

blue-c API: A Multimedia and 3D Video Enhanced Toolkit for Collaborative VR and Telepresence

Martin Naef
Computer Graphics Laboratory
Swiss Federal Institute of Technology
Zurich, Switzerland
naef@inf.ethz.ch

Oliver Staadt
Computer Science Department
University of California
Davis, USA
staadt@cs.ucdavis.edu

Markus Gross
Computer Graphics Laboratory
Swiss Federal Institute of Technology
Zurich, Switzerland
grossm@inf.ethz.ch

Abstract

In this paper we present the blue-c application programming interface, a software toolkit for media-rich, collaborative, immersive virtual reality applications. The blue-c API provides easy to use interfaces to all blue-c technology, including immersive projection, live 3D video acquisition and streaming, audio, tracking, and gesture recognition. The integration of multimedia data, including 2D video, 3D video, and animation, into the scene graph is presented. 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.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism — Virtual Reality

Keywords: Virtual reality software system, 3D video, multimedia, collaborative virtual environments, telepresence

1 Introduction

The *blue-c* system [Gross et al. 2003] developed at ETH 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 and a new real-time video acquisition and transmission approach [Wuermlin et al. 2004].

This paper presents the multimedia integration into the *blue-c application programming interface*, a software toolkit that provides easy access to all underlying blue-c technology for the application developer. Unlike most other VR toolkits that keep graphics, multimedia, and VR specific functionality separated, the blue-c application programming environment is a tightly integrated system, providing consistent interfaces and programming patterns for all tasks involved in building collaborative, immersive virtual reality applications. Besides providing access to blue-c-spe-

cific technology such as real-time 3D video for telepresence, the blue-c API can also be used as a powerful VR toolkit outside the blue-c portals.

After an overview of the software architecture, this paper focuses on 2D video and performance aspects of our 3D video integration. System aspects that are already covered in detail in [Gross et al. 2003, Naef et al. 2003, Wuermlin et al. 2004] will be omitted. We discuss the integration of additional media types into the scene graph, present the different 3D video fragment rendering techniques, and analyze the performance and trade-offs involved in terms of pre-processing time, rendering speed, and visual quality.

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

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 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

There is a significant number of virtual reality toolkits that have been implemented in the past. Most of them mainly provide configurable device I/O and setup of the rendering system. A few support collaborative work by providing some sort of network layer. CAVElib™ development was started with the initial CAVE™ [Cruz-Neira et al. 1993] 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. VR Juggler [Bierbaum et al. 2001] provides a mature, object-oriented approach to VR. It is very actively supported. Similar to CAVElib™, VR Juggler mostly leaves the choice of the rendering system to the application developer and keeps only loose ties to the scene graph. While it can be enhanced with additional toolkits, it does not immediately provide multimedia support. DIVERSE [Kelso et al. 2002] is another approach to VR, it offers similar services and capabilities to CAVElib™ or VR Juggler while trying to stay as open as possible. None of these toolkits were optimized for media rich collaborative applications and telepresence.

Avango [Tramberend 1999] provides a shared scene graph and interaction support. It is closely coupled to OpenGL Performer, but it changes the scene graph interface to an Inventor-style field system that is exposed to a scripting language. The blue-c API leaves

Copyright © 2004 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions Dept, ACM Inc., fax +1 (212) 869-0481 or e-mail permissions@acm.org.
© 2004 ACM 1-58113-884-9/04/0006 \$5.00

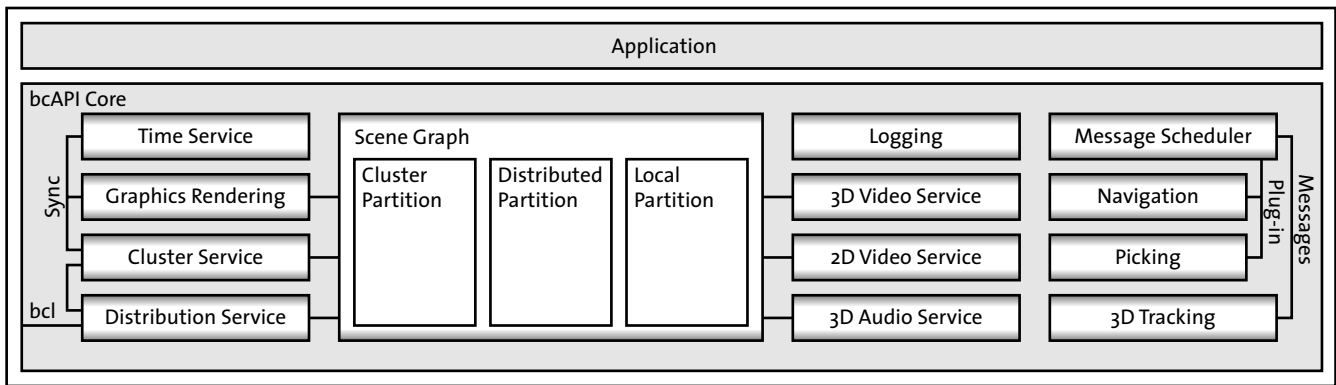


Figure 1: blue-c API System overview: Services accessing the scene graph, and message scheduler with sources and plug-ins.

the Performer interfaces unchanged to support legacy applications and to immediately support new Performer features as they are released.

