guacamole – An Extensible Scene Graph and Rendering Framework Based on Deferred Shading

Simon Schneegans, Felix Lauer, Andreas-C. Bernstein, Andre Schollmeyer, and Bernd Froehlich *Member, IEEE*

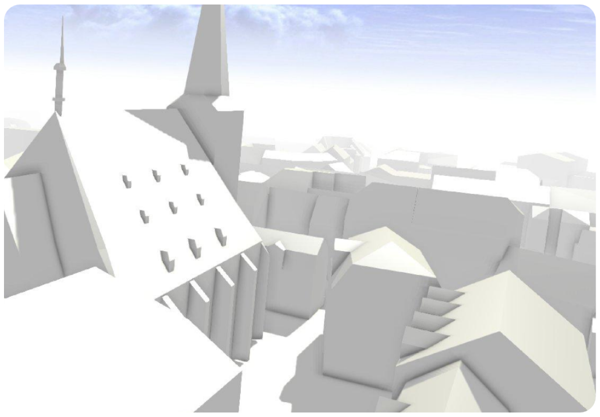


Fig. 1: guacamole allows rapid development of complex and immersive multi-user VR-applications.

Abstract— In this paper, we present guacamole, a novel open source software framework for developing virtual-reality applications. It features a lightweight scene graph combined with a versatile deferred shading pipeline. In our deferred renderer, the geometry processing is decoupled from subsequent shading stages. This allows us to use the same flexible materials for various geometry types. Materials consist of multiple programmable shading stages and user-defined attributes. In contrast to other deferred shading implementations, our renderer automatically infers and generates the necessary buffer configurations and shader programs. We demonstrate the extensibility of our pipeline by showing how we added rendering support for non-polygonal data such as trimmed NURBS and volume data. Furthermore, guacamole features many state-of-the-art post-processing effects such as ambient occlusion or volumetric light. Our framework is also capable of rendering on multiple GPUs for the support of multi-screen displays and multi-user applications.

Index Terms—computer graphics, virtual reality, deferred shading, scene graph

1 Introduction

Most virtual-reality systems offer a high-level interface to hide the complexity of the underlying rendering process from the application developer. Traditionally, a scene graph is used to organize the contents of the virtual scene. For rendering, the scene graph is traversed, serialized and passed to the rendering engine. In many modern 3D engines, such as Unity 3D [7], Unigine [23] or CryEngine [15] [16], rendering is based on a deferred shading concept [20]. Most of these systems perform very well in terms of visual quality, but an integration of user-defined geometry types and materials seems intricate. In particular, the support of advanced rendering techniques, such as volume ray-casting, direct rendering of higher order surfaces or point-based rendering, is difficult to achieve. Furthermore, in the implementations of the aforementioned systems the layout of the geometry buffers is fixed which impedes the ability to add user-defined materials and shading effects.

In contrast, open source scene-graph frameworks are often extensible in terms of user-defined geometry types and shading routines. However, most of them are based on a forward-rendering concept and lack the advantages of deferred shading approaches. Furthermore, in a forward-rendering approach, the programmable shaders used for geometric computations and shading may interfere with each other. In this case, either metaprogramming or deferred shading becomes necessary.

These considerations led us to the following set of requirements for a new framework for virtual-reality applications:

- *Extensibility*: A lightweight scene graph which can easily be extended with arbitrary geometry types and materials
- *Flexibility*: A flexible shading concept which allows for user-defined materials without being limited to fixed geometric representations or shading routines
- *Configurability*: User-configurable support for multi-pass, multi-screen, multi-GPU and multi-user rendering
- *Transparency*: A rendering pipeline which facilitates the integration of volume ray-casting and order-independent transparency

Based on these requirements, we developed a software framework which combines a flexible deferred shading pipeline and an extensible lightweight scene graph. The main contributions of our system are a flexible multi-stage material concept, an extensible design for a deferred rendering system and its versatile multi-pass capabilities. In our concept, a material is defined by a shading model and a user-

- *Simon Schneegans is at Bauhaus-Universität Weimar. E-mail: simon.schneegans@uni-weimar.de.*
- *Felix Lauer is at Bauhaus-Universität Weimar. E-mail: felix.lauer@uni-weimar.de.*
- *Andreas-C. Bernstein is at Bauhaus-Universität Weimar. E-mail: andreas.bernstein@uni-weimar.de.*
- *Andre Schollmeyer is at Bauhaus-Universität Weimar. E-mail: andre.schollmeyer@uni-weimar.de.*
- *Bernd Fröhlich is at Bauhaus-Universität Weimar. E-mail: bernd.froehlich@uni-weimar.de.*

Manuscript received 12 September 2013; accepted 10 January 2014; posted online 29 March 2014; mailed on 1 May 2014.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

defined configuration. A shading model consists of multiple shading stages. In contrast to other deferred rendering approaches, all shading stages remain programmable. This is enabled by automatically generating the layout for the intermediate image buffers according to the materials in use. The linkage between a material and user-defined geometry types is achieved using metaprogramming techniques. In guacamole, a pipeline encapsulates the deferred shading process. The output of a pipeline is a rendered (stereo-)image which can be displayed or serve as input to another pipeline. The extensibility of our design is demonstrated by the integration of direct rendering support for trimmed non-uniform rational B-spline (NURBS) surfaces as well as volume ray-casting. guacamole is platform-independent and available as open source [1].

2 Related Work

Many modern rendering systems are based on the deferred shading approach proposed by Saito and Takahashi [20]. In this concept, the geometric properties of the projected surface are stored in an off-screen render target (G-buffer) to defer shading computations for each pixel. While our system also follows the general idea of deferred shading, it differs from existing approaches in various aspects.

In particular, there are a variety of 3D engines such as Unity 3D [7], Unigine [23] or CryEngine [15] [16]. The deferred lighting path in the game engine Unity 3D is a three-pass rendering approach. In the first pass, a minimal G-buffer is used to gather all information necessary for lighting. In the second pass, all light sources are rendered for lighting computations. After the lighting pass, all objects are rendered again for final shading. However, rendering twice is not an option for us because most of our applications deal with very large models. The game engines Unigine and CryEngine are both equipped with numerous state-of-the-art rendering techniques. In contrast to our approach, both engines are based on a fixed G-buffer layout. While the visual quality is very appealing, their extensibility seems unclear because both implementations are closed source.

