Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Graphics 29 (2005) 3–15
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

Multimedia integration into the blue-c API

Martin Naef[a,*], Oliver Staadt[b], Markus Gross[a]

[a]Computer Graphics Laboratory, Swiss Federal Institute of Technology, Zurich, Switzerland
[b]Computer Science Department, University of California, Davis, USA

Abstract

In this article, we present the blue-c application programming interface (API) and discuss some of its performance characteristics. The blue-c API is a software toolkit for media-rich, collaborative, immersive virtual reality applications. It provides easy to use interfaces to all blue-c technology, including immersive projection, live 3D video acquisition and streaming, audio, tracking, and gesture recognition. We emphasize on our performance-optimized 3D video handling and rendering pipeline, which is capable of rendering 3D video inlays consisting of up to 30,000 fragments updated at 10 Hz in real time, enabling remote users to meet inside our virtual environment.

Keywords: Virtual reality software system; 3D video; Multimedia; Collaborative virtual environments; Telepresence

1. Introduction

The *blue-c* system [1] developed at ETH Zurich provides a novel virtual environment which combines immersive projection with 3D acquisition of the user, allowing remotely located users to meet in a virtual world. blue-c enabling technology includes custom hardware [2] and a new real-time video acquisition and transmission approach [3].

This paper discusses the multimedia integration into the *blue-c application programming interface* (API), a software toolkit that provides easy access to all underlying blue-c technology for the application developer. As opposed to other virtual reality (VR) toolkits that try to separate the VR-specific modules from the graphics rendering and scene graph, the blue-c API tries to integrate much of its functionality into the scene. This helps to keep development interfaces and programming patterns consistent throughout the system without extensive code wrapping efforts. Besides providing access to blue-c-specific technology such as real-time 3D video for telepresence, the blue-c API can also be used as a general-purpose VR toolkit outside the blue-c portals.

This paper focuses on 2D video and performance aspects of our 3D video integration. It also briefly introduces the software architecture. Those aspects that are already covered in detail in [1,3,4] will be omitted. After the system overview, the 2D video system is analyzed. Motivated by the rendering algorithms used for 3D video, we then present a performance-optimized pipeline for turning an incoming dynamic 3D video fragment stream into a vertex array suited for high-performance rendering, discussing the various performance vs. quality trade-offs.

The blue-c application programming environment runs on SGI IRIX™, Linux, and Microsoft Windows™ operating systems. The application code is directly portable between the systems. The API itself has some platform-dependent optimizations to use the available hardware to its full potential.

*Corresponding author.
E-mail addresses: mnaef@acm.org (M. Naef), staadt@cs.ucdavis.edu (O. Staadt), grossm@inf.ethz.ch (M. Gross).

0097-8493/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2004.11.003

The remainder of this paper is structured as follows: Section 2 gives an overview of related work. Section 3 presents the system architecture of the blue-c API. Multimedia services are presented in Section 4, the 3D video service is in more detail in Section 5. We conclude with applications in Section 6 and provide an outlook into the future in Section 7.

2. Related work

Systems for VR are always combinations of many different toolkits and software libraries. The blue-c API is no exception in that respect. This section presents a selection of previous work related to the blue-c API.

Numerous VR application development toolkits have been implemented in the past. CAVElib™ development was started with the initial CAVE™ [5] system and is available as a commercial product (www.vrco.com). It supports device input through the *trackd* system. For rendering, it relies on application-supplied OpenGL code or the Performer scene graph system. Basic networking code for clusters and collaboration is provided, but there is no automatically shared scene graph.

Juggler [6] provides a mature, object-oriented approach to VR. It is very actively supported. As opposed to the blue-c API, but similar to CAVElib™, Juggler mostly leaves the choice of the rendering system to the application developer and keeps only loose ties to the scene graph. Juggler implements a kernel that keeps several "manager" objects. This concept inspired the blue-c API service structure.

Avango [7], formerly called Avocado, provides similar functionality as the blue-c API. Avango exposes most interfaces to its own scripting language. It is closely coupled to OpenGL Performer, but it changes the scene graph interface to an Inventor-style field system whereas the blue-c API leaves the Performer interfaces unchanged to better support legacy applications. Avango relies on total network ordering and strict locking, which imposes significantly higher requirements onto the underlying network layer.

There is a plethora of toolkits available that provide parts for VR systems, including tracking libraries [8,9], scene graphs [10], networking tools [11], audio servers, etc. Using them together to build a large VR system such as the blue-c, however, requires to learn many different interfaces and concepts, and finding ways to get them to work together smoothly is not always trivial. For the blue-c system, the aim was to provide a holistic, consistent, and well integrated toolkit that provides strong multimedia and basic collaboration support. Unlike other toolkits that separate the scene graph from the rest of the VR system for more flexibility, the blue-c API integrates it into the core for more coherence, allowing to integrate media handling directly into the scene graph without compromising performance.

3. blue-c API system architecture

This section briefly introduces the system architecture of the blue-c API, as presented in [12]. An overview of all components and their main dependencies is given in Fig. 1.

3.1. Core and process management

The blue-c core class is a small kernel that handles system initialization and startup, instantiation and discovery of services, runs the main application loop, and takes care of a clean system shutdown. With the exception of the scene synchronization system that only spawns network transmission threads, most blue-c services spawn individual calculation processes that communicate through a shared arena that is managed by the Performer scene graph system [13].

All process management and locking methods are encapsulated by the API to provide platform

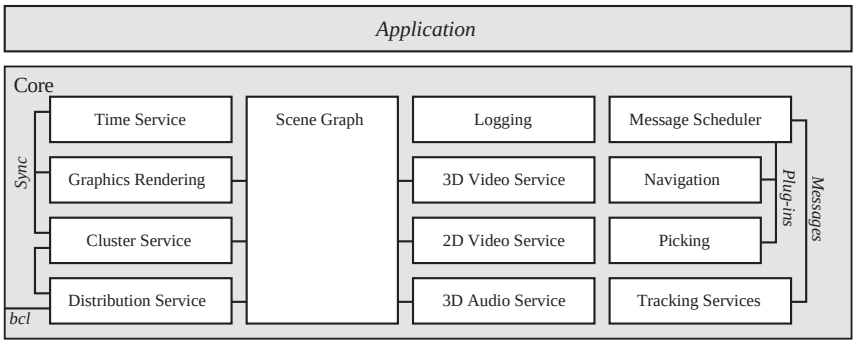

Fig. 1. blue-c API system overview: Services accessing the scene graph, and message scheduler with sources and plug-ins.

compatibility with Windows, which does not provide fork( ) or uslock primitives. The lack of the fork( ) paradigm on Windows imposed no difficulties during porting since none of the services rely on process-local data.

