nodes, all samples should be equally divided between them. If the system discovers differently configured rendering servers in a pre-render step similar to adaptive block scheduling mentioned before, samples can be distributed in a non-uniform way among the servers. This way, the rendering time on all nodes will be roughly equal. This approach does not satisfy scalability requirements, because all rendering servers respond with a full-size image, meaning that network traffic will linearly increase with an increasing number of servers.

The above load balancing approaches make the visualization node responsible for the subtask generation causing some overhead on the client side of the system and also in the network layer since the tasks have to be distributed among the servers. In order to avoid this, autonomous load balancing approaches may be used instead. Such a method requires the implementation of algorithms which on the one hand provide the compute nodes with the possibility to compute their render areas independently from the management node and on the other hand make it possible for the visualization node to assemble the generated parts of the images in an output image. The simplest examples for this strategy are subdivisions of the image in lines (line scheduler) or pixels.

We have already demonstrated the provided method on a distributed rendering system with one visualization node and three compute nodes. Each compute node received an identification number and the amount of render nodes in the entire system. Using these two parameters, it is easily possible to subdivide the render area into lines, which are then computed by the render-engine. This means that e.g. the server node with id 1 starts the rendering at the first line, the next line will be the fourth, because it is known that there are three server nodes in the system presented and the remaining two servers will take care of the second and third line. The next lines will be the seventh, tenth and so on.

After the rendering nodes have completed their work, they respond with the generated sub images, which consist of the rendered lines without gaps in order to avoid a performance hit due to unnecessary network traffic. The visualization node receives the sub images and assembles these into an output image. Block sizes of arbitrary height could also be defined for this scheduling method. This means that each server would render an image consisting of sets of \( n \) consecutive lines, followed by \( n \cdot (\text{numServers} - 1) \) lines handled by other servers. Again, the latter image area is not contained in the result image sent back to the client. The intermediate results and the output image produced by using this method are shown in figure 9. Similarly, this approach may be implemented by using single pixels or groups of pixels instead of lines.

Autonomous load balancing methods may not only decrease the overhead in task computation and distribution, but also decrease...
Figure 9: Autonomous load balancing using lines for screen subdivision with two compute nodes: (a) and (b) rendering results of render-server one and two respectively, (c) and (d) image buffers produced by the render-server, (e) output image composed by load balancer

the probability of some compute node receiving a subtask which is much more complex then the other. Surely a wide range of other algorithms may be developed to be used with this approach.

6 Example applications

The described approach was already successfully applied in the TraCell [1] project. The aim of this project was the development of a distributed rendering system for medical applications, implementing parts of a global illumination rendering model. The main objective and challenge of this project was to make the system capable of being used for real-time visualization. The rendering hardware consisted of several render servers, PlayStation 3 consoles, connected over Gigabit Ethernet. The described interface for distributed rendering was used to connect our own framework for virtual environments, called basho [8], on the client side and a ray tracing render-engine on the server side of the system. The complete system made it possible to utilize the visual effects such as physically correct refractions, reflections and shadows in a wide range of interactive applications.

Currently we are using the presented interface for distributed rendering in the project IV AB. The main objective of this project is the integration of global illumination effects in the Vi2000[15] system. This system for planning and previewing prefabricated houses has an own visualization component. However, it only implements a rendering method based on strictly local illumination models and does not take into account realistic illumination effects as interreflection between multiple surfaces of the scene elements. Integrating these effects should give customers a much more realistic impression of their future homes. The currently used system in the project consists of one visualization node and three compute nodes.

The visualization node, which represents the client side of the system, runs the Vi2000 software with its own visualization component. The hardware of this node is on par with any middle-range personal computer. There are no real computational challenges on this side of the system. The visualization component of Vi2000 is used for navigation through the virtual houses and interaction, like exchanging wallpapers or flooring.

If a customer wants to see how the currently chosen configuration will look like in reality, the compute nodes will get involved into the process by sending the rendering request to them. These nodes represent the server side of the system and run the path tracing rendering engine, which is responsible for generating realistic, high-quality images. The visualization node and compute nodes are physically connected through a 10Gbit Ethernet interface. Each rendering node consists of the following hardware components:

- 2x intel XEON E5-2637 2C 3GHz 5M 8GT/s CPU
- 16GB PC3-12800ECC RAM
- 60GB SATA SSD hard drive
- 2x Nvidia GeForce GTX 680 2GB PCIe-x16 Gen3 graphic cards