In our multi-pass concept, the lighting computations are processed in a separate pass, as suggested by [9]. In addition, we adapt some of the ideas of inferred lighting as proposed by Kircher and Lawrance [11]. In their approach, a full-screen quadrilateral polygon is rendered for each non-shadow casting light source in the scene. For each pixel, a lighting shader is executed and its results are stored in a low-resolution lighting buffer (L-buffer). As the screen-space influence of each light source is limited, scissoring or depth-stencil techniques are proposed to optimize this pass. In contrast to their approach, we utilize proxy geometries of the light sources for optimization.

McCool et al. [14] propose a C++-framework for metaprogramming of GPUs. In their approach, the shader programs and memory bindings necessary to perform general purpose computations are automatically generated. While their implementation provides a generic framework for parallel computations, we require shader meta-programming techniques for shader code generation. Foley and Hanrahan [8] presented the shading language Spark in which logical shading features are encapsulated in classes. When features are composed, the necessary shader code, including the inter-stage communication, is generated by the Spark compiler. In contrast to their approach, the logical stages in our shading concept are in different passes. This requires the generation of shader code and buffers for inter-pass communication.

In our research, we use the software framework Avango [12], a revised version of Avocado [26], for the rapid development of prototypical VR-applications. Avango employs OpenSceneGraph [6] for rendering. While the disadvantages of such forward-rendering approaches have already been discussed, we additionally found that OpenSceneGraph has performance issues with respect to scalability and multi-GPU rendering. Consequently, we integrated guacamole in Avango as a replacement for the OpenSceneGraph rendering module. All other modules either do not depend on the rendering or could easily be adapted which allows us to use existing capabilities such as distribution support and interactive Python scripting.

[1]Download guacamole from `https://github.com/vrsys`.

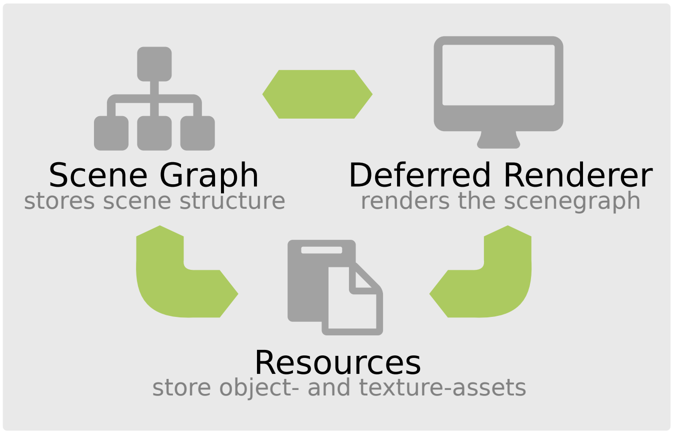


Fig. 2: There are three main components in guacamole which are as loosely coupled as possible. Resources are referenced in the scene graph and accessed by the renderer. The renderer receives a serialized copy of the scene graph in each frame.

```
#create scene graph
graph = SceneGraph()

#add nodes
graph["/"].addChild(Node("car"))
graph["/car"].addChild(Node("wheel"))

#translate node
graph["/car/wheel"].translate(13, 3, 7)
```

Fig. 4: Simple pseudo code of adding, addressing and manipulating nodes using UNIX-style paths.

3 Design

In this section, we present the main structure and components of our system and focus on the most distinctive features with regard to existing works. The three main components of our system are the scene graph, a deferred renderer and the resource management, as shown in Figure 2.

3.1 Scene Graph

One of guacamole's main components is a lightweight, yet flexible scene graph which is used to set up transformation hierarchies. There are several pre-implemented node types such as dedicated nodes for light sources, nodes which instantiate geometry data in the scene or nodes to describe views into the virtual environment. In addition, the scene graph can be easily extended by new node types.

Our scene graph is lightweight in terms of a small memory footprint related to rendering-relevant resources, also referred to as assets. In complex three-dimensional scenes, geometry and texture data may need large amounts of memory. Therefore, nodes representing graphical objects simply hold a reference to an entry in a dedicated database where the actual asset data is stored. Note that nodes are used to instantiate asset data in the scene.

For the convenience of designing a virtual environment, we provide a dedicated class for a scene graph. It stores a pointer to a hierarchy's root node. The pseudo code in Figure 4 shows an example of its usage. Our system supports using multiple instances of this class, each of them representing a virtual scene. This concept simplifies the task to manage multiple virtual scenes in a single application. It is possible to simultaneously show different views of these scenes on multiple displays. Furthermore, the developer may implement scene-in-scene techniques such as world-in-miniature [24] or through-the-lens [25]. A realization of the latter is shown in Figure 7a.

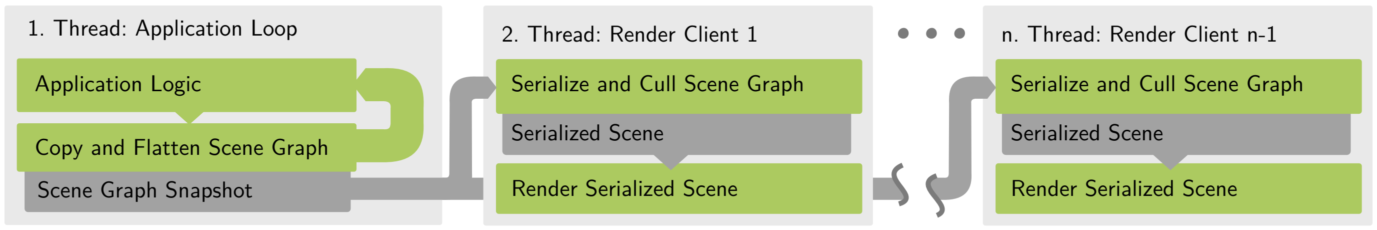

1. Thread: Application Loop
Application Logic
Copy and Flatten Scene Graph
Scene Graph Snapshot
2. Thread: Render Client 1
Serialize and Cull Scene Graph
Serialized Scene
Render Serialized Scene
n. Thread: Render Client n-1
Serialize and Cull Scene Graph
Serialized Scene
Render Serialized Scene

Fig. 3: In guacamole, rendering data is processed in multiple threads. The application's main loop handles user-defined actions, input data and scene graph manipulation. Render clients are assigned to each GPU and informed of application-side scene graph updates. Therefore, a copy of the current graph is generated, whereas the transformation hierarchy is resolved to absolute transforms. Each render client maintains its own thread and performs serialization, culling and rendering of the updated scene graph.