An overview of networked virtual reality environments can be found in [Singhal and Zyda 1999]. Most of these systems aim at efficiently connecting a large number of sites and therefore focus on scalability issues. They rely on other toolkits for VR-specific features. The blue-c however targets a small number of portals with a high-speed interconnect.

There is a plethora of toolkits available that provide parts for virtual reality systems, including tracking libraries [Reitmayr and Schmalstieg 2001], scene graphs [Reiners et al. 2002], networking tools [Park et al. 2000], 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, we aimed at providing a holistic, consistent, and well integrated toolkit that provides strong multimedia and 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 us to integrate media handling directly into the scene graph without compromising performance.

3 blue-c API System Architecture

This section presents the system architecture of the blue-c API. An overview of all components and their main dependencies is given in Fig. 1.

3.1 Core

The blue-c API core class forms the system kernel. It handles system initialization and startup, instantiation of services, runs the main application loop, and takes care of a clean system shutdown.

The core class is a singleton which is accessible from all processes. It provides efficient service discovery methods. The core also keeps a list of spawned child processes on both the master system node as well as remote processes (e.g., 3D video server), it terminates any remaining processes left over after service shutdown.

3.2 Process Management

The blue-c API is built around the OpenGL Performer scene graph and real-time rendering system [Rohlf and Helman 1994] and uses parts of its process and shared memory management for process synchronization and communication.

Performer splits the rendering pipeline into three stages: *App*, *Cull* and *Draw*. There is only one App process, inside which the application can modify the scene. Cull and Draw processes are spawned for each graphics pipeline. blue-c services are responsible to pass data down the pipeline in a multiprocessing-safe manner. On single or dual-processor PC hardware, the different stages of the rendering pipeline are kept in a single process for efficiency.

3.3 Services

All blue-c functionality is implemented as a set of services. These provide a common interface for naming, startup, and shutdown. They are instantiated by the core based on configuration scripts (see Section 3.8). In addition to those explained in Sections 4 and 5, current blue-c services include logging, distributed scene synchronization, and simulation time. Graphics rendering including multi-pipe setup is also encapsulated as a service.

Most services spawn their own processes that communicate through the shared arena memory segment. In addition, service callbacks are invoked once per frame, and once for each display channel during culling and drawing, providing processing time whenever needed inside the rendering pipeline.

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 queuing 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.

The messages are collected and dispatched by the message scheduler, which is hosted by the core object. Messages can be issued from any process, the scheduler takes care of serialization. If low-latency handling is important, the application may register a method which is called immediately inside the issuing process

context. Most applications, however, will choose to receive the messages inside the main application process.

The application developer can define and register new message types. This can be useful to signal events from custom objects during scene graph traversals or to implement applications-specific device drivers without integrating them into the blue-c API.

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.

The tracking service generates messages for button events as well as device movements. The current position and button state is also provided for applications that use a polling paradigm for user-interface input.

The current tracking service supports Ascension and Polhemus magnetic tracking systems based on our own driver code. The integration of other tracking toolkits such as trackd or OpenTracker [Reitmayr and Schmalstieg 2001] would be straight forward.

A first prototype of a vision-based gesture recognition system is integrated as a tracking service. Detected hand and head positions are provided to the application encapsulated as a virtual tracking sensor.

3.6 Navigation and Interaction Plug-ins

Navigation inside the virtual world and picking of objects are common tasks in most VR applications. The blue-c API therefore provides a simple plug-in mechanism for registering navigation and interaction modules. Simple fly-motion navigation with collision detection and pre-defined and dynamic setting of viewpoints, as well as basic object picking are provided by the API. They can easily be customized for individual applications, or be replaced completely with custom implementations. The picking system generates application messages, whereas the navigation plug-in continuously sets the viewpoint.

The plug-ins are hooked into the message passing system. They receive all messages not explicitly flagged as *handled* by the application. In addition, the navigation plug-in is called once per frame to calculate the new viewpoint. The default navigation implementation includes an overridable position validation method which does collision detection using Performer intersection tests.