3.2. Services

All blue-c functionality is implemented as a set of services. These include 2D and 3D video, audio, logging, distributed scene synchronization, and simulation time. Graphics rendering including multi-pipe setup is also encapsulated as a service. Cluster support is implemented as a new cluster service that keeps tight bonds to the distribution service used for remote collaboration. It also synchronizes the time service inside the cluster.

3.3. Scene graph

The blue-c scene graph is based on OpenGL Performer [13] and enhanced with serialization and state update interfaces for collaboration as presented in [4]. The same synchronization system is also used for cluster rendering. Additional custom nodes and attribute objects are integrated for special multimedia functionality. They are described in the sections below.

3.4. Messaging

The message passing paradigm for event signalling is a widely accepted pattern for user interface systems (e.g. Win32, X11). It provides simple queueing of events and allows one to easily cross process or even machine borders. The blue-c API uses messages to signal all user interface events, including movement of tracking sensors and mouse, button or keyboard presses. Messages are also generated to signal ownership changes of distributed objects in the scene graph.

3.5. Device I/O

The blue-c API supports 3D motion tracking devices by providing device drivers as services. A base tracking class provides the necessary state information interfaces, reference frame transformations for the sensors, messaging, and support for compound devices such as the Fakespace Wand™. Derived classes implement the low-level device access code for Ascension, Polhemus and Intersense tracking systems.

3.6. Configuration system

The configuration system keeps a tree hierarchy, which corresponds to the services and their sub-devices. It is generated from human-readable text files. All multimedia services read their settings from this tree and create the required source objects during startup.

4. Multimedia performance

This section presents the blue-c multimedia features, their integration into the scene graph, and analyzes the performance.

4.1. 2D video service

The integration of 2D video streams greatly enhances the visual appeal of virtual environments. It can be significantly more efficient, both in terms of modeling time and real-time rendering performance, to create interesting visual effects using video instead of using animated geometry. Typical applications for 2D video include using a video backdrop to create an attractive environment with a low polygon count. The IN:SHOP application prototype [14] almost exclusively relies on this technique for a visually appealing result. Fig. 2 shows the IN:SHOP prototype application with a video background.

The video integration into the blue-c API consists of two major parts, the video service and a texture proxy inside the scene graph. The video service can be considered the ''back-end'' of the system that manages the list of video data sources. Video texture proxies are regular Performer texture objects that are updated by the video service whenever new image data are available.

4.1.1. Video source objects

The video service keeps a video source instance for each individual video object inside the scene. A video source is a generic object which provides the interface for starting and stopping video streams, and uploading the data into texture memory of the graphics hardware. Video sources are identified with a name, which either corresponds to a camera as defined in the video configuration script or to the filename for file sources. The base video source class also provides a simple

Fig. 2. IN:SHOP application prototype with animated 2D video background and 3D video inlay in the foreground.

chroma-keying algorithm to generate an alpha channel for transparent textures. Each video source object spawns a thread or process for asynchronous decoding and transmission of video data. The blue-c API implements two types of video sources: a video file source and live transmission.

Video files sources provide access to pre-recorded video sequences. They are capable of reading and decoding standard video files such as Windows AVI, MPEG or Quicktime. For video file decoding, we use the SGI digital media library on the Irix platform, libavcodec and libavformat libraries that are part of the ffmpeg (http://ffmpeg.sourceforge.net) distribution on Linux, and DirectX 9 direct show graphs on Windows.

Live video streaming is supported as an additional video source class. The video source object connects to a camera server using the blue-c communication layer or a plain TCP connection. The camera server is implemented as part of the acquisition system, which is used for 3D user acquisition, or as a standalone application. In the first case, the video image is transmitted as a sequence of image operators, resulting in a compressed, progressive transmission. Transparency information is derived as a side effect since only pixels classified as "foreground" are actually transmitted. This live video stream visualizes part of the 3D video fragment processing pipeline [3] and serves mostly as a demo showcase for the blue-c acquisition technology.

All video source types copy the acquired image into a back buffer provided by the base source implementation. After the acquisition step, this buffer is swapped with the active buffer, therefore enabling concurrent acquisition/decoding and uploading into texture memory. The video service registers a callback method for the draw process, which updates the texture object only if the front buffer has updated content. This avoids unnecessary texture uploads. The double buffering scheme completely decouples the video streaming frame rate from the graphics rendering system, allowing the graphics system to render at full speed.

4.1.2. Texture proxy

Video textures are represented as regular Performer texture objects inside the scene graph. They support all texture parameters including blending, wrapping modes, etc. It is possible to attach a video source to any texture object, automatically replacing its content.

The blue-c API provides its own texture class as part of the distributed scene graph, which is derived from the standard Performer texture class and inherits all functionality. Besides support for distribution, it has been enhanced with a special file load method that provides direct support for 2D video. Instead of providing an image file name, a "video": prefix can be added to the name to specify a video source. In this case, the API first tries to locate an existing video source object with the given name and connects it to the texture if it is found. If no corresponding video source is found, a new video file source is created.

If a video texture is used as part of the shared partition of the scene, only the source name is synchronized. 2D video data is not transmitted between the different sites, it is the developer's responsibility to make sure the referenced video files are available at all participating sites. The same applies for cluster nodes.

4.1.3. Performance

The performance of the 2D video system mostly depends on the libraries used and the underlying hardware. Since each video stream object uses its own process for video decoding, the video system can be accelerated significantly by using multiprocessor hardware. On the SGI Onyx with eight processors, several video streams can be decoded and uploaded to texture memory simultaneously without a significant drop in the rendering frame rate. Applications however should only enable video sources that are actually visible to conserve processing power.

Table 1 lists video performance measurements, including video decoding time for 256×256 pixel high-quality MPEG-1 with 2.3 Mbit/s, 256×256 pixel AVI with a Cinepak codec and a data rate of approximately 1.7 Mbit/s. The Cinepak decoding is significantly faster than MPEG-1. A single processor could be used to decode 71 Cinepak frames per second, or almost three video files at 25 frames/s. The table also includes the texture update time for each frame. The short value does not reflect the true texture upload bandwidth as the system caches the data before it is actually sent to the pipeline. Using nine animated textures in the IN:SHOP example application resulted in no measurable drop in the framerate, the video decoding process is clearly the bottleneck with these small video files. All measurement were done on the Onyx 3200.