As a basic feature, each of guacamole's nodes stores a transformation matrix, a name string and is able to hold pointers to child nodes. This is quite unusual and a major difference compared to existing scene graph implementations [3]. As a result, an explicit distinction between leaf nodes (nodes without children) and inner nodes (nodes which may have children) does not exist. Furthermore, each node may have a spatial representation in the scene because each stores its own transformation matrix. We believe that this design decision offers a more intuitive hierarchy construction for the developer and reduces the scene graph's depth and, consequently, traversal overhead.

In many virtual reality scenarios, it is useful to restrict the rendering to specific nodes in the scene. For example, in multi-user scenarios, each user may see private information which is not shared with others[1]. Therefore, we added a group concept to our scene graph. In guacamole, each node can be assigned to different groups. In order to define which of the groups have to be drawn, it is possible to specify a group mask for rendering which may combine multiple groups with logical operators. Our renderer is capable of checking the group mask against the nodes' groups. For example, a node with the group *debugGeometry* could be excluded from the rendering by using the group mask *!debugGeometry*. A similar concept with bit masks is found in most common scene graph implementations [6][19]. With this string based appoach we force the application developer to write more expressive code.

Furthermore, our scene graph class provides a convenience interface which allows accessing scene graph nodes via string labels. These strings represent UNIX-style paths in the scene graph and are used to fetch nodes with a known name in a known hierachy. An example of the usage can be seen in Figure 4.

3.2 Resources

As mentioned in section 3.1, resources such as geometries or textures are strictly separated from the scene graph's structure. Instead of holding these assets in the scene's hierarchy, we provide databases for storing them. Connecting resources and scene graphs is possible through references to database entries. The unique keys to the databases are stored in the corresponding nodes. Thus, several nodes may reference the very same rendering resources without storing the actual data multiple times. The resources are accessed by the renderer after the serialization and culling phase as described in section 3.3.

The classes used for databases and assets provide automatic multi-context uploading and handling. This behavior is required because the rendering system is capable of working on multiple GPUs in parallel. Each of the render clients manages its own OpenGL context and the corresponding graphics card memory.

3.3 Rendering

In guacamole, viewing setups are expressed via camera abstractions. A camera refers to screen and eye nodes in the scene graph and thereby defines a certain viewing frustum. Furthermore, a camera specifies the scene graph instance to be rendered and may contain a group mask as mentioned in section 3.1 to restrict the rendering to parts of the scene graph.

In a virtual environment, smooth interaction requires high update rates of the application. Thus, guacamole's architecture relies on multiple threads as illustrated by Figure 3. The main loop, which is responsible for application logic, scene graph manipulation and input processing, is run in a single thread. However, viewing setups that are rendered on different display devices require that the rendering works in parallel. Therefore, a render client is assigned to each device which maintains its own thread and OpenGL context.

At the end of each application frame, an immutable copy of the scene graph is created. During copying, world space transformations for each node are created and their bounding volumes are updated accordingly. Then, a pointer to the copy is passed to each render client. Since the copied scene graph is immutable, all render clients can process it without requiring any synchronization between the different rendering threads. After the last thread finishes, the data structure will be freed automatically.

The actual rendering is done by the render clients in their respective threads. Each of them traverses the scene graph structure and serializes it while simultaneously culling against its particular viewing frustum. The serialized version of the graph works with pointers to the leaf nodes of the immutable scene-graph copy. After serialization and culling are finished, the rendering of the remaining nodes is triggered and guacamole's deferred renderer processes the data.

This architecture allows independent frame rates for application and rendering. Thus, interaction and scene graph manipulation are not slowed down if the rendering speed is reduced.

Occasionally, the rendering frame rate will drop below the application frame rate which is usually limited to about 60 Hz. In this case updated scene graph copies are completely ignored by the render clients if the actual rendering is still busy. Possible enhancements of this behavior are described in chapter 7.

4 Deferred Shading

In the following sections, we would like to present the design of our rendering concept. As already mentioned, the renderer of guacamole is based on deferred shading. In this approach, the actual shading is decoupled from the scene's projection to the screen. First, all surfaces are projected into screen space and rendered without performing any shading computations. Rendering may be directly performed by the rasterization hardware or in a shader, e.g. for implementing per-pixel ray casting. Instead, all the information that is necessary for shading is written into an off-screen render target, which is also referred to as geometry buffer or simply G-buffer [20]. Finally, this image-based data is used to shade each pixel.

The main advantage of this approach is that computationally expensive shading operations have to be done only once per pixel instead of once per fragment. Therefore, it is possible to create extensively shaded scenes with numerous light sources and a high depth complexity while still maintaining interactive frame rates. An example scene with approximately one hundred light sources is shown in Figure 7c.

4.1 Pipeline Concept

In guacamole, deferred shading is performed in a *pipeline*. The input into a pipeline is a scene graph, the corresponding resources in

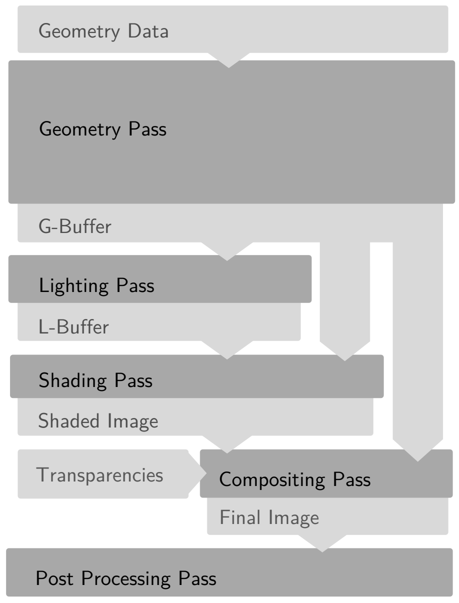


Fig. 5: The deferred renderer of guacamole is split into five passes. The geometry pass projects any kind of geometry data into screen space and stores all relevant data in the G-buffer. The lighting pass accumulates incoming light per pixel. This information is used by the shading pass which performs the actual shading of the pixels. The optional compositing pass may blend transparent objects with the result of the shading pass while considering the G-buffer's depth information. Finally, post processing effects are applied in screen space.

