D4.1.3
Real-time Rendering and Lighting Demonstrator
Deliverable due date: 31 March 2016 (M30)
Actual submission date: 31.03.16

Project no: FP7 - 61005
Project start date: 01.10.13
Lead contractor: The Foundry
<table>
<thead>
<tr>
<th><strong>Project ref. no.</strong></th>
<th>FP7 - 61005</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project acronym</strong></td>
<td>DREAMSPACE</td>
</tr>
<tr>
<td><strong>Project full title</strong></td>
<td>Dreamspace: A Platform and Tools for Collaborative Virtual Production Dreamspace</td>
</tr>
<tr>
<td><strong>Security (distribution level)</strong></td>
<td>PU</td>
</tr>
<tr>
<td><strong>Contractual date of delivery</strong></td>
<td>Month 30, 31/03/2016</td>
</tr>
<tr>
<td><strong>Actual date of delivery</strong></td>
<td>Month 30, 31/03/2016</td>
</tr>
<tr>
<td><strong>Deliverable number</strong></td>
<td>D4.1.3</td>
</tr>
<tr>
<td><strong>Deliverable name</strong></td>
<td>Real-time Rendering and Lighting Demonstrator</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Demonstrator</td>
</tr>
<tr>
<td><strong>Status &amp; version</strong></td>
<td>v2 - Final</td>
</tr>
<tr>
<td><strong>Number of pages</strong></td>
<td>19</td>
</tr>
<tr>
<td><strong>WP / Task responsible</strong></td>
<td>Philipp Slusallek, DFKI</td>
</tr>
<tr>
<td><strong>Other contributors</strong></td>
<td>Arsène Pérard-Gayot, UdS; Georg Tamm, UdS; Richard Membarth, DFKI</td>
</tr>
<tr>
<td><strong>Author(s)</strong></td>
<td>Philippe Bekaert, iMinds</td>
</tr>
<tr>
<td><strong>EC Project Officer</strong></td>
<td>Alina Senn</td>
</tr>
</tbody>
</table>

**DOCUMENT HISTORY**

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason of change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>17-12-2015</td>
<td>Document created by Richard Membarth</td>
</tr>
<tr>
<td>1.1</td>
<td>14-03-2016</td>
<td>Version for internal review (14 days before submission date)</td>
</tr>
<tr>
<td>1.2</td>
<td>31-03-2016</td>
<td>Revisions in response to review: final versions submitted to Commission</td>
</tr>
<tr>
<td>1.3</td>
<td>DD-MM-YYYY</td>
<td>Revised version produced in response to Annual Review</td>
</tr>
</tbody>
</table>
# Table of Contents

1. Executive Summary .......................................................... 4  
2. Introduction ........................................................................ 5  
3. Lighting Simulation ............................................................... 5  
4. Distributed Rendering ........................................................... 6  
   4.1. Setup Flexibility ............................................................. 8  
   4.2. Load Balancing .............................................................. 10  
   4.3. Comparison ................................................................. 11  
   4.4. Performance ............................................................... 13  
5. User Guide ........................................................................... 14  
   5.1. Cluster Setup ............................................................... 14  
   5.2. Display Client Setup ...................................................... 15  
6. Conclusion ........................................................................... 16  
7. References ............................................................................ 17
1. Executive Summary

Dreamspace relies on high-performance ray tracing for live preview of 3D scenes with lighting support. The performance of a preview renderer is critical since the director has to make decisions on-set. To render scenes in real-time, a distributed rendering approach is necessary.

This deliverable presents a demonstrator for integration with the other components of the virtual production pipeline. The demonstrator requires three main components:
- Lighting simulation
- Distributed rendering
- Interaction and connection with LiveView

For lighting simulation, a variant of the Vertex Connection and Merging (VCM) has been developed. This state-of-the-art method allows complex lighting settings to be efficiently and correctly rendered. The light models of WP5T1 have been implemented in the renderer. This makes the rendered image match the real lighting on-set.

The distributed rendering system has been ported to run on Linux systems and supports GPU rendering through the Dreamspace renderer.
2. Introduction

The work of this deliverable focuses on real-time rendering through distributed rendering, which is necessary to make creative decisions on set. We propose a system which integrates a global illumination renderer with the lighting models from WP5T1 “Virtual Production Design, Visualisation and Editing” and a workload distribution scheme on a cluster to provide real-time feedback on the lighting set-up.