4.2. 3D audio

The blue-c API provides a high-quality spatialized 3D audio system that runs as a service, in addition to visual

Table 1
2D video decoding and texture uploading performance. Measured with 256×256 video on SGI Onyx 3200 with 400 MHz MIPS R12000 processors

Task	Time (ms)
Decoding MPEG-1 frame	70
Decoding AVI frame (Cinepak codec)	14
Texture update	0.04

output. It is controlled by active audio nodes from inside the scene graph, which allows to add sound as attribute to geometry objects. Each audio node controls a sound source object in the audio renderer. Its position is updated once per frame following the underlying transformation nodes and the virtual to real world coordinate system transformation that is provided by the graphics rendering system. We refer to [15] for a detailed description of the blue-c audio rendering pipeline, including performance measurements. As opposed to the original implementation, the current version now uses the Portaudio libraries for all device access, which enables trivial portability between the different operating systems.

4.3. Animation

The blue-c API supports animation of 3D geometry using the concept of *animation nodes*. These are transformation nodes that update their transformation matrix for every frame through a virtual method that can be overridden by the application developer. As a starting point, the blue-c API provides a set of default animation nodes implementing rotation and key-frame animation that provides linear interpolation for both position and rotation.

A more complex animation node supports importing animated figures created with Curious Labs' Poser (http://www.curiouslabs.com). A customized file loader reconstructs the model hierarchy from exported geometry files and parses the respective BVH motion data. This allows for a quick population of virtual worlds with moving characters.

5. 3D video fragment rendering

The support for 3D video acquisition and streaming of users is a key feature of the blue-c system. A 3D representation of the user standing inside our portal is acquired concurrently with the immersive stereo projection. Using a shape from silhouette method, the user is reconstructed as a cloud of 3D video fragments, which is a generalization of the pixel concept into three space. These fragments are then encoded and transmitted incrementally across the network to the other participating sites. For details on the 3D video acquisition, processing and transmission pipeline we refer to [1,3]. The following discussion only refers to the receiver side, including rendering. We will first introduce the different options for rendering the 3D video objects, which motivate the design of the data processing pipeline that provides the necessary data structures for efficient rendering.

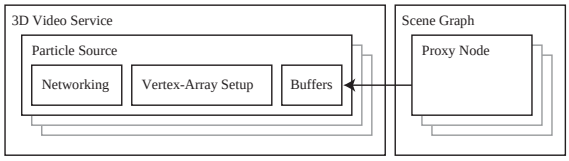

Fig. 3. 3D video service with video fragment sources and proxy object inside the scene graph. The buffers are used to pass data between the processes.

5.1. 3D video service

The 3D video integration into the blue-c API is built upon the 3D video service, which hosts fragment data source objects. Proxy objects inside the scene graph refer the data source objects and provide callback-hooks for rendering in a multiprocessor-safe manner. Fig. 3 depicts the general architecture.

5.1.1. Service

The 3D video service hosts a list of fragment data sources. The service instantiates the dynamic streaming sources according to the information provided by the setup system, or on demand for file-based sources. The source objects can be accessed by name or ID, the necessary enumeration methods are provided by the service.

Typical applications do not directly access any of the methods provided by the 3D video service. Instead, the 3D video proxy objects in the scene graph initiate all necessary connections.

5.1.2. 3D video data source

The 3D video data source objects host virtually all functionality for processing and rendering incoming data. There are two different types of data source classes, both derived from a common class CBCInParticleSource that implements basic multiprocessing synchronization, job management, and defines the callback interface.

For static point objects, the CBCInParticleSourceStatic class retrieves a single frame from a Pointshop 3D file [16]. It retains all settings including point size, normal and color. The blue-c API supports any number of static sources. They are typically created dynamically by the user application by specifying the input filename for a proxy object.

Dynamic video streams are provided by the CBCInParticleDynamic class. Dynamic particle sources are pre-defined in the configuration scripts, including settings such as rendering modes, as well as whether they are streamed from a file or from a live source. Configuration of the streaming channel, namely the source, is defined through the blue-c communication

layer (bcl) [17] configuration and does not influence the pipeline further down.

5.1.3. 3D video proxy node

The 3D video proxy node class represents a 3D video object inside the scene graph. The proxy node basically keeps a reference to the connected data source and calls its methods during the Performer App, Cull and Draw scene graph traversals.

As a regular Performer node, it can be placed anywhere inside the scene graph and follows all underlying transformations. It provides information about the bounding box for culling. The proxy node concepts makes the addition of 3D video to blue-c applications as easy as adding any other node into the scene.

5.2. Fragment rendering options

The blue-c real-time 3D video system represents its objects using 3D video fragments that essentially form a point sample cloud in 3D space. The rendering of these point samples is based on work and by Pfister et al., Zwicker et al. and Rusinkiewicz and Levoy [18–20]. As opposed to the previous work, the blue-c API rendering implementation is optimized for dynamic data sets for the SGI Onyx 3200 platform as used in the first blue-c portal. This section presents an overview of the different rendering modes and its requirements for the data pipeline. The visual differences of the rendering methods are shown in Fig. 4.

5.2.1. GL_POINT rendering

The simplest fragment rendering technique relies on the OpenGL point primitive. Every fragment is rendered with a single point, defined by a 3D vertex and color. In the blue-c API implementation, all points are rendered with the same size. The size in world space is defined in the configuration script and translated into a screen pixel size before rendering. Since the point size is only correct for a single location, objects that are close to the viewer will exhibit strong artefacts (holes or excessively large points). These artefacts could be reduced by introducing variable point sizes. Variable point size rendering requires a sorting step because the point size is not a per-vertex attribute in standard OpenGL. In practice, however, the relatively small resolution of our objects make close-up viewing unsuitable, regardless of rendering mode. The idea of implementing a more efficient variable size GL_POINT renderer was, therefore, abandoned.

Most OpenGL hardware implementations render the point primitive as a square. The SGI Infinite Reality 3 hardware also supports rendering of round, anti-aliased points. The latter mode, however, is only efficient if multi-sampling is supported by the visual, which is not the case when using quad-buffered stereo visuals.

Using point sprites [21] for rendering splats enables the low data transfer requirements of GL_POINT rendering while providing very high-quality visual results. Unfortunately, these primitives rely on extensions not available on the Infinite Reality graphics hardware and were, therefore, not included in the blue-c API.