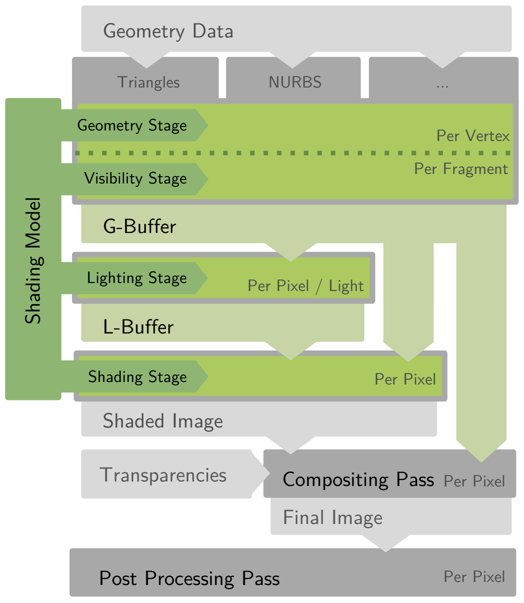


Fig. 6: The shading models of guacamole's materials may influence several passes of the deferred shading pipeline. At run time, the shader code of all user-defined shading models is automatically compiled to shader programs which we refer to as uber-shaders. Based on the material ID, which is stored per pixel in the G-buffer, the appropriate shader code gets executed. Using the four programmable stages of a shading model, complex materials and effects can be defined.

the databases and a camera. In general, the output of a pipeline is a rendered (stereo-)image. If the pipeline is assigned to an output window, it displays the final image. This output window defines on which graphics device the rendering is performed. While a detailed description of this pipeline is given later in this section, we would like to point out a major strength of our system: the capability of easily combining pipelines for *multi-pipeline rendering*.

In particular, it is possible to define a dependency graph between multiple pipelines in order to define their execution order. Furthermore, a pipeline's final image can be used as texture input for a material. Thus, multiple pipelines can be combined to achieve advanced rendering effects such as virtual portals, as shown in Figure 7a, or the screen-space refractions shown in Figure 7b.

A pipeline is divided into five *passes*, as shown in Figure 5. Each pass uses the output of previous passes and computes new per-pixel information for subsequent passes. The following sections describe the purpose of each of the five passes.

4.1.1 Geometry Pass

The main task of this pass is to project any kind of geometry to screen space and render the geometry. For each geometric representations, a type-dependent rendering method is used. The required resources are bound and the respective shader programs are executed. For each pixel, all data which is relevant for shading such as depth value, material index, surface normal and additional, material-dependent data is stored in the G-buffer. Therefore, a dedicated interface consisting of several built-in variables and functions is used.

For non-polygonal representations such as heightfields, solids, parametric surfaces, point clouds or even 3D-video, an advanced rendering method may be required. However, the provided G-buffer interface is the only connection between the rendering method and the actual shading process. This design allows us to integrate any rendering approach, whether it is based on ray casting or a multi-pass method. In section 5.2, we demonstrate this feature by extending our system with a rendering approach for parametric surfaces.

4.1.2 Lighting Pass

In this pass, lighting is computed for each light source for each pixel. The result is stored in the L-buffer, whose layout is entirely dependent on the employed materials. For example, if all materials used in the scene are based on Phong lighting, it will contain information on diffuse and specular lighting per pixel. The following steps are performed for each light source:

1. *Shadow map generation:* If the light source casts shadows, shadow maps are generated. Our implementation currently supports shadows for spot lights only. A geometry pass is executed with a viewing frustum corresponding to the light's cone. The resulting depth buffer is used as a shadow map in the subsequent steps.

2. *Fragment generation:* Fragments are generated for each lit pixel by rendering a proxy geometry. The proxy geometry depends on the light source's type. Spheres are used for point lights, cones for spot lights and full screen rectangles for sunlight.

3. *Shading:* Based on the information in the G-buffer and the optional shadow map, the amount of incoming light is calculated and accumulated per pixel.

4.1.3 Shading Pass

The shading pass processes lighting and geometry information from the L-buffer and G-buffer. It combines them with material-dependent operations. Its output is an intermediate pixel color which is passed to the compositing pass.

4.1.4 Compositing Pass

The compositing pass uses the color and depth information from the G-buffer to perform blending with transparent objects. The result is stored in the same output texture used for the shading pass. The depth value in the G-buffer remains unchanged because it is not possible to composite an appropriate depth value from multiple transparent contributions. After compositing, we perform the post-processing pass.

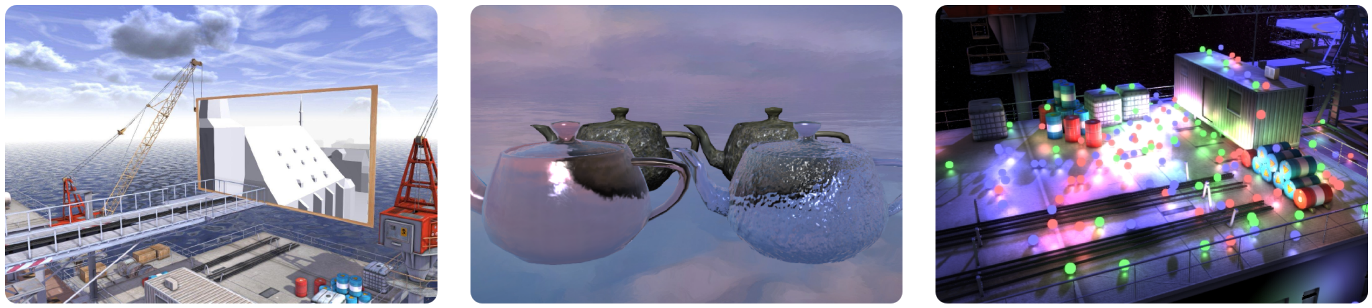

(a) (b) (c)

Fig. 7: In image (a), the combination of two pipelines is used to create a portal which shows a view into another virtual scene. The first pipeline is rendered to a texture which is then used by the second pipeline. Image (b) shows the use of multiple pipelines for advanced material effects. In this example, the illusion of refraction is achieved by using a pipeline's result as input for the glass material. The very large amount of small light sources shown in image (c) demonstrates one advantage of deferred lighting.