The hardware requirements for the cluster are reasonable, and only uses consumer-level products (CPUs, GPUs). Communication of the scene is done via an HTTP scene server, and the rendered images are transmitted using standard Ethernet. This makes our system affordable and interesting for practical uses.

Figure 1: Overview

Our system is integrated into LiveView from The Foundry, which acts like a hub for all the components of the project. We have developed a plugin that transfers the scene to our render cluster and communicates the rendered image back to LiveView. The lighting parameters, calculated on set from W5T1, are sent to LiveView which in turn sends them to our plugin.

3. Lighting Simulation

Our renderer supports three lighting simulation algorithms: Path Tracing (PT), which was already implemented for D4.1.2 “Implementation of Selected Modules for Real-Time Ray Tracing and Advanced Lightning Simulation”, Bidirectional Path Tracing (BPT) and Vertex Connection and Merging (VCM), which are recent additions developed since then. PT is the simplest of those three, and converges slowly for complex lighting situations. It is however performing better than the other two for simple lighting (no specular-diffuse-specular interactions, no caustics).
BPT is more robust and can render complex materials better than PT, but still cannot render efficiently scenes with specular-diffuse-specular (SDS) interactions. VCM further improves on BPT by adding a merging strategy – akin to photon mapping – when evaluating paths, which increases convergence for SDS paths.

The choice of rendering algorithm highly depends on the scene and we used PT for this deliverable, since the scenes we are working with are using only simple materials (mostly glass and Lambertian surfaces). This can of course be changed by a simple configuration flag, currently not exposed to the user.

![Comparison of PT, BPT, and VCM](image)

Figure 2: Comparison of PT, BPT, and VCM (from left to right). All images have been rendered in 60 seconds.

4. Distributed Rendering

Figure 3 illustrates the distributed rendering pipeline. The goal of the pipeline and the Dreamspace ray-tracer is to provide the display client with high-quality, globally
illuminated rendering results. The pipeline should be real-time capable, and give the user immediate feedback when navigating or manipulating the scene.

![Architecture of the distributed rendering system.](image)

The display client runs the LiveView application, which can load and interactively render scenes authored in Katana. LiveView supports a plugin mechanism to integrate renderers. A plugin has access to the initial scene and consequent updates (like camera movement), and passes its results back to LiveView for display.

The Dreamscape rendering plugin exports a scene into a generic and portable XML format, which supports transformations, instancing and object groups and hierarchies. The export is cached on disk, thus speeding up the next loading of the scene. Deleting the cache will trigger a fresh export should the scene have changed or been replaced. The exported scene is made available to the outside via an HTTP server. The HTTP server runs independently to provide the scene even if LiveView is not up.

The plugin does not render locally, but connects to the master node of a rendering cluster which downloads the newly exported scene from the HTTP server. The master translates the scene into a binary format cached on disk. The cache allows to quickly load the scene into a renderer. The master also creates a second cached version optimized for network transfer, and distributes this cache across the rendering nodes. Download and translation only occur if a new version of the scene is advertised by LiveView. Transfer to a rendering node only occurs if the node has not already cached the latest scene version.

The rendering server provides a generic API allowing different renderers to be integrated. The display client may select any of the available renderers. The main one is the Dreamscape renderer described in D4.1.2 “Implementation of Selected Modules for Real-Time Ray Tracing and Advanced Lightning Simulation”. Once the renderer on each node has loaded the scene, updates issued by LiveView trigger the rendering of new frames. The server supports lightweight updates like camera, light, and material property changes, as well as dynamic buffers and textures. The plugin sends updates to the master...
node, which distributes them in the cluster. The master collects the partial images produced by the nodes and forwards the final result to LiveView for display.

The pipeline is specifically designed for real-time operation. The execution model is asynchronous: allowing client, network, and rendering to operate in parallel. While the cluster renders the current frame, the network transfers the previous one to the client which already prepares and sends updates for consequent frames. A server queues updates when occupied and restarts its renderer immediately from the queue, enabling full utilization. However, filling the queue with outstanding frame requests should be avoided, since it results in the decoupling of the user input with the displayed result. Therefore, the client only requests a selectable number of frames in advance.

