Virtual Reality Surgical Simulator Software Development Tools

Article · December 2012

DOI: 10.1057/jos.2012.22

CITATIONS
14

READS
1,727

2 authors:

Greg Ruthenbeck
Flinders University
25 PUBLICATIONS **266** CITATIONS
SEE PROFILE

Karen J Reynolds
Flinders University
175 PUBLICATIONS **1,799** CITATIONS
SEE PROFILE

Journal of Simulation (2013) 7, 101–108

www.palgrave-journals.com/jos/

Virtual reality surgical simulator software development tools

GS Ruthenbeck* and KJ Reynolds

Flinders University, Adelaide, Australia

Virtual reality (VR) surgical simulations are among the most difficult software applications to develop mainly because of the type of user interactions that they must support. Surgery typically includes precise cutting of often intricate structures. Modelling these structures and accurately simulating their response to user interaction requires many software components to effectively work in unison. Some of these components are readily available but are tailored to more common applications such as computer games or open-world simulations such as flight-simulators. This article explores the software libraries that are currently available to developers of VR surgical simulation software. Like computer games and other VR simulations, VR surgical simulations require real-time lighting and rendering systems and physics-based interactions. However, in addition they require haptic interaction with cut-able and deformable soft-tissue, a key requirement that is not supported by the majority of the available tools. In this article, we introduce currently available software development tools and the specific benefits and limitations that can be encountered when using them to develop VR surgical simulations. We also provide a detailed review of collision detection libraries that are central to achieving reliable haptic rendering.
Journal of Simulation (2013) **7,** 101–108. doi:10.1057/jos.2012.22; published online 14 December 2012

Keywords: virtual reality; simulation; surgery; software

1. Introduction

Simulation development combines assets such as models, shaders, shader resources, animations, and physics rigs together to create virtual environments. Game and simulation engines provide reusable frameworks to simplify development. However, oversimplification complicates the development of important custom features specific to each type of simulation. Whether it be the open-expanses and atmospheric lighting effects of a flight simulator, or the proper handling of self-collisions of parts of the simulated anatomy of a virtual surgical patient, each software component imposes constraints in return for specific gains (features, improved performance and/or reduced developer effort).

Selecting and combining libraries is a good way to avoid re-implementing common tasks. Care must be taken to ensure that libraries, written with a certain usage in mind, do not inhibit implementation of key features. Hence, despite the prevalence of game development APIs and open-world simulation APIs (commonly used for flight or combat simulations), there are very few software libraries that cater directly to medical simulation developers.

Perhaps the biggest challenge facing developers of virtual reality (VR) simulations for surgical training is the lack of accessible developer tools that can deliver the required user experience, for example, that support interactively cut-able and deformable soft-tissues. The quality of surgical simulations is increasing as the core techniques and technologies improve and reduce the developer effort required to create simulations for different surgical interactions and reliable haptic rendering. However, the software components used to support surgical interactions like cutting and volumetric tissue removal are fragmented. This article outlines the tools available to assist developers in developing simulations with surgical interactions. We also describe the role of the available haptic tool-kits in supporting surgical interactions.

2. Software tools

VR surgical simulations typically consist of one or more haptic interface device(s), a display, and a computer that is running the simulation software (Figure 1). The simulation software simulates an interactive anatomically accurate model of a virtual patient (eg, Figure 2 shows a visually realistic tissue model being interactively cut using a surgical instrument known as a micro-debrider). The anatomical models are simulated graphically, physically (eg, soft-tissue deformation and other types of physics-based animation), and haptically (summarised in Figure 3) to deliver an interactive visual and tactile experience with sufficient realism to engage the user and produce the desired learning outcome. The success of a simulation relies on the interactive tissue models used to compose the anatomical model

*Correspondence: GS Ruthenbeck, School of Computer Science, Engineering and Mathematics, Flinders University, Sturt Rd, GPO Box 2100, Adelaide, SA 5001, Australia.
E-mail: greg.ruthenbeck@flinders.edu.au

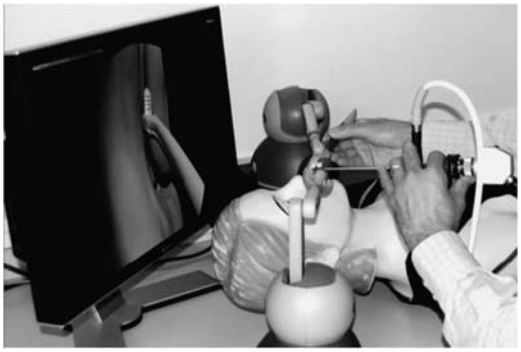

Figure 1 A VR Endoscopic Sinus Surgery Simulation by the authors that uses 2 haptic devices to control; 1. Surgical tools, 2. The endoscope.