4.1.5 Post-Processing Pass

This final pass may use all data of the previous passes in order to create full screen post-processing effects. For now, we implemented FXAA [10], high dynamic range tone-mapping, volumetric light, bloom, screen space ambient occlusion (SSAO) [5] [22], fog and a vignette effect. This set of effects is easily extensible. Two examples of some of these effects are shown in Figure 1. These effects can be enabled, disabled and configured by the application developer but are executed in a fixed order. In future versions, this will be configurable, as well.

Be mindful that, in the compositing pass, no other information except the pixel color was changed. This may affect the results of this pass. In particular, for some post-processing effects such as volumetric light or SSAO, a valid depth information is required. If the depth and color information are inconsistent, the results of these methods are undefined and may contain visual artifacts. This is a limitation in our current design. However, in a future version, we will solve this problem by splitting this pass into two separate stages. Thereby, it will be possible to perform compositing between these stages; the first stage applies all screen-space effects based on geometry, the second those based on color only.

4.2 Material Concept

The main motivation for the design of the material concept was to provide the maximum flexibility for influencing the deferred shading process. As described in section 4.1, our rendering pipeline accumulates the output of five passes. The first three perform on individual object information, the latter two mainly on the shaded image. In our material system, all object-dependent passes remain programmable, which is a major difference to existing deferred shading implementations.

Thereby, a material is designed such that it treats all objects the same, independent of the underlying geometric representation. For example, a developer may assign the same material to a triangular mesh or a parametric description.

In guacamole, materials consist of two parts – a shading model and a material description. A *shading model* describes the operations and parameters used to define the appearance of an object's surface. A *material description* contains a reference to a certain shading model and the parameter values to configure the shading operations. Multiple material descriptions may thereby refer to the same shading model, providing different values, such as textures, colors etc.

A shading model consists of four different, programmable stages as shown in Figure 6. Each of them contains the shader code for the desired computations and specifies the output variables which are passed to subsequent stages. The input to these stages also defines the material description which typically consists of a set of parameters, e.g. textures, color coefficients or constants for functional shading.

4.2.1 Geometry Stage

The geometry stage offers the developer to alter the object's appearance based on a geometric level. The shader is executed in a per-vertex manner on the previously processed geometry data. The input is the same vertex information for each geometry type and is therefore crucial for material-data independence. This is enabled by providing the shader interface mentioned in section 4.1.1. An application developer can influence a vertex' position and normal or may even create new vertices in the geometry shader. Rendering techniques such as displacement mapping may be applied in this stage. Additionally, any user-defined output depending on vertex information may be passed to the following stages.

4.2.2 Visibility Stage

The visibility stage provides the developer with the possibility to directly influence the data which is passed to the G-buffer. It performs on the fragments generated by the rasterization and may use the user-defined per-vertex output of the previously run geometry stage. The developer may alter geometric properties, e.g. surface normal and depth, before they are stored in the G-buffer or even discard fragments to enable see-through techniques [2]. Typical examples of the usage of this stage would be normal or parallax mapping. Note again that any additionally user-defined output depending on an object's fragments may be generated and passed to the following stages.

After the geometry stage and the visibility stage are completed, all generated information is written to the G-buffer. The two remaining stages perform on the per-pixel information stored in the G-buffer, i.e. only the visible parts of the objects are actually being shaded.

4.2.3 Lighting Stage

The lighting stage is used to accumulate all light incident to a certain pixel. Our system provides the distance and direction to and the color and intensity of every light source in the scene. Additionally, shadow maps for shadow-casting lights are generated and combined with the light intensity. In the lighting stage, a developer has access to all data stored in the G-buffer and the parameters of the light source. A major use case for this stage is the implementation of an individual lighting model such as anisotropic or Phong shading. The output of the lighting stage is written to the L-buffer. The layout for this buffer is inferred automatically, as described in section 4.3. For Phong shading, the diffuse and specular intensity would be stored in separate layers.

4.2.4 Shading Stage

The shading stage is used to combine the lighting information from the L-buffer with object information from the G-buffer. While the previous lighting stage is intended to compute intensities only, the shading

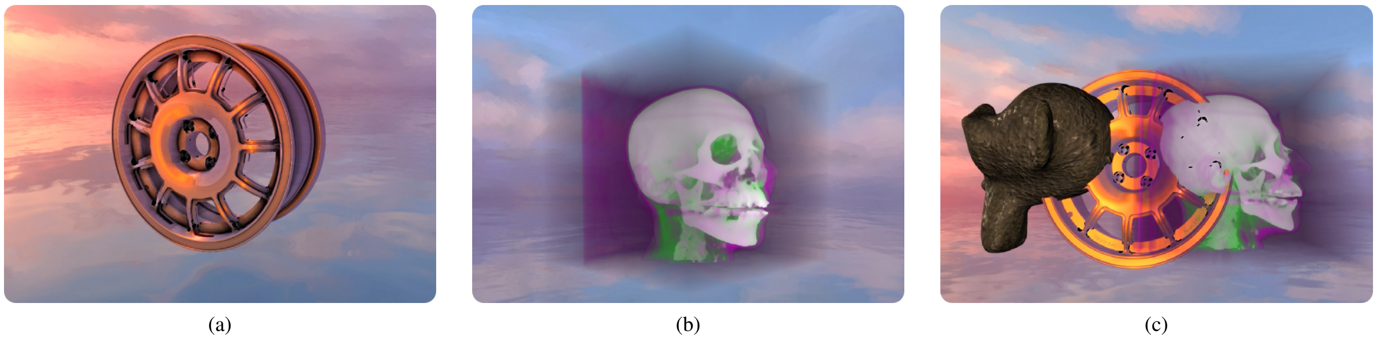


Fig. 8: These images show some results of the rendering methods we integrated into our system. The rim model rendered in image (a) consists of trimmed NURBS surfaces. Image (b) shows the result of the integrated volume ray-casting. The composition of mesh-based geometry, trimmed NURBS surfaces and volume data is demonstrated by image (c).

stage may add color information. Furthermore, techniques like reflection mapping or advanced computations that utilize image-space ray tracing based on the G-buffer may be applied at this stage.