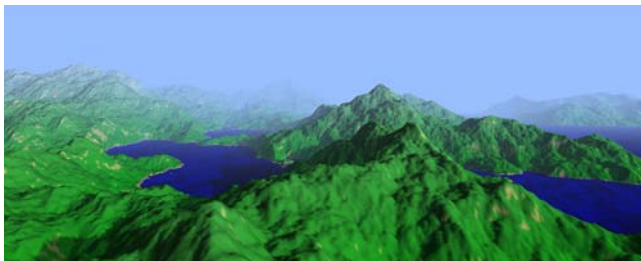


Figure 2: Demo application, featuring efficient terrain follower code in a customized navigation plug-in.

In the demo application as shown in Fig. 2, the default collision detection is replaced with a simple terrain follower code that uses

the application height field representation instead of the geometry mesh for improved efficiency.

3.7 Scene Graph

The blue-c scene graph is based on OpenGL Performer. To support collaboration and cluster rendering, it has been enhanced with node serialization and state update interfaces. These features are presented in [Naef et al. 2003]. Additional custom nodes and attribute objects are integrated to support multimedia elements. These objects are tied together with their respective services by the use of callback methods that are invoked during application, cull, or draw traversal (see Section 3.2). These proxy objects are described in Section 4.

3.8 Configuration System

The configuration information is stored in a tree hierarchy. It reflects the service structure, including sub-devices such as tracking sensors. The configuration scripts are human-readable text files. Include directives allow to link different configuration files, allowing to easily combine common items into manageable configuration sets.

4 Multimedia Support

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

4.1 2D Video

The integration of 2D video streams greatly enhances the visual appeal of virtual environments. It is significantly more efficient, both in terms of modeling time and real-time rendering performance, to create interesting animated backgrounds for a scene with a video backdrop instead of animated geometry. Fig. 3 shows a screenshot from the IN:SHOP [Lang et al. 2003] prototype application with a video background. It provides an attractive environment with a very low polygon count.



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

Video-integration into the blue-c API consists of two parts. The video service handles a list of video sources, each delivering video data from either a file or a video camera. It includes a simple control interface for starting, pausing or stopping the video stream. Texture attribute objects inside the scene graph then connect to the

video source. Texture objects are derived from standard Performer textures, providing an identical interface to the developer. Instead of specifying an image file to load, a "video:" prefix and the requested filename or camera name is set. The API takes care of starting the necessary video sources, either as defined in the configuration scripts for external cameras, or dynamically by opening the requested video files.

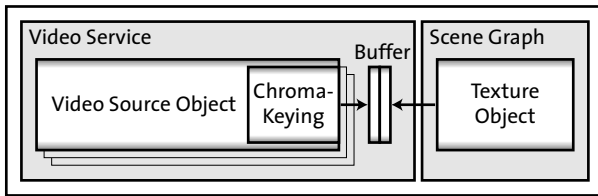


Figure 4: Video decoding service and texture object in scene.

Video streaming and decoding is performed in a separate process for each source, completely decoupling the video frame rate from the 3D graphics rendering rate. Texture data is uploaded to the graphics pipe by the video service whenever a new frame is available in a draw callback. For video file decoding, we use the SGI digital media library on the Irix platform, and libavcodec on Linux. On our 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.

A plug-in mechanism into the decoder allows to do blue-screening for video files on the fly. This enables to use transparency effects in video files based on color keying for video file formats that typically do not include alpha channels. Alpha channels could also be used to support telepresence from portals that do not use the full 3D video acquisition system.

4.2 3D Audio

In addition to visual output, the blue-c API provides a high-quality spatialized 3D audio system that runs as a service. It is controlled by active audio nodes that are part of the scene graph, which allows to add sound as attribute to scene graph 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. For a detailed description of the blue-c audio rendering pipeline, we refer to [Naef et al. 2002].

4.3 Animation

The blue-c API supports animation of 3D geometry using the concept of *animation nodes*. Animation nodes are transformation nodes that update their transformation matrix for every frame. The animation node base class calls a virtual update method once per frame inside the application process and provides functionality to pass the matrix down the rendering pipeline in a multiprocess-safe way.

The main idea behind animation nodes is to provide the application developers with a framework to integrate their own animation code. In addition, the API provides some derived animation nodes that implement continuous rotation, and a key-frame animation class that provides linear interpolation for both position and rotation.

As a special "fun" feature, a 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. Fig. 5 shows a screenshot of our virtual museum with two dancing mannequins.



Figure 5: Poser-animated figures in the virtual museum.

5 3D Video Fragments 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 [Gross et al. 2003] and [Wuermlin et al. 2004]. The following discussion only refers to the receiver side, including rendering.

Most current point rendering systems [Rusinkiewicz and Levoy 2000], [Botsch et al. 2002] are suitable for static point sampled models only. They focus on high quality rendering. However, the proposed solutions are not suitable for real-time rendering of dynamic objects because the steps required to generate the underlying data structures are not feasible within the given time bounds. For the blue-c, we therefore had to optimize the full 3D fragment rendering pipeline. Our 3D video inlays typically consist of approximately 20,000 to 30,000 fragments that are updated ten times per second.