Figure 2 The simulated endoscopic view of a surgical tool removing tissue from the sinuses by the authors.

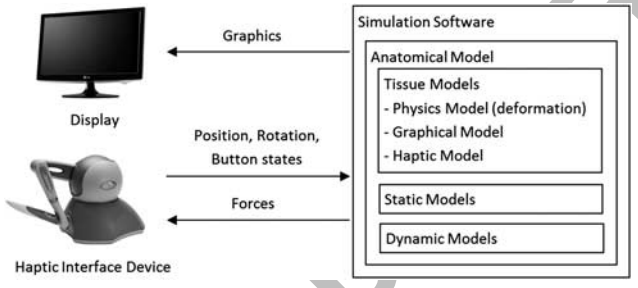

Figure 3 VR simulation system block diagram.

behaving plausibly in response to user interaction. The software sub-systems required to achieve this are summarised in Figure 4.

There are many software tools that must be combined to efficiently realise the visual and haptic realism required. These tools must work together efficiently in order to maintain a sufficiently high visual update-rate, as well as a high haptic update-rate to avoid distracting visual and tactile artefacts that reduce the simulations' effectiveness. The following sections discuss these simulation software development tools under the following headings; Medical Simulation APIs,

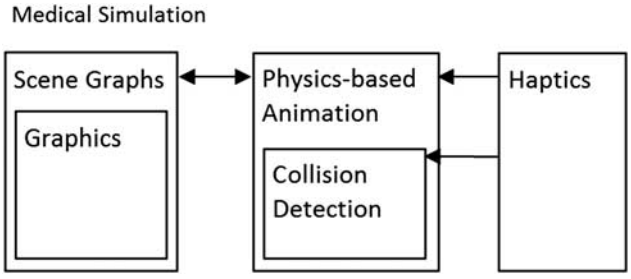

Figure 4 Simulator software block diagram.

Scene graph APIs, Graphics Rendering libraries, Physics-based Animation libraries, Collision detection libraries, GPU-based collision detection, and Haptic libraries.

2.1. Medical simulation APIs

Virtual scenes in surgical simulations are different to most other types of simulations and visualisations where scene graphs are typically employed. Medical simulations often consist of enclosed environments densely populated with interconnected structures that must respond plausibly to user interactions such as palpation. Under such circumstances there may be little benefit in performing common scene graph optimisations such as per-object occlusion tests. Instead, a per-polygon occlusion is more appropriate if detailed deformable objects are used since different parts of a single model may be occluded depending on current deformation and positioning (which rarely occurs in more open environments). Hence, many of the advantages typically provided by scene graph APIs do not apply to the types of scenes commonly found in medical simulations. However, some features still provide significant advantages in this context.

Medical Simulation APIs cater to developers of medical simulation by providing similar functionality to game engines with additional modules, interfaces, and capabilities tailored to the medical simulation application domain. These APIs are often collections of research work and as such tend to be less tightly integrated than commercial APIs.

The Simulation Open Framework Architecture (SOFA) is an open-source project founded by researchers of the Alcove group at INRIA(Cotin *et al*, 2005). The project aims to facilitate development of simulators using a modular architecture that maximises component re-use while 'minimizing the impact of this flexibility on the computation overhead' (Cotin *et al*, 2005). It is being actively maintained and developed by developers at INRIA and more widely contributed to by researchers such as the CIMIT Sim Group, ETH Zurich, and CSIRO (SOFA project—(C) INRIA, 2009).

The current version of SOFA (1.0 beta 4) (SOFA project—(C) INRIA, 2009) includes support for different types of deformable models based on mass-springs or linear

and co-rotational finite element method (FEM), fluid models, and a number of collision detection and response methods. Each component is configurable via XML to facilitate experimentation and simulation development.

SOFA is designed for use in medical simulation development. SOFA uses a scene graph to describe not only the rendered scene but also the simulation components that combine to produce interactive simulation elements such as interactive tissues. The component types include mechanical system models, surface representations, collision detection algorithms, and constraint solvers. As such, SOFA is maturing to become an invaluable contribution to the field of medical simulation research. However, the use of such a versatile architecture is not without its limitations.