4.3 Automatic Buffer Concept

When implementing a shading model, a developer may define arbitrary output variables for the different shading stages. It is necessary that all variables are written to the corresponding buffers, in order to provide them to the subsequent stages. Thus, guacamole configures G-buffer and L-buffer layouts automatically and compiles so-called *uber-shaders* at run-time. These uber-shaders contain the shader code for each shading model and the uniform input-mapping for each material. Based on each pixel's material ID stored in the G-buffer, the appropriate shader code is executed.

Our system determines an optimal buffer configuration and creates the corresponding uber-shaders automatically. It is possible to pass different data types such as floats, float vectors, booleans or integers between stages. If the number and type of output variables differ between shading models of a virtual scene, one buffer layer will be used by multiple shading models for different purposes. As a consequence, adjacent pixels referring to distinct shading models may store different kinds of information. For example, a diffuse color of one shading model is stored in the same buffer as another shading model's texture coordinates.

Some post-processing effects, such as SSAO or fog, are in need of information in addition to the final color value. Therefore, a few layers of the G-buffer and L-buffer have to be predefined. An example of a buffer configuration is visualized in Figure 10.

The G-buffer contains a 24 bit depth buffer which is also used to reconstruct the fragment's position in world space. Furthermore, a 16 bit unsigned integer stores the ID of the material for the pixel. Finally, the surface normal in world space is stored in a 16 bit RGB buffer. All other layers in the G-buffer depend on the employed shading models.

The L-buffer contains no predefined buffers. In a virtual scene containing only materials which are not affected by light sources, no L-buffer will be generated. Typically, each shading model will store diffuse and specular lighting information.

5 Extensibility

The main motivation for the multi-pass design of our system is the integration of advanced rendering techniques. In this chapter, we would like to present some of the extensions we have already integrated into the system. These extensions relate to following parts of our system: material concept, geometry pass and compositing pass. In section 5.1, we show how materials that rely on multi-pass rendering can be configured easily. Afterwards, we describe an extension which adds direct rendering support for trimmed NURBS surfaces in the geometry pass (see section 5.2) and in section 5.3, we describe how the composite pass is used to integrate volume ray-casting.

5.1 Materials

The material concept of guacamole allows great extensibility on the application side. As mentioned in section 4.2, the developer may influence an object's appearance on the vertex and fragment level. Furthermore, multiple pipelines may be concatenated for the rendering process. The final image of each pipeline can be accessed by the materials. Thus, the material system can be used to implement advanced effects such as screen space local reflections or screen-space refractions (as shown in Figure 7b) based on the previous frame's final image or another pipeline's output.

As another example, multi-user see-through techniques [2] may be applied by appropriately configuring the visibility stage. For that purpose, the world position of the object not to be occluded is specified as input of the shading model. This position is updated in the application's main loop each frame. In the visibility stage, all fragments that are between the object's position and the viewing position are discarded in a fixed or configurable cone.

The extensibility of guacamole's material system enables the developer to implement many other advanced visual effects. In order to support and simplify the material development, we provide a lightweight editor written in C++ with GTKMM [2]. Figure 9 shows the graphical user interface of the material editor.

5.2 Direct Rendering of Trimmed NURBS

In our current implementation, we added direct rendering support for trimmed NURBS surfaces. This parametric representation is typically used in the CAD industry, for details see [18]. While there are numerous approaches based on either tessellation or ray casting [17], we propose a hybrid approach which combines the advantages of both techniques.

The challenge of rendering trimmed NURBS can be divided into two major tasks: projecting the surface to the screen and trimming the surface with regard to its domain. For the projection, we utilize the tessellation capabilities of current graphics hardware. However, our adaptive tessellation scheme is implemented as a two-pass algorithm, as shown in Figure 11, because of the limited vertex output of the tessellation control shader.

In the first pass, a quad patch is rendered for each surface in the scene. We approximate the tessellation level which is necessary to guarantee that all triangular edges are shorter than the tolerated screen-space error. If this level exceeds the hardware limit, we tessellate such that a second tessellation at maximum level would achieve an accurate result. In the other case, we just pass the original patch. In both cases, the resulting patches are passed to the geometry processing and they are stored as a transform feedback. In the second pass, we complete

[2] `http://www.gtkmm.org`, last visited on 01/23/2014

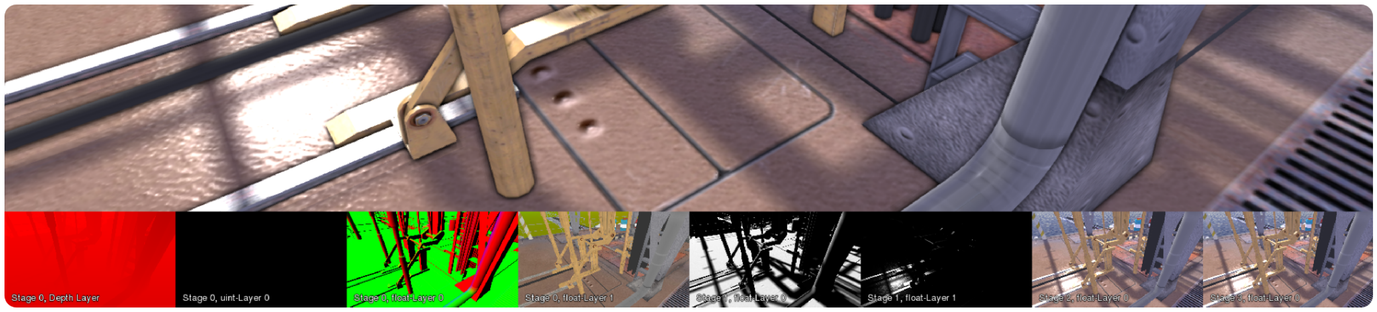


Fig. 10: The automatic buffer configuration produces shading-model dependent buffer layouts for each G-buffer and L-buffer. The picture shows an example configuration, including a G-buffer with (from left to right) a 24 bit/pixel depth layer, a 16 bit/pixel unsigned int layer for material IDs, a 48 bit/pixel float RGB layer for normals and a 48 bit/pixel float RGB layer for diffuse material colors. The L-buffer consists of a 48 bit/pixel float RGB layer for diffuse light colors and a 48 bit/pixel float RGB layer for specular terms as used in standard Phong shading. Additionally, one 48 bit/pixel float RGB buffer is used for combining G-buffer and L-buffer information. The rightmost layer is used by the compositing pass and is identical to the previous one. However, another 48 bit/pixel float RGB layer is produced by the post-processing as the final image (partly visible at the top).

the desired tessellation and apply the trimming during fragment processing. Therefore, we use the direct trimming approach proposed by Schollmeyer and Froehlich [21].