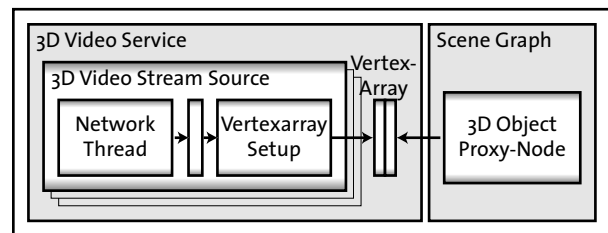


Figure 6: 3D video service with video fragment sources and proxy object inside the scene graph. A double-buffered vertex array is used to pass data between the processes.

The integration of 3D video streams into the blue-c API follows a two-tier approach, similar to 2D video textures. The 3D video service as shown in Fig. 6 handles asynchronous streaming,

decoding and data structure setup of the 3D video fragments. A proxy node inside the scene references the source service and calls the rendering methods.

5.1 3D Video Service

The 3D video service can host several fragment data sources. Each source implements a pre-processing and rendering pipeline for a single 3D video object. Fig. 7 depicts the pipeline stages, Fig. 8 shows the main data structures used. The following subsections describe the stages and their performance properties as 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 PC with a Pentium 4 2.8 GHz processor and nVidia Quadro4 750 XGL graphics board.

Incoming data: 3D video objects are transmitted using fragment operators [Wuermlin et al. 2004]. They are decoded and used to update a sparse, linear data array incrementally. A hash function of the position is used as index into this array, which enables fast update operations if the position or color of a fragment changes only. This data array structure holds the complete object as a collection of 3D video fragments that include position, color, and surface normal.

Especially on the 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.

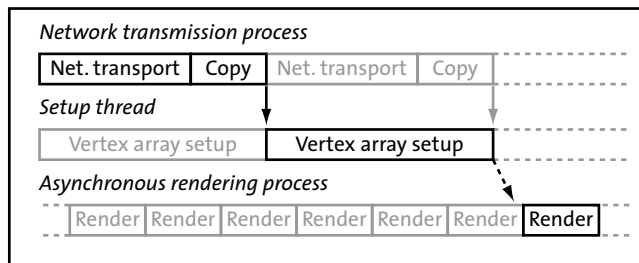


Figure 7: Video fragment rendering pipeline.

Vertex array setup: For efficient rendering, the point samples should be copied into a linear 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. This vertex array can be sent to the OpenGL rendering system with a single system call (`glVertexArray`). Due to the dynamic nature of the input data, the vertex array must be set up completely for each incoming frame. The update must therefore be as efficient as possible. This acquisition frame rate is typically in the order of 10 Hz, depending on the configuration of the acquisition system.

5.2 Splat Setup

For high quality rendering, we use a method which is a simplified version of EWA splatting [Zwicker et al. 2001], [Ren et al. 2002], accelerated by video hardware. Each 3D video fragment is treated as a flat disc in object space with a given radius that is blended with the neighbors according to a weight defined by a Gaussian function.

For actual rendering, each point sample is represented with a `GL_QUAD` primitive. The quad is set up perpendicular to the frag-

ment normal and has a variable size. This square primitive includes an alpha texture representing the Gaussian function, which is used for smooth blending. The same texture is also used in an alpha testing step, resulting in circular splats in object space.

Four vertices, totalling 96 bytes, must be written into the vertex array for each point sample. Setting up the splats is limited in performance by the memory write bandwidth. Including traversal of the sparse linear data structure, each splat takes approximately 1 μ s on the Onyx and 0.5 - 0.8 μ s on the PC, leaving very little time for additional processing such as dynamic splat size calculation. Instead of setting up the vertex data structure inside the network streaming process, we therefore create a linearized copy of the video fragment data structure, which only takes 24 bytes per fragment. This copy operation takes approximately 0.4 μ s per fragment on the Onyx, leaving about 90% of the processing time to the network code. The copy step is necessary because the fragment data structure does not allow for efficient concurrent access due to unpredictable ordering of the update operations.

After creating the linear fragment list inside 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 to 5 μ s per fragment is available, leaving room for a heuristic calculation of the splat size. The splat size is a function of the distance to the nearest neighbors.

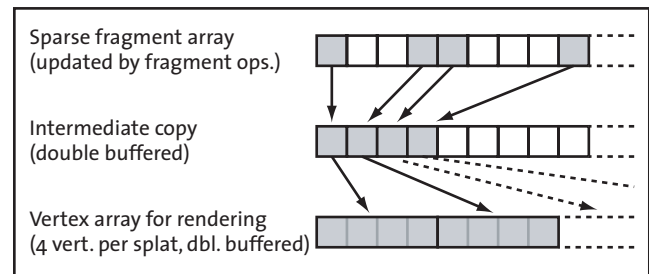


Figure 8: Data structures for 3D fragment processing and rendering.