Flexibility in software architectures is a trade-off with specialisation. Specialisation brings speed and higher performance. It therefore becomes a question of whether sufficient performance can be achieved while retaining the desired flexibility. With the rapid changes occurring in general purpose graphics processing unit (GPGPU) APIs, the overhead of re-engineering flexible interfaces is particularly challenging. For example, CUDA (from NVIDIA) has evolved from C-only language support to include C++ language features. Retaining flexibility of components written for radically changing APIs is difficult and time consuming. Hence, although SOFA has a lot to offer, it requires developers to conform or adapt to its architecture.

H3D is an open-source engine targeted at medical simulation developers. The H3D API is maintained by SenseGraphics (Kista, Sweden) and is a well-tested platform for the development of medical simulations with support for haptic interaction. The H3D software development kit (SDK) includes the source code for a demonstration application in which the user can use a haptic device to cut a flat plane of simulated skin using a scalpel. Further evidence of the maturity and capacity of H3D to facilitate medical simulation development is the number of applications that have been developed using it for example, a liver biopsy simulation (Villard *et al*, 2009), and Imagine-S (Gould *et al*, 2011). However, as with any open-source project (particularly large and complex ones) one pitfall of employing H3D is that developers must learn a new architecture and adapt any additional features to function within this architecture.

GiPSi (General Physical Simulation Interface) integrates open-source libraries (including the open-source C# game engine TAO (Ridge *et al*, 2011), the open-source collision detection library OPCODE (Terdiman, 2003), and OpenHaptics (from Sensable, Wilmington, MA) to simplify development of re-usable organ-level simulation components for tactile medical simulation (Cavusoglu *et al*, 2006). Unfortunately the project no longer appears to be active (the last release of the API was 29 October 2008), and it does not include any re-usable components for tissue simulation. Other simulation APIs include NeuroVR (Faletti and Vezzadini, 2007), SPRING (Montgomery *et al*, 2002), and SSTML (Bacon *et al*, 2006) that cater to simulation developers, but do not include tissue simulation capabilities.

With similar features to OpenHaptics, CHAI (Conti, 2007) provides a rapid prototyping system with interactively deformable soft-bodies and animated rigid bodies that include haptic interaction and graphics rendering capabilities. Finally, Ogre Haptics is built using ODE, OPCODE (see Physics libraries section), and Ogre 3D; an open-source scene graph API (Junker, 2006). Example application screenshots of CHAI and Ogre3D are provided in Figures 5 and 6, respectively.

2.2. Scene graph APIs

VR surgical simulations must render 3D scenes effectively. Typically scenes contain a number of models where each object must be rendered with different surface properties and lighting effects. This requires management of resources such as textures, shaders and other assets such as bump-maps or normal-maps. Although it is possible to manage these assets and tasks manually, there are a number of tools which can reduce the development effort needed and efficiently manage rendering the 3D scene that have been well tested and thereby avoid time-consuming trouble-shooting and more rigorous testing.

A scene graph is a tree or graph data-structure that stores a set of assets used to render a scene (models, matrix-transforms to locate objects relative to one-another, textures, shader programs, etc). The use of a scene graph is essential for scenes consisting of large numbers of objects, especially when only a small fraction of objects from the entire scene are visible at any given time. Scene graphs optimise rendering operations by employing fast sorting and searching algorithms to perform tasks such as occlusion culling, z-sorting, and batched render calls to render scenes more efficiently (by, eg, minimising resource switching). However, existing scene graph APIs such as NVIDIA's Scenix (NVIDIA Corporation, Santa Clara, CA), or Open Scene Graph (Reiners and Müller, 2002) do not provide any

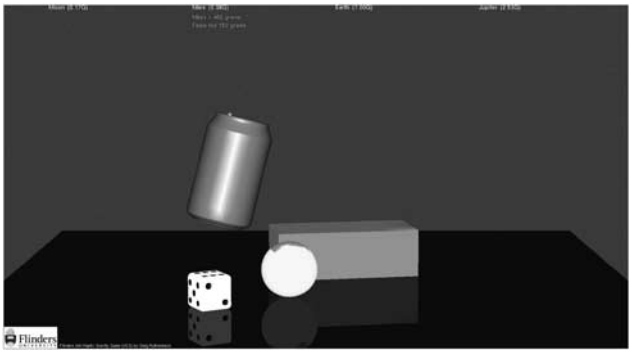

Figure 5 A VR simulation that simulates gravity on different planets to teach the difference between mass and weight using haptic interaction that was built using CHAI by the author.