In order to define the binding between geometry pass and shading model, we use a set of built-in variables. These comprise geometric properties such as position and normal. The integration of trimmed NURBS shows that even multi-pass rendering methods can easily be added to our system.

5.3 Volume Ray-Casting

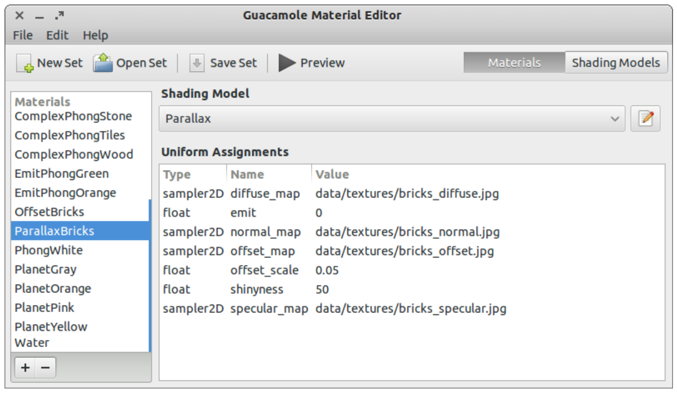
In our pipeline concept, all transparent objects as well as graphics generated by external libraries need to be composited with the intermediate image result in the compositing pass. Thus, we perform volume ray-casting in the compositing pass by a two-pass approach: first, the volume's bounding box is rendered into a depth texture for ray generation. Secondly, ray casting is initiated by rendering a fullscreen quad. All rays are then sampled in front-to-back order until the depth of corresponding pixel in the G-buffer is reached or the ray exits the volume. The accumulated color is written into the G-buffer. Therefore, the color and depth texture of the G-buffer needs to be bound as both input and render target. This is a conservative approach because only one ray is processed per-pixel. This approach obviously also allows for early ray termination. Note that the texture for ray generation is the only additional memory necessary to perform the compositing. All other textures either bound as shader input or render target are an existing part of the G-buffer.

Figure 8 shows the resulting images of our approach. The volume is composited correctly with the existing geometry in the G-buffer. Currently, our system is limited to a single volume because multiple volumes would need to be rendered from front to back for each pixel. We suggest to support multiple volumes by using either a spatial partitioning scheme [13] or depth intervals based on dynamically-linked per-pixel lists [27]. The latter could also be used to store, sort and composite all transparent fragments from the geometry pass. This, however, remains future work.

6 Discussion

In this section, we would like to discuss some design decisions that may be considered controversial. Our discussion also includes some flaws in the current software architecture that we encountered during the implementation. We would like to reflect on them and identify possibilities for future improvements.

The flexibility and extensibility of our system comes at a cost which is mainly due to the memory overhead introduced by deferred shading. In our current implementation, rendering requires many screen-size buffers. As shown in Figure 10, even simple shading models such as Phong shading need several buffers. The configuration in this exam-



Materials
ComplexPhongStone
ComplexPhongTiles
ComplexPhongWood
EmitPhongGreen
EmitPhongOrange
OffsetBricks
ParallaxBricks
PhongWhite
PlanetGray
PlanetOrange
PlanetPink
PlanetYellow
Water

Shading Model: Parallax

Uniform Assignments

Type	Name	Value
sampler2D	diffuse_map	data/textures/bricks_diffuse.jpg
float	emit	0
sampler2D	normal_map	data/textures/bricks_normal.jpg
sampler2D	offset_map	data/textures/bricks_offset.jpg
float	offset_scale	0.05
float	shinyness	50
sampler2D	specular_map	data/textures/bricks_specular.jpg

Fig. 9: This Figure shows our graphical user interface for material development. As indicated by the upper right toggle button, it is currently in material edit mode. In the left box, there are all materials of the project. The right box shows the material description corresponding to the selected shading model. In this example, a parallax-mapping material is configured by setting the appropriate textures and parameters.

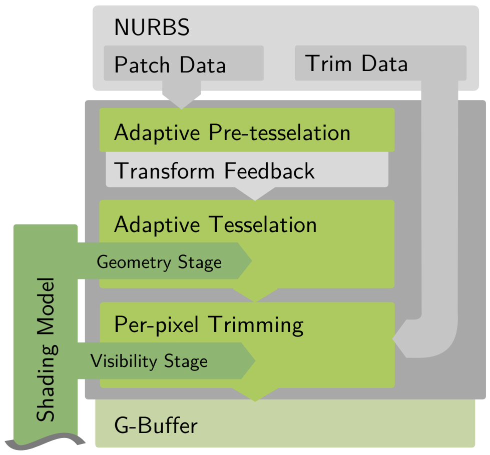


Fig. 11: This Figure illustrates the implementation of the geometry pass for trimmed NURBS surfaces. In our two-pass approach, the routines of the shading model are merged into the shaders for tessellation and trimming using the dedicated built-in interface.

ple uses eight buffers, one of them consuming 24 bit/pixel, one of them consuming 16 bit/pixel and six of them consuming 48 bit/pixel. At a resolution of 1920×1080 pixels, the overall memory allocation for this simple shading model adds up to approximately 81 MiB (stereo 162 MiB). This reduces the amount of memory on the graphics card that can be used by geometry or textures and significantly increases the memory through-put per frame. As a remedy, we consider smaller-sized buffers for different passes as proposed in the inferred lighting approach [11]. However, an optimization step and re-usage of textures could be a better solution to reduce the memory and bandwidth requirements. As another improvement, we may introduce a dedicated thread for scene-graph serialization and culling, which would be an implementation of the classic app-cull-draw pattern [19].

A controversial part of guacamole's interface is the UNIX-path-style to access nodes. While this provides convenience for application developers, it implies a memory overhead since a name string has to be stored in each node. Besides string parsing and comparison is needed to resolve node paths, which introduces computational overhead.