The splat size should be big enough to avoid visible holes, yet small enough to avoid unnecessary blurring. Due to the non-uniform sampling, each fragment's radius should be adapted individually according to its closest neighbors. The linear data structure, however, does not allow for fast querying of the nearest neighbor. Still, due to quantized encoding of the data during network transmission, and due to the nature of the hashing function used to index the sparse fragment array, many fragments are actually clustered together in the data structure. Our experiments show that a close neighbor can usually be found within a search window looking at the last and next 32 fragments to be processed inside the array. For each fragment, we therefore search for the nearest other fragment inside this window. If the distance is below a configurable threshold, we use the distance as the splat radius. Otherwise, a default size is used.

As opposed to traversing the sparse fragment array, the linearized copy results in much better cache efficiency and therefore higher read throughput, allowing one to keep the typical neighbor search window between 32 and up to 256 fragments, depending on the total number input fragments to be processed. The size of the window is automatically adapted to the available processing time.

Both the intermediate fragment array and the output vertex array are double-buffered. Buffers are locked and swapped at the end of the calculation process. The latest complete vertex array is therefore always available for asynchronous reading of the render-

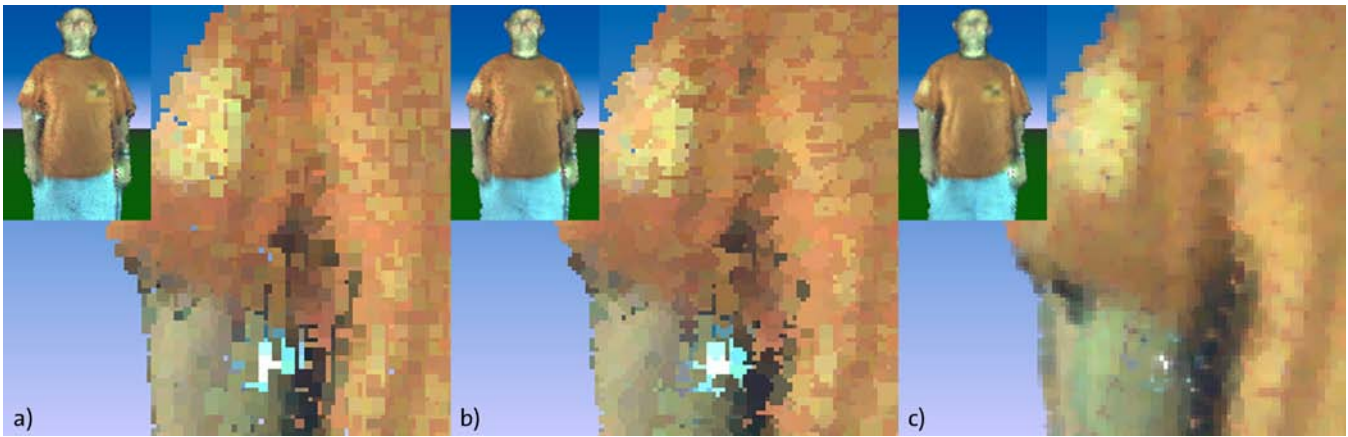


Figure 9: Comparison of fragment rendering options: a) Points, b) single pass, and c) two-pass circular splats.

ing system. On a single processor machine, the copy step is omitted and the splat sizes and resulting vertices are calculated immediately.

5.3 Splat Rendering

3D video objects can be referenced anywhere in the scene graph hierarchy using proxy nodes. Rendering the fragments is triggered by the node callback method inside the draw process.

Rendering is done using OpenGL vertex arrays in a two pass procedure. In the first pass, the depth buffer is set up: Rendering quads using the Gaussian textures with a greater-than alpha function results in circular splats. These are rendered into z-buffer only, the color buffer is left untouched. During the second pass, z-buffer writing and alpha testing is disabled. The colors are now blended with the frame buffer color according to the alpha texture value, which results in smooth corners and blending of overlapping splats. A polygon offset is added to the z-value to avoid z-fighting artefacts during blending. Splat rendering is visualized in Fig. 10.

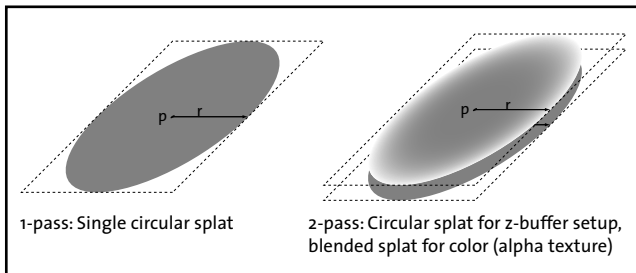


Figure 10: Splat rendering: Single pass and two-pass rendering modes. Circular splats are rendered with a quad primitive and an alpha texture as stencil. The same alpha texture is used for blending. p denotes the fragment position, r the size of the splat.