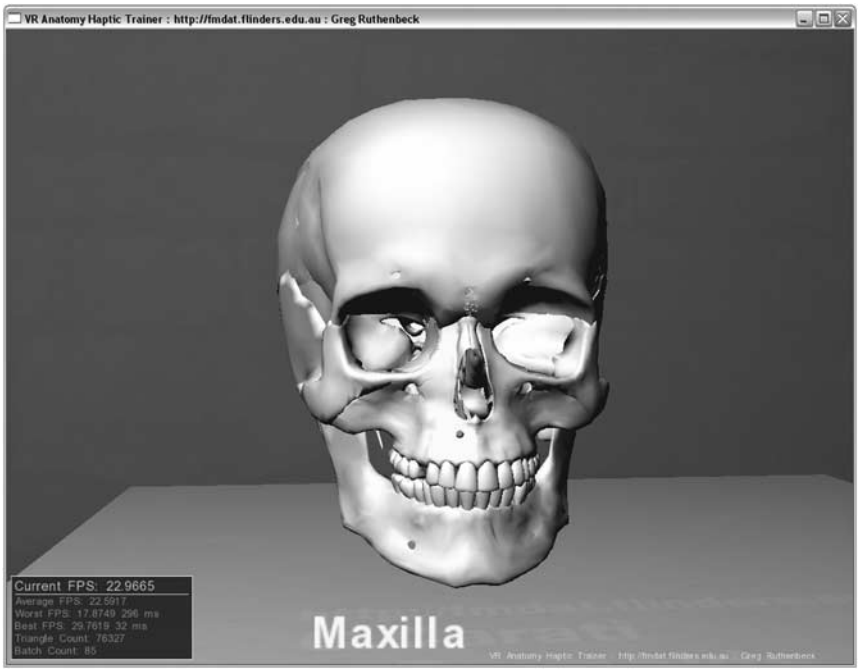

Figure 6 A VR Skull Jigsaw built using Ogre Haptics by the authors.

support for interactive tissues. Tissue simulation components can be integrated, but the scene graph's encapsulation of low-level data sets such as vertex-buffers make it difficult to integrate soft-tissue simulation components that must update vertex positions efficiently. Further, haptic rendering requires fast access to model geometry at the vertex-level, rather than object-level access that is commonly optimised in scene-graphs.

2.3. Graphics rendering libraries

Scene graph APIs and Medical Simulation APIs both provide simplified interfaces to the Graphics Rendering library. In doing so, many of the complexities of programming the graphics rendering algorithms and managing the data exchanged between the simulation application and the graphics hardware are conveniently abstracted away for the developer.

Graphics rendering libraries provide an interface to the graphics rendering pipeline that provides hardware accelerated processing of digital assets (typically models, textures, and shader programs) into the final image that is rendered on-screen. Without the assistance of higher-level libraries such as a scene graph API, the developer must manage the state of the graphics hardware. This is quite complex as it involves setting up vertex and index buffers of appropriate formats, compilation of shader programs, binding of shader program parameters, and generally conforming to the requirements of specific sub-versions of both the graphics rendering API itself, and the graphics processor that is installed on the system running the simulation software.

There are two widely used Graphics Rendering APIs: (1) Microsoft's DirectX and (2) OpenGL. Both began as fixed-function graphics pipelines designed to use triangular meshes that are transformed, and rasterized before the final pixel colour is computed and displayed on the screen. In the past five years, both libraries have become increasingly flexible largely by adding features and freedoms to the shader programming languages they support. Hence, numerous lighting techniques can be performed in real-time using these libraries including ray-tracing and other approaches that do not necessarily make use of the rasterization processes of the conventional graphics pipeline. Moreover, this makes new approaches to rendering possible in shader programs.

General purpose programming of the graphics processing unit (GPU) is being used increasingly to improve the performance of rendering, collision detection, and physics-based animation. When OpenGL is used for rendering, OpenCL is commonly used to implement GPU-accelerated features. Similarly, Compute Shaders can be used to GPU-accelerate parallel algorithms when DirectX is used. Finally, NVidia's CUDA (Compute Unified Device Architecture) is the most mature and feature-rich GPU programming language but is limited to running on NVidia GPUs. If AMD/ATI graphics cards are being used, then either OpenCL, or the lower-level AMD/ATI GPU programming toolset referred to as Close To Metal (CTM) may be used.