5.2.2. Circular splats

Unlike GL_POINTS, which always render a primitive in screen space, rendering discs perpendicular to the fragment normal in object space results in a much better approximation of the underlying surface. These discs are defined by a 3D position defining the center, a size in object space, the fragment normal and a color.

OpenGL does not provide a "circle" primitive. Approximating the geometry with a complex polygon or triangle fan, however, would result in an excessive number of vertices. Instead, the common solution is to render a single quad primitive and use an alpha texture as "stencil", reducing the number of required vertices to four per splat at the expense of a slightly increased

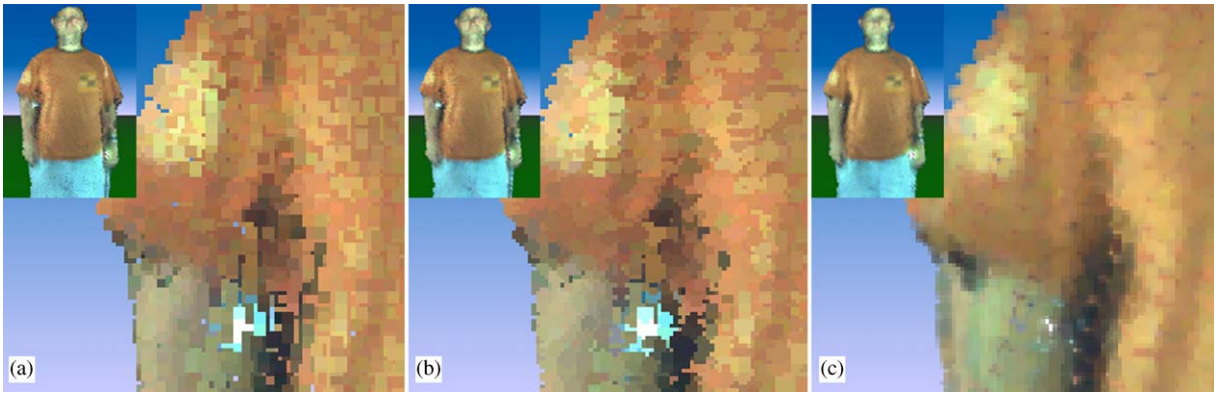

Fig. 4. Comparison of fragment rendering options: (a) points; (b) single pass; and (c) two-pass circular splats.

fill-rate requirement. The texture is a 2D grayscale image of a Gaussian function. Using an alpha compare function (glAlphaFunc(GL_GREATER, etc.)) cuts off the corners of the quad, resulting in more or less circular splats depending on the reference value. Of course, a simple black/white texture as stencil would work just as well, the current implementation was mainly chosen for compatibility reasons to the two-pass renderer. Compared to the point renderer, rendering circular splats increases the required transfer bandwidth by a factor of six, as the texture coordinates (two floats) must be transmitted in addition to color and position per vertex.

The current implementation allows to turn off the texturing and therefore the stencil operation, effectively rendering quadratic splats instead of discs. However, this does not result in increased performance because current hardware supports simple texturing at no additional cost.

Rendering variable sized splats introduces no additional overhead since the size of the splats is defined by the distance of the vertices in object-space. Unlike GL_POINTS that require individual vertex or index arrays to support different sizes, all circular splats can be fed to the graphics pipeline in a single transfer operation.

5.2.3. Two-pass splat rendering

Both previous rendering methods do not take any anti-aliasing or smoothing measures. However, the visual quality of our 3D video objects benefits from a smooth blending of splats instead of rendering circles with sharp edges. Instead of representing surface samples with discs, the surface is defined as the sum of Gaussian field functions with the center defined by the fragment center. An approximation to this interpretation is accomplished with the two-pass splat rendering method as depicted in Fig. 5. As opposed to the original EWA splatting method [19], this implementation does not provide a final alpha normalization step as this operation cannot be implemented efficiently on the graphics hardware available.

In theory, correct blending of the splats could be achieved by rendering from back to front in a single pass. However, the required sorting is not practical considering the large number of fragments. Therefore, a first rendering pass is required to setup the z-buffer for correct occlusion handling.

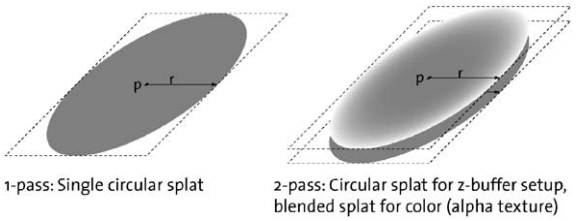

Fig. 5. Splat rendering: single pass and two-pass rendering modes. Circular splats are rendered with a quad primitive and an alpha texture as stencil. The same alpha texture is used for blending. p denotes the fragment position, r the size of the splat.

The first rendering pass is very similar to the single-pass rendering method. The splats are rendered as discs, generated by quads with a Gaussian texture as stencil. Writing to the frame-buffer (color and alpha components only), however, is disabled during that step. Instead, only depth values are written, resulting in a z-buffer which represents the final surface of the fragment object, but leaves the color buffer intact.

In a second rendering step, all splats are rendered again. Now, the splat color values are blended with the framebuffer content according to the alpha value defined by the Gaussian texture. At the center of the splat, the background color is replaced, whereas the corners slowly fade out, resulting in a very smooth and filtered appearance. Z-buffer writing is disabled during the second pass. To make sure neighboring splats without precisely the same depth still ''merge'', the second rendering pass needs to introduce some tolerance in the z-buffer test. This is accomplished with the glPolygonOffset method that offsets the actual depth value by a fixed amount. Pixels that would fail a regular depth test by a small amount, and therefore most probably belong to the ''front'' surface, can still pass after applying the polygon offset, therefore contributing to the overall blended color. Rendering without the polygon offset would results in many semi-transparent holes, and strong artefacts in places where splats intersect.

Defining the polygon offset means deciding on a compromise between correct occlusion culling and hiding artefacts introduced by object reconstruction inaccuracies. It is, therefore, a user-configurable option.

5.2.4. Compositing

All rendering options implemented in the blue-c API make use of standard OpenGL rendering primitives without the need for auxiliary buffers or special rendering order. Since all point rendering primitives use the same coordinate space and setup as the traditional scene geometry, compositing 3D fragment objects into the scene is implemented trivially using the z-buffer in the current rendering context. Performer optimizations such as visibility culling apply equally to 3D fragment objects as they only rely on a bounding box, which is set per node.