If the user interaction stops, the server allows progressive rendering to refine the image up to production quality if uninterrupted. This feature will be available in the next update of the framework.

Each node encodes its partial rendering result using S3 texture compression (S3TC). S3TC is a block-based encoder which allows fast parallel execution. The distributed encoding overhead is minimal and negligible (with eight nodes, around 0.07ms per node for 720p images on a six year old Xeon X5650 CPU). The compression ratio is fixed at 8:1 for 32bit RGBA images. This enables even a 1GBit/s cluster network to accumulate the results at the master node quickly. A high-bandwidth Ethernet or even InfiniBand setup is therefore recommended, but not mandatory. Further, the client can directly upload S3TC images to the GPU for decoding and display (currently not supported by LiveView).

S3TC can produce artifacts for sharp edges and gradients. Also, the bandwidth from the master node to the display client may be limited, for example when the client connects from a remote location over the Internet. Therefore, alternatives like JPEG images or H.264 video will be supported in a future version of the system. These methods require the master to perform a slower, non-distributed encoding pass from raw image input. Thus, the pipeline latency in the cluster increases. But at the same time, there is a more bandwidth-efficient transport to the display client, and a better image quality can be provided in some scenarios.

The master node may participate in the rendering, which is recommended as it reduces the network load in the cluster. The master can also be a dedicated network hub only giving access to the rendering nodes. If the master performs JPEG or H.264 encoding, it may also be feasible to avoid competition with a local renderer already working on the next frame.

4.1. Setup Flexibility

In addition to LiveView, a browser display client is available. The client utilizes an extended XML3D [1] which supports server-based rendering. Users have access through a standard web page without requiring a plugin. While this client currently only allows navigating through the scene, it can be extended to support further updates like adding
and manipulating lights. Figure 4 showcases the flexibility in the setup which is possible with the display client and cluster architecture.

![Figure 4: Exemplary distributed rendering architecture with browser clients and two server-side renderers.](image)

LiveView connects to a master node, which has access to five rendering nodes. The master keeps all these nodes up-to-date if a new scene export occurs. The upper three nodes plus the master are dedicated to provide the best quality renderings to the main display client. They could be in a high-speed cluster network with InfiniBand support. The lower two nodes provide preview renderings (for example using a rasterizer) to the remaining devices, which run the XML3D client. A commodity machine and network setup could be adequate here. A node may also support both renderers. All clients can independently navigate and update the scene within their rendering session, enabling different settings to be tested in parallel.

Lastly, the XML scene export provided by the HTTP server is XML3D compatible. The scene could therefore also be distributed to a XML3D client to perform local rendering and editing. This is an outlook not supported by the current system.
4.2. Load Balancing

Ray tracing is an embarrassingly parallel problem. The key for high performance is therefore the parallel utilization of execution units. However, the workload can be highly heterogeneous. Some image parts may be more expensive to compute than others, which depends on the number of intersection tests required to find hit points, the properties of the hit materials, and the amount of secondary rays being cast. To achieve linear scaling with more workers, load balancing aims to keep all workers busy until a frame concludes.

The rendering backend employs a two-level load balancing strategy. The master splits a frame between the nodes. On each node, the renderer distributes the traversal and shading across the available logical CPU cores and GPUs.

Dynamic load balancers initially assign tasks of possibly varying cost to the workers. When a worker becomes idle, it still receives ready tasks on demand. The worker either requests tasks from a central queue [2] [3] [4] or attempts to steal from other workers [5] [6] [7]. Task stealing removes the queue as a possible communication bottleneck as worker pairs can coordinate in parallel.

Under optimal conditions, dynamic approaches are known to scale linearly, and naturally handle heterogeneous workers. However, they induce communication and task management overhead during rendering, which increases with the number of workers. A low latency link between the master and the workers, and in case of task stealing between all workers, is mandatory. Dynamic load balancing is therefore ideally suitable for local thread-level scheduling with a moderate amount of CPUs being present on a single machine.

The rendering backend supports a commodity network setup, where low latency between the nodes cannot be guaranteed. A high number of nodes might also be used at some point to provide high-quality global illumination already at interactive frame rates. Further, dynamic load balancing requires to split the frame into tasks of fine granularity. Otherwise, in the end of the frame, some nodes might stall the rendering when busy with a demanding task while there are no more tasks left for the idle nodes. The Dreamspace renderer traces rays on a GPU. Dynamically assigning or removing small tasks is inefficient due to the transfer overhead between CPU and GPU.