OpenGL and DirectX are very similar. DirectX is limited to running on the Microsoft Windows operating system,

whereas OpenGL is cross-platform. DirectX is, however, more advanced in its ability to efficiently run across multiple threads via its asynchronous rendering operations in our experience. Even when an unconventional graphics rendering approach is used, these graphics APIs provide a convenient interface for accessing the high-performance features of the graphics hardware, particularly the screen-buffers.

2.4. Physics-based animation libraries

The key interactions in surgical simulation involve real-time modelling of interactive deformable structures. Capturing the behaviour of interacting rigid bodies (eg, surgical instruments) and articulated bodies (eg, blood vessels) is also required to improve realism and increase immersion. Physics libraries provide these features via simplified interfaces, and hence are useful tools for enhancing VR simulations with interactive physics-based effects with minimal development effort.

PhysX was originally developed by Ageia (Santa Clara, CA) as the first physics API optimised to run on specialised parallel-processing hardware referred to as the PhysX Processor. In 2006, with the release of CUDA, a new version of PhysX that executes on the GPU was released. PhysX is based on technology developed by Novodex AG. Today, it is a popular physics engine used in interactive entertainment. PhysX is now owned and maintained by NVIDIA Corporation.

PhysX is capable of handling large numbers of rigid bodies and articulated bodies. Moreover, it supports real-time interactive cloth simulation with variable simulation characteristics. Most importantly (for surgical simulation), it is capable of simulating fluids and deformable soft bodies. All features can interact with each other and the user, though in testing this can be critically limited for certain types of interactions important to medical simulation. In PhysX soft bodies can be simulated using either volumetric meshes, or shell meshes. These volumetric meshes are based on tetrahedra. This type of soft-body simulation exhibits more realistic mechanical characteristics compared with shell meshes. However, the structure of the volumetric-mesh is fixed; it cannot be cut like the shell mesh. Shell-mesh-based soft bodies can use an internal gas pressure constraint to improve the behaviour. Without this constraint the mechanical behaviour is significantly less realistic if used to simulate living tissue. However, the gas pressure constraint cannot be used where the object may be cut or ablated (as is often required in surgical simulation). PhysX supports collision detection and handling of the entire scene. Unfortunately, experiments with PhysX show that collisions between soft-bodies or cloth with other soft-bodies or rigid-bodies are not detected sufficiently reliably for haptic interaction. Even with careful tuning of vertex-spacing in colliding object pairs, objects may pop through one another.

Since neither approach to soft-body simulation supports key interactions, PhysX has limited utility in medical simulations. PhysX does, however, have much to offer in managing secondary objects in the simulation. For example, PhysX can simulate the surgical cloth draped over the patient, or the behaviour of tubes or wires connected to instruments. These are not of critical importance though they may improve immersion. The question for developers remains: Does PhysX provide enough useful functionality to justify the time required to integrate it into the simulation? The answer depends entirely on the application.

Support in other physics APIs for GPU accelerated calculations is growing; a critical feature if cloth or deforming soft bodies are important. Havok Physics (produced and maintained by Havok, an Irish company owned by Intel) has features relevant to surgical simulation that are on par with PhysX. The open-source project Open Dynamics Engine (ODE) has seen declining support and limited growth over recent years. Conversely, Bullet Physics (Coumans, 2011) is an open-source project that is growing rapidly and already includes GPU accelerated features.

OpenTissue is a collection of works maintained by the Datalogisk Institut på Københavns Universitet (DIKU) (Department of Computer Science at the University of Copenhagen). The collection is quite diverse and includes some interesting works on elastically deformable solids, fluid simulation, and collision detection. The source code of demonstration applications is provided, although documentation can at times be sparse. These works may be useful stepping stones, although reviewing what exactly is on offer can be time consuming.

In summary, there are a number of physics APIs with growing feature sets that typically cater to the requirements of interactive entertainment and open-world simulations. Depending on the functionality sought, these APIs can provide developers with ready access to implementations of optimised physically based simulation algorithms. Unfortunately, none of these APIs are targeted directly at surgical simulation development. This is particularly evident in soft-body intersection handling, which is either not supported or is insufficiently reliable for haptic interactions that are central to VR surgical simulations. Hence, the use of physics APIs in medical simulation is limited to providing supporting effects and capabilities rather than the support for the core interactions.