5.3. Rendering performance

The performance of the different rendering methods was measured on an SGI Infinite Reality 3 graphics system and on an nVidia Quadro4 750 XGL graphics board. Considering the small size of the individual

Table 2
Rendering time per fragment

Platform	Point (μs)	Circular 1 pass (μs)	Circular 2 pass (μs)
Infinite Reality 3	0.1	0.5	1.3
n Vidia Quadro 4	0.02	0.15	0.3

splats, it was found that rendering time was predominantly a function of the number of vertices sent down the graphics pipeline. The maximum fill-rate has only a minor impact during two-pass rendering on the SGI hardware. Table 2 lists the measured rendering time in microseconds per fragment for both platforms and all three rendering methods. Both hardware platforms achieve close to their theoretical peak performance: 10 million vertices per second on the IR3 and 50 million vertices per second for the nVidia Quadro4. The measured times do not include the setup of the vertex array data structure. This performance property is discussed in Section 5.5.

With the typical 3D video objects in the blue-c environment, the rendering speed is sufficient to keep frame rates in the order of 15–20 Hz in stereo mode with the highest quality renderer on the Onyx. As soon as the scene complexity is increased, a faster renderer should be selected to keep a smooth application flow. Since the rendering is completely decoupled from the vertex array processing pipeline that runs on a different processor, performance can be evaluated separately. On the PC platform, rendering the fragment objects imposes virtually no performance loss. The setup of the vertex arrays, however, becomes more significant due to the small number of available CPUs and the less favorable memory bandwidth to raw processing power ratio.

5.4. Data processing pipeline

The rendering hardware only achieves the peak performance if the data are fed through a packed vertex array structure. This array, however, is significantly different from the actual data structures used for receiving 3D video fragments. Each fragment data source, therefore, implements a pre-processing pipeline that transforms the incoming 3D fragment data structure into a vertex array suitable for concurrent rendering.

This section presents the pre-processing pipeline including a performance analysis. Fig. 6 depicts an overview of the pipeline stages and shows the main data structures used. The performance properties were measured on our SGI Onyx 3200 with 8 MIPS R12000 processors running at 400 MHz and two IR3 graphics pipes. Additional measurements were done on a standard Dell PC with a Pentium 4 2.8 GHz processor and nVidia Quadro4 750 XGL graphics hardware.

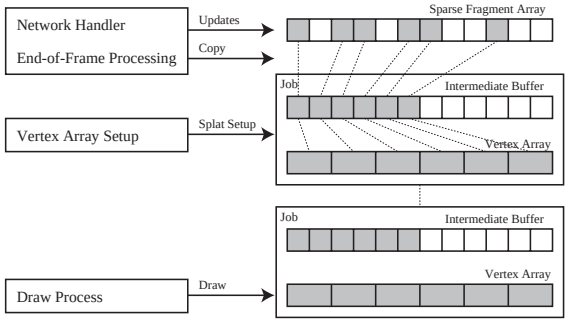

Fig. 6. Vertex array setup pipeline. The network process updates the fragment array and creates the intermediate copy in a job object for further processing. The vertex array setup process generates the vertex array data structure. Once the job object is ready, it is passed to the draw process.

5.4.1. Incoming data

3D video objects are transmitted using fragment operators [3]. They are decoded and used to update a sparse, linear data array incrementally. The position of the entries in this array are defined by a hash function based on the 3D position of the fragment. The incoming data structure is fully provided and handled by the 3D video streaming software modules [17]. The data structure allows for very fast random access to update individual particles. Sequential access, however, suffers from the sparse structure, requiring more memory bandwidth.

Especially on SGI Onyx hardware, the network transmission and decoding process is currently the main bottleneck limiting the number of active fragments. It therefore runs on its own processor, and further processing inside the same process is kept to a minimum.

The incoming data structure does not allow for efficient, concurrent access. The processing of incoming fragment operators and further handling of the array are, therefore, kept as sequential block operations inside the network transmission process. The transformation from the incoming data structure into the vertex array for rendering is triggered once per acquisition frame. With the current acquisition system, this happens roughly ten times per second. Typical representations of remote users consist of approximately 25,000 fragments, leading to a total of 250,000 fragments to be processed per second, or a time budget of 4 μs per fragment. A more detailed analysis of object sizes and update frequency is given in [22].

Each fragment is specified by a 3D position, color, transparency (encodes lifetime), surface normal, and some additional book-keeping flags. The normal is

encoded as two bytes representing quantized spherical coordinates.

5.4.2. Vertex array setup pipeline

For efficient rendering, the point samples should be copied into a linear vertex array which packs position, color and texture coordinates into a data structure that can be read directly by the graphics hardware with a single burst transfer. Due to the dynamic nature of the input data, this array must be set up completely for each incoming frame. The pipeline is depicted in Fig. 6.

Each circular splat (see Section 5.2) is defined by four corner vertices. Each vertex contains the position (3 floats), color (4 bytes), and texture coordinates (2 floats), resulting in a packet data structure of 24 bytes per vertex, or 96 bytes per splat. The normal is not specified as the current acquisition system does not provide data that is precise enough for re-shading of the 3D video object. To enable re-shading, the reconstructed normals must be further processed (e.g., smoothed) to avoid disturbing artefacts.

The relative positions of the corner vertices to the fragment center are defined by **u** and **v** vectors that are efficiently derived from the encoded fragment normal. The necessary angular functions (sine and cosine) are retrieved through a lookup table. Experiments showed that the calculation cost of the corner vertex positions is completely hidden by the memory write bandwidth on both SGI and PC platforms.

If the vertex array is directly derived from the incoming data structure, each splat takes approximately 1 μs on the Onyx, and 0.5–0.8 μs on the PC. This includes the traversal time for the sparse array and assumes a fixed splat size. Given a 3D object with 25,000 fragments updated at 10 Hz, which is typical for the blue-c system, this translates into roughly 25% processing time in the same process that also handles the incoming fragment operators. Experiments showed that increasing this processing time stalls the incoming fragment stream on the SGI platform. As long as only a single processor is dedicated to a fragment source pipeline, the system is therefore limited in the complexity of the object and to rendering using a fixed splat size.