The rendering backend consequently uses a static load balancer. A static approach assigns a fixed task to each node, which cannot change during a frame. Communication overhead during rendering is avoided, and the renderer can process its tasks in one batch. To achieve scalability, tasks must be chosen to equalize the rendering time on the nodes.

The classic approach is a pseudo-random scattering of pixels or pixel blocks among nodes [8] [9], which can achieve an even cost distribution. The disadvantage is the loss in cache locality as each node works across the entire image. Scattering easily integrates with any ray-tracer, while task stealing requires an invasive adaption of the renderer to allow handing back tasks from the renderer’s internal scheduling to the network.
More recent methods attempt an estimation of the cost distribution to determine a continuous task for each node [10] [11]. However, these newer methods scale not accurately on their own, but need the incorporation of a dynamic load balancer to account for possible imbalance.

In Dreamspace, a novel static load balancer was developed for a custom CPU ray-tracer which runs as a secondary renderer in this framework. The load balancer exploits frame-to-frame coherence in a real-time context. Based on timings acquired for the previous frame, a balanced task distribution can be derived for the next frame. This fine-grained timing mechanism is not applicable to a GPU renderer, though.

The master node runs a scattering load balancer once in the beginning of each rendering session. The tasks do not change between frames, so there is no further load balancing overhead. The scattering operates on pixel blocks which are aligned to enable SIMD instructions. The larger the block size, the better the renderer on each node can benefit from cache locality. On the other hand, a higher granularity can enable a more accurate load balance. Since the impact of both effects is scene- and view-dependent, the granularity is a selectable setting to fine tune the performance. Figure 5 illustrates how the image space may be scattered across the nodes.

![Figure 5: The load balancer splits a 720p image into blocks of 32x32 pixels, and scatters the blocks to ten nodes. Each colour indicates one node.](image)

### 4.3. Comparison

Several commercial solutions supporting distributed ray tracing have emerged [12] [13] [14] [15]. We focus on the two most prominent examples specifically targeting interactive rendering. V-Ray provides a setup with a scattering load balancer, where the master also acts as the display client. The master connects to the nodes via TCP/IP to collect raw
pixels. While V-Ray focuses on offline rendering, the extension V-Ray RT [12] enables interactive performance. The rendering engine is split into two versions. The CPU version is feature-rich and resembles the regular V-Ray offline renderer. The GPU version is optimized for performance, but has limited features.

NVIDIA offers a special hardware called Visual Computing Appliance (VCA) [13] at a price of around 50,000$. The VCA contains several high-end GPUs and comes with built-in support for OptiX [16] and Iray [17]. OptiX is a framework to enable generic ray tracing applications leveraging the GPU. Iray is a rendering solution built on top of OptiX which supports photorealistic, interactive, and real-time rendering modes. Other production-level renderers accelerated with OptiX are FurryBall [18] and Mental Ray [19].

The display client connects with 1 or 10GBit/s Ethernet to the VCA, which returns lossless images, JPEG, or H.264 video. The VCA takes care of distributing OptiX calculations and Iray rendering tasks to the GPUs. The interconnection of several VCAs via InfiniBand is possible. The cluster manager automatically handles scene distribution and load balancing for OptiX and Iray. Third-party applications are possible, but must handle the pipeline on their own. An example is V-Ray RT which can run on VCA.

Above solutions provide fully featured production-quality renderers and are already integrated into display client software like 3ds Max and Maya. While a complete set of state-of-the-art rendering features could not be addressed within Dreamspace, we provide a more flexible framework. V-Ray RT requires to connect all nodes including the display client in a local high-bandwidth network to transfer raw pixels. The setup is not extendable with third-party renderers. While VCA supports streaming encoded results to a display client outside the cluster, it depends on dedicated high-end hardware. VCA only has built-in support for OptiX and Iray. Third-party renderers must implement a custom scene distribution and load balancing pipeline.