2.5. Collision detection libraries

Collision detection and computation of the collision response are among the most processor-intensive tasks that must be handled in real-time VR simulations. Algorithms, techniques, and libraries have been developed to reduce the amount of processing required while maximising the reliability of detection and realism of the collision response.

Collision detection is the process of identify intersecting objects, and often the geometric primitives (tetrahedra, triangles, edges and lines or points) that intersect. Details of precisely which primitives are colliding are commonly required to compute an appropriate collision response such as the rebound trajectory of a bouncing ball. Once collisions are detected, a collision response must be computed. Details of the intersecting primitives and other data (eg, the location of the collision relative to the centre of mass, friction models etc) are used to un-intersect the models without causing artefacts (eg, jitter, popping etc), deform and deflect surfaces, and exchange kinetic energy between colliding objects.

Collision detection in surgical simulations is particularly challenging because models are typically deformable and densely spaced. Deforming objects can fold, and folds can result in self collisions, which are contacts between different parts of the same model. The number and complexity of objects in a given volume of surgical scenes gives rise to more collisions than open virtual worlds. Hence, not all algorithms employed in more sparse scenes are suitable.

Collision detection is typically performed in two stages: a broad-phase pass identifies intersecting volumes potentially containing intersecting objects for the second pass, and a narrow-phase pass that identifies individual primitives that are intersecting. Developers of VR medical simulations can choose to develop collision-detection systems from scratch, they can use the collision API from a physics API or, they can use a special-purpose collision API.

Collision detection and response are pre-requisites of most physics simulation capabilities. However, even though Physics APIs include these capabilities, the software interfaces to enable collision detection to be performed by the developer are often simplified and do not provide access to the types of data required to support, for example, tactile feedback. Therefore, physics APIs such as PhysX, Havok, and ODE, while containing excellent collision detection and collision response capabilities, do not provide access to the low level algorithms surgical simulation developers require. OPCODE is a small collision-detection library developed by Terdiman in 2001 (Terdiman, 2001). The library uses a bounding volume hierarchy based on an axis-aligned bounding-box (AABB) tree. OPCODE is optimised for minimal memory usage. It is capable of fast detection of triangle-triangle collisions on conventional hardware. However, OPCODE is not optimised for fast updates to the AABB tree. Nor is it designed for use on parallel hardware such as GPUs. This limits its attractiveness for use in medical simulations since any data structure employed for accelerating collision detection must be capable of efficient updates to mesh changes because meshes in medical simulations are typically non-rigid.

The SOLID collision-detection library was originally developed by van den Bergen in 2001 (Van den Bergen, 2004). It uses an iterative algorithm for computing the distance between objects that was originally described by Gilbert, Johnson and Keerthi in 1988 (Gilbert *et al*, 1988). SOLID includes optimisations for fast updates to deformable solids (Van den Bergen, 1998). It also supports fast penetration depth estimation (Van den Bergen, 2001), which is essential for haptic interaction and computation of reactive forces. SOLID is optimised for execution on the CPU but it does not natively work using polygonal meshes. Instead, objects are internally represented as primitive shapes and complexes of polytopes. To overcome this limitation the authors suggest that another library (Qhull (Barber *et al*, 1996)) be used to decompose complex objects into a compatible format. Clearly, while SOLID has promising functionality well suited to application in medical simulations, it is not ideal. RAPID is a research project developed at the University of North Carolina based on oriented bounding box trees (Gottschalk *et al*, 1996). It recursively subdivides polygonal objects and groups the subdivisions for fast collision tests. One criticism raised by Terdiman (2002) is the relatively large memory footprint when compared with OPCODE (Terdiman, 2002). Terdiman addresses these deficiencies by re-engineering RAPID into another library named Z-Collide (Terdiman, 2002). Z-Collide removes some redundant data stored in tree nodes and replaces the matrices used to represent rotations ($3\times3=9$ elements) with normalised quaternions (4 elements). Neither RAPID nor Z-Collide are optimised for deforming objects. Finally, V-Collide 'combines I-COLLIDE's sweep "n" prune with RAPID' (Hudson *et al*, 1997). H-Collide is a more recent collision detection library optimised for use in haptic applications (Gregory *et al*, 2005). It uses a hybrid hierarchical representation that uses spatial partitioning to separate objects occupying large areas into a hash-table for efficient lookups, an oriented bounding box tree (OBB-Tree) within hashed volumes to accelerate lookups of small sets of potentially intersecting primitives, and frame-frame coherence to accelerate lookups based on previous results (since changes between frames are minimal). H-Collide has been used to produce haptically interactive applications based on rigid objects. However, like most of the existing libraries, it is not optimised for deforming structures where the additional processing overhead of updating the collision detection system at run time is not considered.