To overcome these processing power limits, an optional pipeline mode first creates a linearized copy of the video fragment data structure instead of setting up the vertex data structure inside the network streaming process. The resulting array only takes 24 bytes per fragment. This copy operation takes approximately 0.8 μs per fragment on the Onyx, leaving about 80% of the processing time to the network process, a different thread is triggered to calculate the vertex array structure from the intermediate copy. Since this thread runs on a separate processor, a time budget of typically 4 μs per fragment is then available, leaving room for a simple

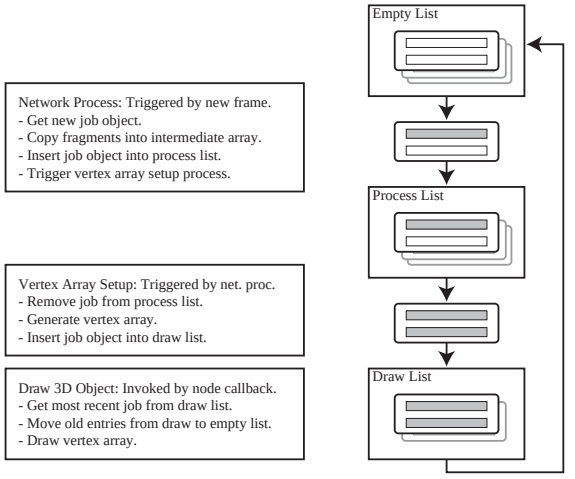

Fig. 7. Job processing in the vertex pipeline. Job objects are stored in lists for processing. The setup process is triggered by the previous stage in the pipeline using a semaphore.

calculation of an individual size per splat as discussed in Section.

Both the intermediate fragment array and the output vertex array are kept in job objects. Job objects are kept in a queue and passed to the worker thread for processing and to the graphics process for rendering. This job processing system, depicted in Fig. 7, enables parallel operation with minimal locking interference. Job queues and objects are locked only at the end of each calculation process. The latest complete vertex array is, therefore, always available for asynchronous reading of the rendering system. After a rendered job has been superseded by a more recent job object, the object is inserted into an empty queue for reuse. This avoids constant allocation and de-allocation of large memory blocks.

Thanks to the high system memory bandwidth of the Onyx architecture, using the additional processor fully delivers the expected speedup. Using multiple buffers, however, adds some latency to the overall processing and transmission pipeline. In practice, the difference between single-processed and parallel handling is not perceivable by the user, even in local-feedback applications such as our 3D mirror.

5.4.3. Splat size calculation

The size of the rendered splats is an important factor for the resulting visual quality of the rendered 3D fragment object. Small splats do not fully cover the surface they represent, leading to visible holes. Large splats with excessive overlap on the other hand result in a blurred appearance, small details are lost.

In the optimal case, neighboring splats have a slight overlap and fully cover the surface after blending. Since

3D fragment objects are not necessarily sampled on a regular grid, the size of each individual splat is a function of the distance to its nearest neighbors. The data structures used in the blue-c 3D video pipeline, however, currently do not allow for fast querying of the nearest neighboring fragments. The blue-c API, therefore, provides several methods to generate approximations to an ideal splat size. Fig. 8 shows the resulting visual difference between fixed and variable splat sizes.

Fixed overall splat size: The splat size is assumed constant for all splats. This option works reasonably well as long as the 3D object is sampled with a reasonably regular pattern, which is the case within the blue-c 3D system, and as long the number of fragments stays constant. Fixed splat size is the only available option for the GL_POINT rendering method, and the fastest option for the circular splats rendering system. The splat size is defined in the configuration script.

Dynamic splat size—image-space method: The original 3D video renderer on Linux [3] implements a nearest-neighbor search in two directions using an image space method. This method results in a very good fit of the individual splats. Unfortunately, it is not practical for objects with more than roughly 15,000 fragments on a fast PC, and not usable at all on the SGI Onyx due to the processing time requirements. It is still available in the blue-c API for experimental reasons and included in the comparison (Fig. 8) as a higher quality reference.

Dynamic splat size—window search method: The window method limits the nearest-neighbor search to the last and next n fragments in the linear data array. n is a variable number that is dynamically adjusted depending on the available CPU time. The window method relies on the assumption that spatially close fragments also appear close inside the linear data structure. This property was verified experimentally. The clustering of fragments is a consequence of the chosen hashing function that is used to index the array, which is actually a quantized encoding of the 3D position. (Fig. 9)

For each fragment while traversing the linear data array, we therefore search for the nearest other fragment inside the window. If the distance to the nearest other fragment is below a threshold, we use the distance as the splat radius. Otherwise, a default size is used instead. By setting the maximum splat size, the method guarantees that no disturbing artefacts are introduced when no close neighbor is found. The array is extended with dummy-entries at the beginning and end to avoid the necessity for special treatment when the search window extends beyond the array bounds. The method successfully reduces blurring artefacts; however it often fails at filling large holes.

The current 3D video acquisition system often changes the overall resolution of the 3D object in response to network drop-outs or in the case of camera switching. To deal with these situations, the default splat size is adjusted according to the number of active fragments, e.g. the configured default is valid at 10,000 fragments, and half the size at 40,000 fragments.

The efficiency of the window search method is greatly increased by using the packed copy of the fragment data structure. As opposed to traversing the sparse fragment array, the linearized copy results in much better cache

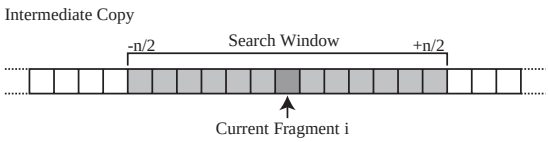

Fig. 9. Nearest-neighbor window search method. For each fragment i, the distance between the fragment and each of the previous $n/2$ and next $n/2$ is calculated. The smallest distance is used as the splat radius if it is below a threshold.