Listing 1: Example for usage of the distributed rendering interface

```c
/ / Load server config and initialize server system
 drive::ServerConfig sconf;
 sconf.readFromFile( pathToFile );
 drive::getClientServerSystem( sconf );

/ / Pass the drive client an arbitrary triangle mesh
drive::setGeometry( triMeshData );

/ / Initialize task scheduler
/ / Set resolution and number of samples for path tracing
 drive::LineTaskScheduler aLTS;
 drive::RenderTaskProvider *aRTP;
 aRTP = aLTS.getWorkProvider( );
 aRTP->setScreen( 0, 0, renderWidth, renderHeight );
 aRTP->setNumSamples( numSamples );
 drive::getClientTaskScheduler( &aLTS, aRTP );

/ / Main loop
while (!quit) {
  drive::updateCamera( camera );
  drive::renderScene( );
  drive::syncRenderer( );

  / / Get and process returned image
  uint8_t* result = drive::getClient::getColourBuffer( );
  ...
}
```

Such a configuration of the render nodes is roughly equal to some high-end personal computer, which helps keeping the price of the whole system reasonable. The path tracing render-engine is implemented using NVIDIA’s OptiX technology, while post-processing
of the image data is done using our CUDA-based post-processing framework GIP[13].

The IVAB project is not completed yet and the first thorough tests will start in the near future. We show early test results, giving first hints on the performance of our non-optimized example implementation in figures 10 and 11.

Figure 10 shows results for the rendering of a simple cornell box with a varying number of compute nodes and rendered samples using the line scheduler. Our results have shown that the speedup stays more or less constant from two samples upwards and is relatively far away from the optimum. This means that we achieve a speedup of 1.68 with two nodes and 2.34 with 3 nodes. Our tests using the sample scheduler with the same scene however have shown that a better speedup is easily achievable. This scheduler delivers a speedup of 2.19, 2.76 and 2.89 for 12, 60 and 120 samples computed in one chunk, respectively. Clearly, the network overhead can be compensated with a bigger amount of samples.

Figure 11 shows a comparison of the line and sample schedulers for the ISS scene shown in figure 12, consisting of roughly 1.5 million triangles. This illustrates the aforementioned dependency of the sample scheduler’s performance on the requested number of samples. When rendering only a few samples, it is beneficial to use the line scheduler, while the sample scheduler approaches the optimum speedup with an increasing number of samples.

As this is our first implementation of an interactive, parallel global illumination renderer for the DRIVE concept, we are currently experimenting with different load balancing approaches to further improve the parallel efficiency. Also, currently the path tracing renderer is not meant to be used for interaction, but to be combined with another rendering approach such as basic rasterization, mainly due to the high noise in images with low sample numbers.

However, we have already experimented with various post-processing approaches for image denoising, so that using this rendering technique for interaction could also be possible in the near future. For purely diffuse scenes, we can already perform path tracing at lower resolutions like 720p at interactive frame rates. Note that it is quite difficult to give exact numbers on which frame rate can be achieved with a specific number of nodes, as this depends not only on the amount of geometry, but also on its spatial distribution, which might lead to difficult lighting conditions, thus making a bigger number of samples necessary. Accordingly, the viewer’s position and orientation also play an important role. Thus, while for one perspective path tracing may work very well for real-time visualization, another one might take half a minute until acceptable quality is achieved.

7 Conclusion and future work

In this paper we have shown how the concept of distributed rendering may be simply integrated into existing VE applications or frameworks. The entire process only requires the creation of a tiny, flexible interface, which describes the communication routines between application and rendering engine and the integration of a network library of the developer’s choice.

The interface can be adopted by any application and render-engine due to uniform data types. At our university we are using such an interface to achieve a reasonable speed-up for render-engines based on global illumination concepts, but there are no reasons not to use this interface for any other application types. So for example, render-engines based on OpenGL or DirectX technologies may be put on compute nodes, which will render the entire scene with different shaders and transfer the results back to the visualization node, where they will be assembled in the desired way.

The interface can also be used for multiple-screen rendering, which is used by CAVE-like systems. Thanks to a generic implementation of the task objects, different load-balancing approaches may be integrated without any big effort. The interface is fully extendable, which makes it open for further experiments and feature implementations like server-server communication. Due to implementation of the uniform data types, both the VR framework on the client side and the render-engine on the server side may be freely
exchanged.

**ACKNOWLEDGEMENTS**

This work was sponsored by the German Federal Ministry of Economics and Technology (grant no KF2644106ED1). We would like to thank our project partners Softwareparadies GmbH, PDV-Systeme Sachsen and Fraunhofer IAO for the close cooperation as well as Andrey Klimenko for the ISS model.

**REFERENCES**