2.6. GPU-based collision detection

The trend toward GPU-based simulation components (eg, GPU physics-based animation and GPU collision detection) further complicates development of VR surgical simulations by forcing consideration of the memory bandwidth usage between the CPU and GPU. For example, if a soft-body simulation is performed on the GPU, and the collision detection on the CPU, then the model would need to be copied into CPU-readable memory at least once every visual update (30–60 Hz), and at most once every haptic update

(1 kHz). Hence, even if the GPU-based software component is extremely fast, it may introduce a bottleneck into the simulation where the CPU must wait for data to be copied from the GPU.

Tang *et al*'s GPU-based collision detection system uses refitting algorithms to efficiently update a bounding volume hierarchy tree (BVH tree) on the GPU. The tree is then traversed to identify intersecting bounding volumes before narrow phase collision tests are performed. Additionally, they describe a locking-free mechanism for sharing the collision results with the CPU asynchronously; an important feature for good multithreading and thereby achieving full utilization of modern multi-core processors.

Lauterbach *et al*'s 'gProximity' focuses on data-parallelism (*versus* task-parallelism) for fast construction, updating, and traversal of a variety of types of BVHs to achieve 'over an order of magnitude performance improvement over prior GPU-based' collision detection algorithms' (Lauterbach *et al*, 2010). Further, it also has the ability to efficiently perform distance queries, which can be useful for medical simulation applications that, for example, only wish to perform haptic rendering of models within a certain range of the haptic instrument model.

Pabst *et al* describe a fast GPU-based collision detection system for rigid or deformable surfaces. Their approach uses Teschner *et al*'s spatial hashing scheme, specialised for the GPU using LeGrand *et al*'s approach; the hashing simply uses a box-grid-index that has spatial relevance (Pabst *et al*, 2010). Hash values are sorted on the GPU, and once colliding box-pairs are identified, the box contents (triangles) are tested for collisions (also on the GPU) before being output to the CPU for collision response processing.

However, it is worth noting that the volume of intersection is required for some haptic rendering algorithms and it is not provided by any of these collision detection libraries (although it can be computed once colliding (intersecting) parts of the models are identified). A finely tessellated 3D mesh (composed of small triangles) when 'collided' with another similar mesh with these libraries (both CPU and GPU-based) will identify intersecting triangle-pairs. When models overlap sufficiently, it is difficult to expand the identified colliding triangle set to include those that are completely within the other model. In summary, conventional collision detection systems will readily identify intersecting surface elements (typically triangles) in a contour of intersection, but not the triangles that are completely within the intersecting model. Although workarounds exist, such as using the colliding triangle history, this exemplifies the types of subtle differences between mature tools for computer games, and the requirements of VR surgical simulation.

2.7. Haptic libraries

Haptic devices are commonly used to control VR surgical simulators because they allow virtual surgical instruments to be controlled using the same movements as used during real surgery; further, the tactile feedback enables users to feel tissue boundaries, features, and anomalies that can be critical for informing where to cut and the users interactions more generally. Haptic rendering is the process of computing the reactive forces to be delivered by the haptic device. Each haptic device manufacturer has its own software interface library (refer to Table 1). Some of these libraries also provide solutions for performing haptic rendering of a variety of (generally simple) effects. However, in most cases, VR surgical simulation developers will need to develop their own haptic renderers that work effectively with their specific tissue simulation solution. Physics libraries can be used to simplify development of certain types of haptic renderers, for example Virtual Proxy haptic rendering (Ruspini *et al*, 1997) relies on a spring connected to a proxy point that collides with model surfaces. The physics libraries outlined above can be used to simulate the spring and also handle the collision. However, where pop-through is an issue, the collision detection and collision response libraries (outlined above) may provide a better solution.

3. Conclusions

Surgical simulation and medical part-task or procedural trainers that rely on haptically enabled interactions with soft-tissue can, to some extent, be developed with existing tools.