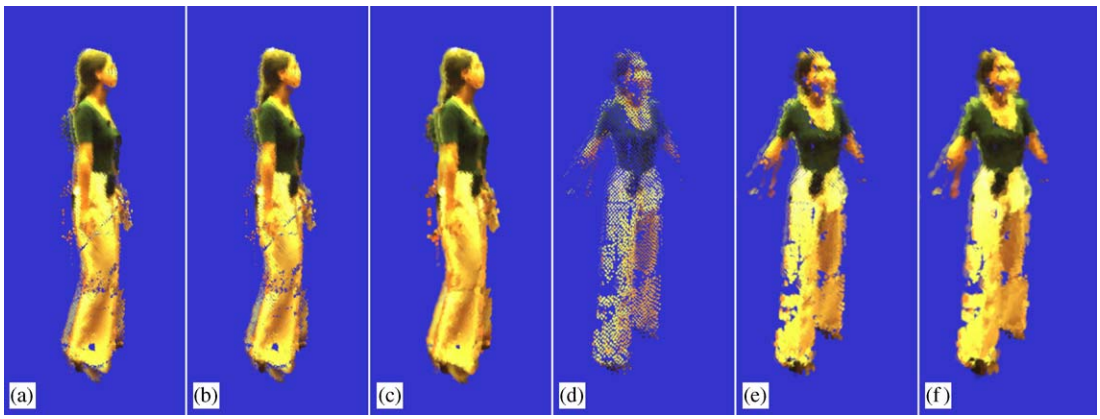

Fig. 8. Fixed splat size vs. variable splat size. (a) and (d) use a fix splat size, (b) and (e) use the window search method, (c) and (f) use the image space method. Image (d) exhibits holes because the fixed size method cannot adapt to a significant change in sampling density.

efficiency and therefore higher read throughput. On the Onyx system, the typical neighbor search window can usually be kept between 32 and up to 256 fragments, depending on the total number of input fragments to be processed. Larger sizes result in highly decreased cache efficiency and therefore significantly worse performance.

The rendering pipeline detects the available processing time by analyzing the number of outstanding buffers in the processing queue. If a new buffer is added for processing before the current is finished, the search window size is reduced in half. Otherwise, the window size is gradually increased until the processing time limit is reached.

The window search method is also available in a non-multiprocessed mode which looks at a window of 16 entries inside the network process. Instead of searching inside the linearized buffer, it keeps a small circular buffer on the stack where it stores the last entries. This buffer is small enough to be kept in the first level cache and therefore very efficient to read.

5.5. Pipeline performance overview

The following Table 3 compares the processing time per fragment for the individual options. The table lists the measurements on the Onyx 3200 as well as on a dual CPU Athlon PC. The results measured on the PC include some variations due to process scheduling effects. The values given are average values, measured with a pre-recorded video sequence of an object with roughly 14,000 fragments.

Table 3
Fragment pipeline processing time. All results are given in microsecond per fragment

Method	Onyx (μs)	Athlon PC (μs)
Point (fixed size)	0.8	0.65
Quad splat (fixed size)	1.0	0.85
Quad splat (window: 16 entries, no multiprocessing)	1.8	1.1
Quad splat (window: intermediate copy step/size calculation at window size 256)	0.8 5.5	0.65 2.9
Image space method (PC only)	—	4.2

6. Example applications

During the three years of the blue-c project, several applications demonstrating various features of the blue-c system have been developed. They present features such as immersive stereo projection, the audio system, 2D video, 3D video streaming and recording, collaboration, animation, and user interaction.

The *Visdome* and *Virtual Museum* applications as depicted in Fig. 10 are both simple architectural walk-through demonstrations. They feature only a very limited amount of application-specific code to load a model file, setup the navigation system, and provide some customized lighting including optional shadows in the case of the Virtual Museum. In that respect, they represent the minimal blue-c application.

Exploring the multimedia capabilities of the blue-c technology for real world applications, Infoticles (see Fig. 11) and IN:SHOP (screenshot in Fig. 2) were developed.

The *Infoticles* series of applications were developed by Andrew Vande Moere [23]. They explore a new metaphor for visualizing complex data sets by the means of moving particles. The visualization technique was applied to several different data sets: monetary flow between the different departments at ETH, web-access logs, and stock market data.

IN:SHOP presents a concept for implementing a distributed shopping environment using tele-presence and 3D video technology [14]. Two example applications have been built based on the same ideas, a fashion shop and a car configurator. The concept of IN: SHOP extends real shopping places with portals into a virtual

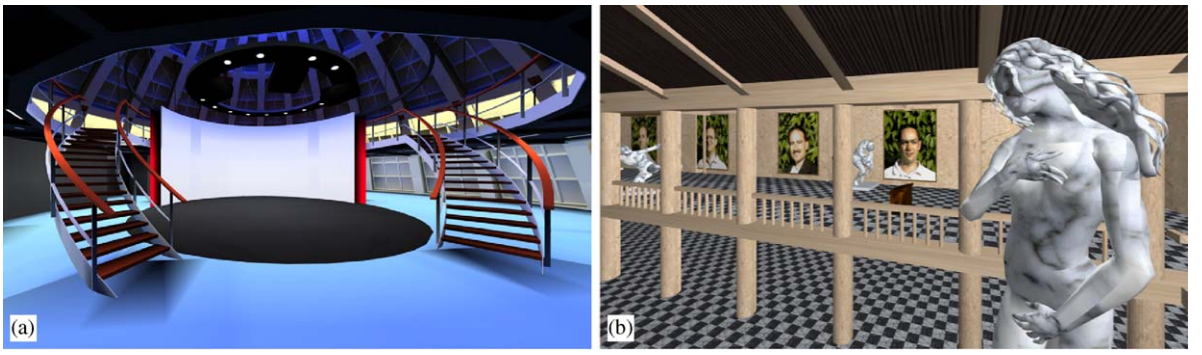

Fig. 10. (a) Visdome, and (b) Virtual Museum walkthrough applications.

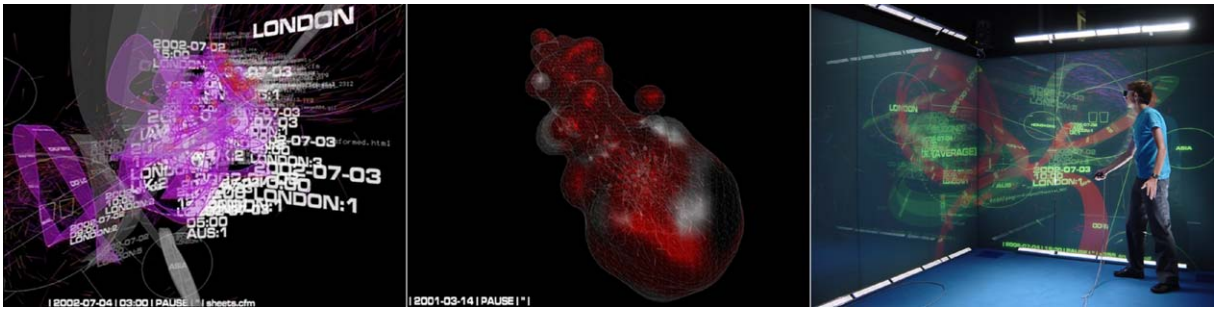

Fig. 11. Infoticles: information visualization application built upon the blue-c API.

world. It profits all multimedia features of the blue-c API including 2D video, 3D video, and audio for background music.