Rendering performance: Our rendering method produces smooth looking surfaces at interactive frame rates. On the SGI Infinite Reality 3 graphics pipe, we achieve rendering times in the order of $1.3 \mu\text{s}$ per fragment, resulting in typical frame rates around 15 to 20 Hz in stereo mode for applications with low scene geometry complexity.

For applications that require higher frame rates, a single pass rendering mode is available that either renders circular splats with

out blending, or points only. Rendering performance is mainly a function of the number of bytes and vertices transmitted per fragment. The Quadro4 graphics hardware is significantly faster than the IR3, the relative characteristics however remain the same. Table 1 summarizes the rendering times for both platforms. The visual results of the different rendering methods are shown in Fig. 9.

Table 1: Rendering time per fragment.

Platform	Point	Circular 1 pass	Circular 2 pass
SGI Onyx / IR3	$0.1 \mu\text{s}$	$0.5 \mu\text{s}$	$1.3 \mu\text{s}$
nVidia Quadro 4	$0.02 \mu\text{s}$	$0.15 \mu\text{s}$	$0.3 \mu\text{s}$

To save on processing power, the splat size approximation and multiprocessed vertex array setup can be turned off. A fixed splat size is used instead, which typically still results in acceptable visual quality due to the low variation in sampling density of our acquisition system.

6 Example Applications

The usability of an application programming interface is best verified by actual application development. During the three years of the blue-c project, several applications have been developed. They present the different blue-c features, such as immersive stereo projection, the audio system, 2D video, 3D video streaming and recording, collaboration, animation, and user interaction.

Exploring the multimedia capabilities of the blue-c technology for real world applications, Infoticles (see Fig. 11) and IN:SHOP (screenshot in Fig. 3) were developed.

Infoticles [Vande Moere 2002] is an immersive data visualization application which uses the motion of particles as a visual metaphor to enable the discovery of unexpected data patterns in large, time-based datasets. It has been implemented for the analysis of monetary flows as well as for corporate knowledge document usage.

IN:SHOP [Lang et al. 2003] introduces a new shopping experience where a remote sales person assists a customer. The basic idea is to extend real shopping places with portals into a virtual world. It uses all multimedia features of the blue-c API including 2D video, 3D video, and audio for background music.

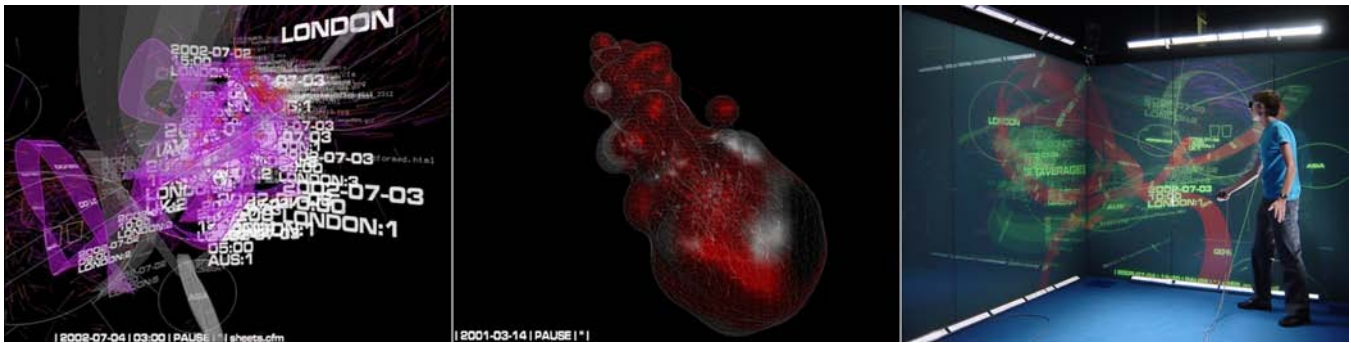


Figure 11: Infoticles: Information visualization application built upon the blue-c API.

7 Conclusions and Future Work

This paper presents the blue-c application programming interface, a new virtual reality development toolkit which combines collaboration, telepresence, multimedia, high performance rendering, and interaction tools, into a single, coherent package. It combines current state of the art in real-time 3D video technology and multimedia into a VR environment.

Future work will focus on application development, testing the possibilities and limitations that the blue-c system and application programming interface offer for collaborative virtual environments and telepresence. We will also exploit programmable graphics hardware for higher performance 3D video rendering and to improve the visual quality of the virtual environments. Especially point-sprite primitives supported by the latest generation of graphics accelerators enable to reduce the bandwidth requirements between main memory and the graphics system.