The Dreamspace framework provides a flexible architecture which is already functional with commodity networking in the cluster. The display client can run in a web browser without requiring a plugin, giving users access from a standard web page with their possibly mobile devices. In addition to LiveView, the integration into other clients like 3ds Max and Maya is possible.

The pipeline overhead on top of the rendering is kept at a minimum through asynchronous execution of pipeline stages, distributed encoding of rendering results, and non-invasive static load balancing inducing zero overhead during rendering. A generic API allows to plug in third-party renderers. Several renderers can coexist to provide different levels of service as illustrated in Figure 4. The static load balancer applies to any scalable renderer. Furthermore, describing the core ray tracing algorithms in AnyDSL facilitates faster prototyping and integration of future developments in real-time ray tracing.
The scene distribution approach is similar to Iray on VCA and V-Ray RT. There is an initial heavy distribution and caching step of the on-disk scene managed by the display client followed by subsequent incremental live updates.

4.4. Performance

For performance evaluation, we considered a cluster made of up to 4 nodes, each containing a NVIDIA Maxwell GPU (GTX 970) and an Intel Skylake quad-core CPU (i7-6700K). We rendered the San Miguel scene at HD resolution (720p) using Path Tracing and 1 sample per pixel (per frame). The nodes in the cluster were connected using standard Ethernet on a GNU/Linux system running Fedora 23.

Using this setup, 3 frames per second (fps) can be rendered on a single node. This scales up to 10 fps when using 4 nodes. We expect further linear scaling when using more nodes.

Rendering setup:
- San Miguel scene
- HD resolution (720p)
- Path tracing
- 1 sample per pixel (per frame)

Software setup:
- GNU/Linux: Fedora 23
- CUDA 7.0
- Threading Building Blocks 4.3

Hardware setup:
- 1 Gigabit Ethernet interconnect
- 1 Intel Skylake quad-core CPU (i7-6700K)
- 1 NVIDIA Maxwell GPU (GTX 970)

Performance results:
- 1 Node: 327ms - 387ms / frame ~ 3FPS
- 2 Nodes: 218ms - 250ms / frame ~ 4.5FPS
- 4 Nodes: 96ms - 140ms / frame ~ 10FPS

The current implementation uses an optimized implementation of the ray traversal part in AnyDSL. This part is mapped to the GPU and achieves comparable performance to state-of-the-art implementations on the GPU and even outperforms them as shown in D4.1.2 “Implementation of Selected Modules for Real-Time Ray Tracing and Advanced Lightning Simulation”. The shading part of the renderer is implemented in C++ and executed in parallel on the CPU. This part restricts the overall performance since it is a) not vectorized for optimal resource utilization on the CPU and b) is not yet written in AnyDSL to alternatively make use of the GPU to avoid costly data transfers between CPU and GPU.
5. User Guide

The distributed renderer will be executed on a cluster that runs GNU/Linux and has standard Ethernet connection. The user will interact with the distributed renderer via a plugin in LiveView.

5.1. Cluster Setup

In order to deploy our renderer, we provide system images that contain the executables and configuration files necessary to run the cluster. The system installation will have a local account with the following credentials:

Username: dreamspace
Password: dreamspace

The distributed renderer is installed in the home folder in the dreamspace directory: 
/home/dreamspace/dreamspace

This folder contains 3 sub-folders for the scene, the launch configuration and the renderer itself:
/home/dreamspace/dreamspace/scene
/home/dreamspace/dreamspace/config
/home/dreamspace/dreamspace/renderer

The master node has to be configured so that it connects to the other nodes. The procedure is very simple, and consists in editing a text file called “nodes.txt”, present in the folder containing the renderer executable. This file contains the following:

0-1, 1-1
<addr>:<port>, 0; 0-1, 1-1
<addr>:<port>, 0; 0-1, 1-1
...

The first line sets some internal parameters currently not exposed to the user. Removing this line results in the master node not participating in the rendering, which can be feasible as described in Section 4. The following lines contain the IP addresses and ports of the rendering nodes to connect to. The master’s IP doesn’t need to be specified in the nodes.txt. The following example configuration runs on 4 nodes (including the master):

0-1, 1-1
192.168.2.100:8080, 0; 0-1, 1-1
192.168.2.101:8080, 0; 0-1, 1-1
192.168.2.102:8080, 0; 0-1, 1-1

The master and child nodes have to be run from the config directory. To do so, we recommend to open a terminal and type:
cd /home/dreamspace/dreamspace/config
./XML3DServerFork <port>

with <port> being the port to listen to.