7. Conclusions and future work

This paper presents selected aspects of the blue-c application programming interface, a toolkit for VR development which combines collaboration, telepresence, multimedia, high performance rendering, and interaction tools, into a single, coherent package. Real-time 3D video technology is integrated into a VR environment to reach a new level in telepresence applications.

Future work will concentrate on application development, to validate and demonstrate the use of telepresence in a real-world application context. The possibilities and limits of 3D video in VR should be analyzed. On the technical side, performance and visual quality can be further improved by exploiting programmable graphics hardware that enables new algorithms for 3D video rendering.

Acknowledgements

We would like to thank all members of the blue-c team for many inspiring discussions. Special thanks to those who contributed code and applications: Oliver Kreylos, Edouard Lamboray, Silke Lang, Sascha Scandella, Andrew Vande Moere, Tim Weyrich, and Stephan Würmlin. Additional thanks go to CIPIC at UC Davis for providing the environment for the Linux port. This work has been funded by ETH Zurich as a "Polyprojekt" (Grant no. 0-23803-00).

References

[1] Gross M, Würmlin S, Naef M, Lamboray E, Spagno C, Kunz A, Koller-Meier E, Svoboda T, Van Gool L, Lang S, Strehlke K, Vande Moere A, Staadt O. blue-c: a spatially immersive display and 3D video portal for telepresence. In: SIGGRAPH 2003 conference proceedings, ACM SIGGRAPH annual conference series; 2003.

[2] Spagno C, Kunz A. Construction of a three-sided immersive telecollaboration system. In: Proceedings of the IEEE virtual reality conference 2003 (VR 2003). IEEE: IEEE Computer Society Press; 2003. p. 37–44.

[3] Wuermlin S, Lamboray E, Gross M. 3D video fragments: dynamic point samples for real-time free-viewpoint video special issue on Coding, Compression and Streaming Techniques for 3D and Multimedia Data. Computers & Graphics 2004;28:3–14.

[4] Naef M, Lamboray E, Staadt O, Gross M. The blue-c distributed scene graph. In: Deisinger J, Kunz A, editors. Proceedings of the IPT/EGVE workshop 2003 May 2003.

[5] Cruz-Neira C, Sandin DJ, DcFanti TA, Surround-screen projection-based virtual reality: the design and implementation of the cave. Proceedings of SIGGRAPH 93; August 1993. p. 135–142.

[6] Bierbaum A, Just C, Hartling P, Meinert K, Baker A, Cruz-Neira C. VR Juggler: a virtual platform for virtual reality application development. In: Proceedings of the IEEE virtual reality conference 2001 (VR 2001), Yokohama, Japan, (IEEE): IEEE Computer Society Press; March 2001.

[7] Tramberend H, Avocado: a distributed virtual reality framework. In: Proceedings of the IEEE virtual reality conference 1999; 1999. p. 14–21.

[8] Reitmayr G, Schmalstieg D. An open software architecture for virtual reality interaction. In: Proceedings of the ACM symposium on virtual reality software and technology (VRST) 2001. Banff, Alberta, Canada: ACM; 2001.

[9] Taylor II, RM, Hudson TC, Seeger A, Weber H, Juliano J, Helser AT. Vrpn: a device-independent, network-transparent vr peripheral system. In: Proceedings of the ACM symposium on virtual reality software and technology. New York: ACM Press; 2001. p. 55–61.

[10] Reiners D, Voss G, Behr J, OpenSG—basic concepts. 1. OpenSG symposium; 2002.

[11] Park KS, Cho YJ, Krishnaprasad NK, Scharver C, Leuwis MJ, Leigh J, Johnson AE. CAVERNsoft G2: a toolkit for high performance tele-immservie collaboration. In: Proceedings of the ACM symposium on virtual reality software and technology (VRST) 2000; 2000. p. 8–15.

[12] Naef M, Staadt O, Gross M. blue-c API: a multimedia and 3D video enhanced toolkit for collaborative vr and

telepresence. In: Proceedings of VRCAI 04. New York: ACM Press; June 2004.

[13] Rohlf J, Helman J. IRIS Performer: a high performance multiprocessing toolkit for real-time 3D graphics. In: Proceedings of SIGGRAPH 94, ACM SIGGRAPH annual conference series; 1994. p. 381–95.

[14] Lang S, Naef M, Gross M, Hovestadt L. IN:SHOP: using telepresence and immersive VR for a new shopping experience. In: Proceedings of the eighth international fall workshop on vision, modelling and visualization 2003, IEEE; November 2003.

[15] Naef M, Staadt O, Gross M. Spatialized audio rendering for immersive virtual environments. In: Sun H, Peng Q, editors. Proceedings of the ACM symposium on virtual reality software and technology 2002. New York: ACM Press; November 2002. p. 65–72.

[16] Zwicker M, Pauly M, Knoll O, Gross M. Pointshop 3D: an interactive system for point-based surface editing. In: Proceedings of the 29th annual conference on computer graphics and interactive techniques. New York: ACM Press; 2002. p. 322–29.

[17] Lamboray E. A communication infrastructure for highly-immersive collaborative virtual environments. PhD thesis, ETH Zurich, 2004, No. 15618.

[18] Pfister H, Zwicker M, VanBaar J, Gross M. Surfels: surface elements as rendering primitives. In: SIGGRAPH 99 conference proceedings, ACM SIGGRAPH annual conference series; 1999.

[19] Zwicker M, Pfister H, VanBaar J, Gross M, Surface splatting. In: SIGGRAPH 2001 conference proceedings, ACM SIGGRAPH annual conference series; 2001. p. 371–78.

[20] Rusinkiewicz S, Levoy M, QSplat: a multiresolution point rendering system for large meshes. In: SIGGRAPH 2000 conference proceedings, ACM Siggraph annual conference series; 2000. p. 343–52.

[21] Botsch M, Wiratanaya A, Kobbelt L. Efficient high quality rendering of point sampled geometry. In: Proceedings of the 13th eurographics workshop on rendering; 2002. p. 53–64.

[22] Würmlin S. Dynamic point samples as primitives for free-viewpoint video. PhD thesis, ETH Zurich; 2004, No. 15643.

[23] Vande Moere A, Infoticles: information modeling in immersive environments. In: Proceedings of the sixth international conference on information visualisation (London, England); July 2002. p. 457–61.