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

- BIERBAUM, A., JUST, C., HARTLING, P., MEINERT, K., BAKER, A., AND CRUZ-NEIRA, C. 2001. VR Juggler: A virtual platform for virtual reality application development. In *Proceedings of the IEEE Virtual Reality Conference 2001 (VR 2001)*. IEEE, IEEE Computer Society Press, Yokohama, Japan.
- BOTSCH, M., WIRATANAYA, A., AND KOBELT, L. 2002. Efficient high quality rendering of point sampled geometry. In *Proceedings of the 13th Eurographics Workshop on Rendering*. 53–64.
- CRUZ-NEIRA, C., SANDIN, D. J., AND DEFANTI, T. A. August 1993. Surround-screen projection-based virtual reality: The design and implementation of the cave. *Proceedings of SIGGRAPH 93*, 135–142.
- 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., AND STAADT, O. 2003. blue-c: A spatially immersive display and 3D video portal for telepresence. In *SIGGRAPH 2003 Conference Proceedings*. ACM SIGGRAPH Annual Conference Series.
- KELSO, J., ARSENAULT, L. E., SATTERFIELD, S. G., AND KRIZ, R. D. 2002. DIVERSE: A framework for building extensible and reconfigurable

device independent virtual environments. In *Proceedings of the IEEE Virtual Reality Conference 2002 (VR 2002)*. IEEE, IEEE Computer Society Press, Orlando, Florida, 183–190.

- LANG, S., NAEF, M., GROSS, M., AND HOVESTADT, L. 2003. IN:SHOP: Using telepresence and immersive VR for a new shopping experience. In *Proceedings of the 8th International Fall Workshop on Vision, Modelling and Visualization 2003*. IEEE.
- NAEF, M., LAMBORAY, E., STAADT, O., AND GROSS, M. 2003. The blue-c distributed scene graph. In *Proceedings of the IPT/EGVE Workshop 2003*, J. Deisinger and A. Kunz, Eds.
- NAEF, M., STAADT, O., AND GROSS, M. 2002. Spatialized audio rendering for immersive virtual environments. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology 2002*, H. Sun and Q. Peng, Eds. ACM Press, 65–72.
- PARK, K. S., CHO, Y. J., KRISHNAPRASAD, N. K., SCHARVER, C., LEUWIS, M. J., LEIGH, J., AND JOHNSON, A. E. 2000. CAVERNsoft G2: A toolkit for high performance tele-immersive collaboration. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST) 2000*. 8–15.
- REINERS, D., VOSS, G., AND BEHR, J. 2002. OpenSG - Basic concepts. 1. OpenSG Symposium.
- REITMAYR, G. AND SCHMALSTIEG, D. 2001. An open software architecture for virtual reality interaction. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST) 2001*. ACM, Banff, Alberta, Canada.
- REN, L., PFISTER, H., AND ZWICKER, M. 2002. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Proceedings of Eurographics 2002*. COMPUTER GRAPHICS Forum, Conference Issue. 461–470.
- ROHLF, J. AND HELMAN, J. 1994. IRIS Performer: A high performance multiprocessing toolkit for real-time 3d graphics. In *Proceedings of SIGGRAPH 94*. ACM SIGGRAPH Annual Conference Series. 381–395.
- RUSINKIEWICZ, S. AND LEVOY, M. 2000. QSplat: a multiresolution point rendering system for large meshes. In *SIGGRAPH 2000 Conference Proceedings*. ACM Siggraph Annual Conference Series. 343–352.
- SINGHAL, S. AND ZYDA, M. 1999. *Networked Virtual Environments: Design and Implementation*. ACM Press - SIGGRAPH Series. Addison-Wesley.
- TRAMBEREND, H. 1999. Avocado: A distributed virtual reality framework. In *Proceedings of the IEEE Virtual Reality Conference 1999*. 14–21.
- VANDE MOERE, A. 2002. Infoticles: Information modeling in immersive environments. In *Proceedings of the 6th International Conference on Information Visualisation*. London, England, 457–461.
- WUERMLIN, S., LAMBORAY, E., AND GROSS, M. 2004. 3D video fragments: Dynamic point samples for real-time free-viewpoint video. *Computers & Graphics, Special Issue on Coding, Compression and Streaming Techniques for 3D and Multimedia Data 28*, 1 (Jan.), 3–14.
- ZWICKER, M., PFISTER, H., VANBAAR, J., AND GROSS, M. 2001. Surface splatting. In *SIGGRAPH 2001 Conference Proceedings*. ACM SIGGRAPH Annual Conference Series. 371–378.