If any connection problem occurs, check that the nodes are accessible in the network by pinging them and make sure every node is running correctly. If the problem is still present, check the network configuration, in particular the firewall settings. For each rendering session, a node uses an additional port starting with 19010 for the first client. Therefore, make sure to have a port range (e.g. 19010 to 19030) accessible on each node depending on the number of concurrent clients to be run.

Also check that the master can establish a HTTP connection to port 80 of the scene server to enable downloading a newly exported scene (see Section 5.2). If the master cannot connect to the HTTP server, it will use the currently cached version if available.

The node that first receives a connection request becomes the master. Only the master reads from the configuration file, there is no need to change the configuration files of the other nodes. The process to start the whole cluster is hence very simple:

1. Start the renderer on every node
2. Connect to the master node

5.2. Display Client Setup

LiveView should be installed with the plugin on the target machine. This requires a correct LiveView installation, and an environment setup to load the plugin at startup. These steps are described by the documentation of the LiveView system. Should the plugin not be the active renderer initially, the user can select “Dreamspace” in the schemas list or load the “dreamspace.schema” file from the “schemas” directory.

The rendering resolution should be a multiple of four on both axes to work with the S3TC encoder, which operates in blocks of 4x4 pixels. This gives support for most common resolutions like 1280x720 and 1920x1200.

There needs to be a file called config.txt in the directory where LiveView is executed. This file contains settings for the plugin, which are mostly optional. There is one mandatory line, which tells the plugin the address and port of the master node:

Server: IP:port (e.g. 192.168.2.99:8080)

The optional settings are (specify each on a new line):

Renderer: string
The renderer to use. Defaults to “dreamspace”.

Nodes: int
The number of rendering nodes to use. If this value exceeds the number of available nodes, it is automatically capped. Defaults to the maximum integer.

Real-time Quality: int
A value between 0 and 10 telling a renderer the desired quality for real-time rendering, with 0 being the lowest quality. The scale is abstract to be applicable to any renderer. Thus, it is up to a specific renderer to map the values to internal parameters. For example, the user may set this to 0 if only a single node is available, and increase the value as more nodes are added. Defaults to 0.

Maximum Frames Pending: int
The maximum number of frames the client may request in advance before having received a rendering result. Setting this to 1 creates a synchronous pipeline since the client waits for the result of each requested frame before requesting the next one. A value higher than 1 enables asynchronous execution, but may decouple the user interaction from the displayed result if the rendering is a noticeable amount of frames behind. The recommended value is 2 or 3. Defaults to 2.

Separately to LiveView, a standard HTTP server like Apache needs to be started on port 80 with its root directory pointing to a folder named “xmlCache” in the LiveView directory. The server gives the master node access to the scene in XML format exported by the plugin. When starting, the plugin exports the scene to the "xmlCache" folder if it does not already exist. Thus, simply deleting the folder triggers a new export and distribution of the scene should it have been replaced or edited in Katana. It is also possible to overwrite the folder with a previous export to trigger the distribution step only.

The overall procedure to export and distribute a scene can take several minutes. Due to the client- and server-side caching in place, consequent loadings of the scene take seconds.

The alternative browser display client is provided as a stand-alone folder, which can be made available via any standard web server. To setup the client, search for the “xml3d” tag in the index.xhtml web page. Edit the “server” attribute to point to the master node. Edit the “rendererType” attribute to select the renderer. Edit the “renderingNodes” attribute to change the number of nodes to use. Edit the “style” attribute to change the rendering resolution. Search for “camera_transformation” to edit the initial camera translation and rotation.

6. Conclusion
In this work, we have extended our high-performance ray traversal code running on a CPU as well as on a GPU with advanced implementations for global illumination for lighting support. For this, implementations using Path Tracing (PT), Bidirectional Path Tracing (BPT), and Vertex Connection and Merging (VCM) are provided. Distributed rendering on a cluster allows real-time rendering and lighting for on-set decisions.
The deliverable provides a working system which is extendable and provides a real-time preview of the scene lighting. We plan to further improve performance and simplify the system so that it is easier to install for the end-user.

7. References