We introduced potential issues for the post-processing pass by using the compositing pass to blend opaque and transparent objects. In particular, all screen-space effects which rely on the per-pixel depth and surface normal may result in visual artifacts. This is because, so far, the depth and normal information remains unchanged in the compositing pass. In order to tackle this issue, the order of compositing and individual post-processing effects should be configurable by the application developer. This would even allow to use the same effects multiple times and with varying configurations.

7 Conclusion and Future Work

We presented a novel software framework for real-time rendering applications. We have shown that the implementation of our multi-pass deferred shading pipeline allows for easy extensibility. The extensibility of the different passes is demonstrated by integrating rendering support for trimmed NURBS surfaces and translucent volume data. Our multi-stage shading model allows for flexible materials which remain programmable in multiple stages. The development of new materials is highly simplified by providing a graphical user interface. The integration of our rendering system into the VR-framework Avango allows us to rapidly develop visually appealing virtual reality applications.

However, many improvements and optimizations still remain. In particular, we seek to support order-independent transparency. Therefore, the G-buffer could be extended with an additional texture storing per-pixel linked lists [27] of all potentially visible transparent fragments. These fragments would then be sorted, shaded and finally composited with all volumes and opaque surfaces in the compositing pass.

Furthermore, we are currently working on the implementation of new geometry types. In particular, we will add a 3D-video node which allows for immersive telepresence applications for groups of distributed users. The input for this node could be a real-time reconstruction from multiple depth cameras similar to that presented by Beck et al. [4]. Finally, it is highly desirable to add level-of-detail approaches for both polygonal meshes and point-based rendering.

References

[1] M. Agrawala, A. C. Beers, I. McDowall, B. Froehlich, M. Bolas, and P. Hanrahan. The two-user responsive workbench: Support for collaboration through individual views of a shared space. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 327–332, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[2] F. Argelaguet, A. Kulik, A. Kunert, C. Andujar, and B. Froehlich. See-through techniques for referential awareness in collaborative virtual reality. *Int. J. Hum.-Comput. Stud.*, 69(6):387–400, June 2011.

[3] A. Bar-Zeev. Scenegraphs: Past, present and future. 2007.

[4] S. Beck, A. Kunert, A. Kulik, and B. Fröhlich. Immersive group-to-group telepresence. *IEEE Transactions on Visualization and Computer Graphics*, 19(4):616–625, 2013.

[5] M. Bunnell. *Dynamic Ambient Occlusion And Indirect Lighting*, pages 223–233. Addison-Wesley, 2005.

[6] D. Burns and R. Osfield. Open scene graph a: Introduction, b: Examples and applications. *Virtual Reality Conference, IEEE*, 2004.

[7] R. H. Creighton. *Unity 3D Game Development by Example Beginner's Guide*. Packt Publishing, 2010.

[8] T. Foley and P. Hanrahan. Spark: modular, composable shaders for graphics hardware. *ACM Trans. Graph.*, 30(4):107, 2011.

[9] R. Geldreich, M. Pritchard, and J. Brooks. Presentation on deferred lighting and shading. In *Game Developers Conference 2004*, 2004.

[10] J. Jimenez, D. Gutierrez, J. Yang, A. Reshetov, P. Demoreuille, T. Berghoff, C. Perthuis, H. Yu, M. McGuire, T. Lottes, H. Malan, E. Persson, D. Andreev, and T. Sousa. Filtering approaches for real-time anti-aliasing. In *ACM SIGGRAPH Courses*, 2011.

[11] S. Kircher and A. Lawrance. Inferred lighting: Fast dynamic lighting and shadows for opaque and translucent objects. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, Sandbox '09, pages 39–45, New York, NY, USA, 2009. ACM.

[12] R. Kuck, J. Wind, K. Riege, and M. Bogen. Improving the avango vr/ar framework: Lessons learned. *In 5th Workshop of the GI-VR/AR Group*, pages 209–220, 2008.

[13] C. Lux and B. Froehlich. Gpu-based ray casting of multiple multi-resolution volume datasets. In *ISVC (2)*, pages 104–116, 2009.

[14] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[15] M. Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, pages 97–121, New York, NY, USA, 2007. ACM.

[16] M. Mittring. A bit more deferred - cryengine 3. In *In Triangle Game Conference*, 2009.

[17] H.-F. Pabst, J. Springer, A. Schollmeyer, R. Lenhardt, C. Lessig, and B. Froehlich. Ray casting of trimmed nurbs surfaces on the gpu. *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 151–160, Sept. 2006.

[18] L. Piegl and W. Tiller. *The NURBS Book*. Monographs in Visual Communication. Springer-Verlag New York, Inc., New York, NY, USA, 2nd. edition, 1997.

[19] J. Rohlf and J. Helman. Iris performer: A high performance multiprocessing toolkit for real-time 3d graphics. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 381–394, New York, NY, USA, 1994. ACM.

[20] T. Saito and T. Takahashi. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, Sept. 1990.

[21] A. Schollmeyer and B. Froehlich. Direct trimming of nurbs surfaces on the gpu. In *ACM SIGGRAPH 2009 papers*, SIGGRAPH '09, pages 47:1–47:9, New York, NY, USA, 2009. ACM.

[22] P. Shanmugam and O. Arikan. Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 73–80, New York, NY, USA, 2007. ACM.

[23] D. Shergin. Unigine engine render: Flexible cross-api technologies. In *ACM SIGGRAPH 2012 Computer Animation Festival*, SIGGRAPH '12, pages 85–85, New York, NY, USA, 2012. ACM.

[24] R. Stoakley, M. J. Conway, and R. Pausch. Virtual reality on a wim: Interactive worlds in miniature. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 265–272, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[25] S. L. Stoev and D. Schmalstieg. Application and taxonomy of through-the-lens techniques. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, VRST '02, pages 57–64, New York, NY, USA, 2002. ACM.

[26] H. Tramberend. Avocado: A distributed virtual reality framework. In *Proceedings of the IEEE Virtual Reality*, VR '99, pages 14–21, Washington, DC, USA, 1999. IEEE Computer Society.

[27] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. Real-time concurrent linked list construction on the gpu. *Computer Graphics Forum*, 29(4):1297–1304, 2010.