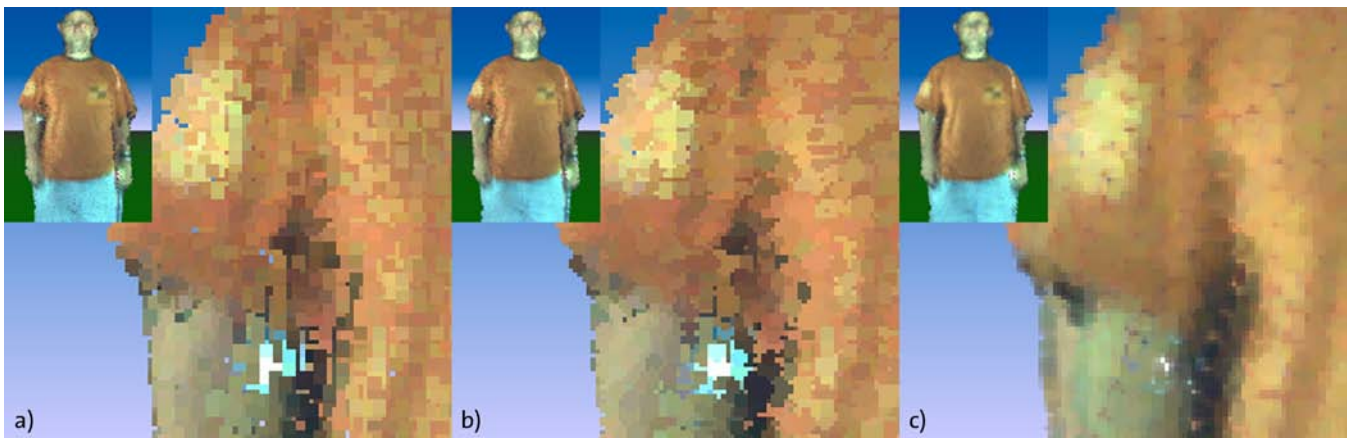


Figure 9: Comparison of fragment rendering options: a) Points, b) single pass, and c) two-pass circular splats.

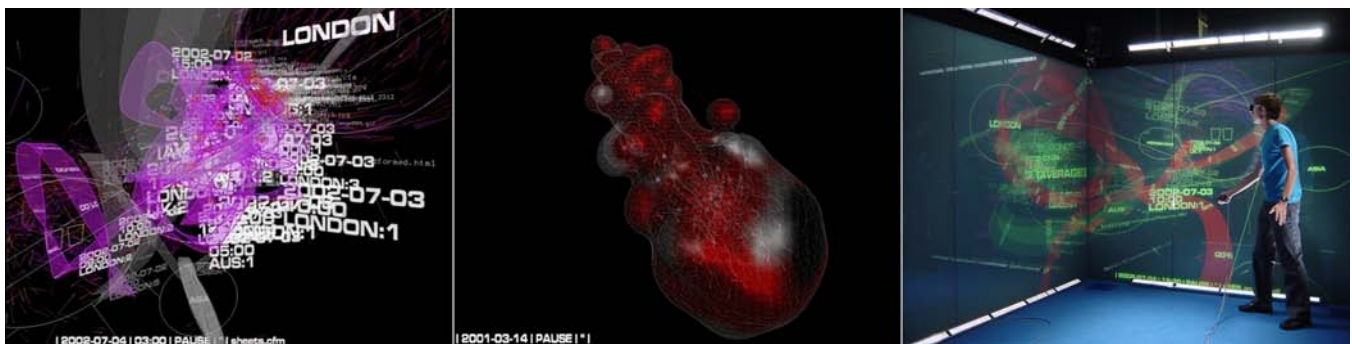


Figure 11: Infoticles: Information visualization application built upon the blue-c API.

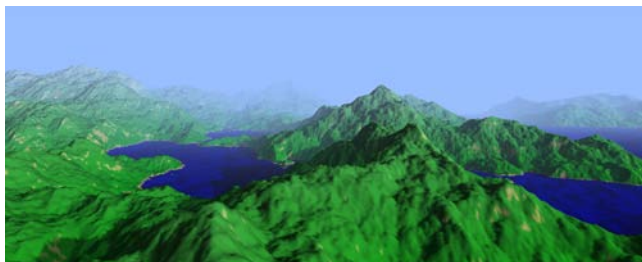


Figure 2: Demo application, featuring efficient terrain follower code in a customized navigation plug-in.

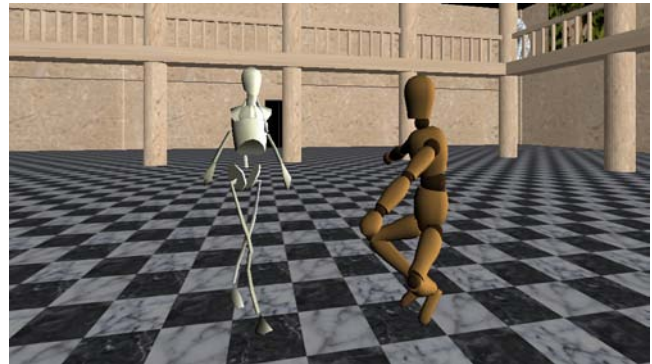


Figure 5: Poser-animated figures in the virtual museum.



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