Table 1 Haptic libraries

Author	*Library*	*Summary*
SensAble	HD (Open Haptics)	Low-level device interface
SensAble	HL (Open Haptics)	Haptic rendering in OpenGL of rigid
SensAble	QuickHaptics	Haptic VR API
Novint	HDAL	Low-level device interface
Force Dimension	Haptic SDK	High and Low level device interfaces
Open Source	CHAI3D	Device agnostic abstraction layer for all common haptic devices
Open Source	OgreHaptics	Plugin for Ogre3D that adds rigid-body haptics (compatible with Sensable devices)
Open Source	HaptiCast	Haptic VR API using Irrlicht/Newton Physics/Open Haptics

However, the perfect tool for developing VR medical simulations does not exist. There are many that contribute different features and capabilities, but none is ideal. SOFA and H3D are excellent starting points. However, in order to produce a flexible and efficient solution, a range of techniques and technologies may need to be combined depending on the specific interactive tissue experience that is sought. Doing so efficiently is especially challenging particularly when supporting haptic interaction.

There are caveats to using existing software libraries that can retard the development of features specific to medical simulation development. With careful consideration and design, combining existing components carefully can produce higher quality simulations with far less developer effort. Re-use of existing libraries will enable researchers and developers alike to focus their efforts to make increasingly more specialised contributions, thus enabling rapid improvement in the quality of VR medical simulators.

References

Bacon J *et al* (2006). The surgical simulation and training markup language (SSTML): An XML-based language for medical simulation. *Medicine meets Virtual Reality 14: Accelerating Change in Healthcare: Next Medical Toolkit* **119**(1): 37.

Barber C, Dobkin D and Huhdanpaa H (1996). The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software* **22**(4): 469–483.

Cavusoglu MC, Goktekin TG and Tendick F (2006). GiPSi: A framework for open source/open architecture software development for organ-level surgical simulation. *IEEE Transactions on Information Technology in Biomedicine* **10**(2): 312–322.

Conti F (2007). Chai 3D: An open-source framework for haptics and dynamics simulation. In James D *et al* (eds). *Medicine Meets Virtual Reality 15 - in vivo, in vitro, in silico: Designing the Next in Medicine 2007*, IOS Press: Amsterdam/Berlin/Oxford/Tokyo/Washington, DC, P 528.

Cotin S *et al* (2005). Collaborative development of an open framework for medical simulation. MICCAI open-source workshop, Copenhagen.

Coumans E (2011). Bullet physics library. Retrieved 31 October 2011, http://bulletphysics.org/wordpress/.

Faletti G and Vezzadini L (2007). NeuroVR: An open source virtual reality platform for clinical psychology and behavioral neurosciences. Medicine Meets Virtual Reality 15. J. D. Westwood. Newport Beach, CA, IOS Press, p 394.

Gilbert E, Johnson D and Keerthi S (1988). A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation* **4**(2): 193–203.

Gottschalk S, Lin M and Manocha D (1996). OBBTree: A hierarchical structure for rapid interference detection. *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, San Diego, CA, ACM New York.

Gould DA *et al* (2011). Simulation: Moving from technology challenge to human factors success. *Cardiovascular and Intervational Radiology* **35**(3): 445–453.

Gregory A, Lin M, Gottschalk S and Taylor R (2005). A framework for fast and accurate collision detection for haptic interaction. *SIGGRAPH*, Los Angeles, CA, ACM.

Hudson T *et al* (1997). V-COLLIDE: Accelerated collision detection for VRML. *Proceedings of the Second Symposium on VRML*, Monterey, CA, ACM.

Junker G (2006). *Pro OGRE 3D Programming*. Apress, Springer-Verlag. New York, NY, USA.

Lauterbach C, Mo Q and Manocha D (2010). gProximity: Hierarchical GPU-based operations for collision and distance queries. *Computer Graphics Forum* **29**(2): 419–428.

Montgomery K, Bruyns C, Brown J, Sorkin S, Mazzella F, Thonier G, Tellier A, Lerman B and Menon A (2002). Spring: A general framework for collaborative, real-time surgical simulation. *Medicine Meets Virtual Reality 02/10 Digital Upgrades, Applying Moore's Law to Health* **85**(1): 296.

Pabst S, Koch A and Straßer W (2010). Fast and scalable CPU/GPU collision detection for rigid and deformable surfaces. *Computer Graphics Forum* **29**(5): 1605–1612.

Reiners D (2002). OpenSG: A scene graph system for flexible and efficient realtime rendering for virtual and augmented reality applications. PhD thesis, Technische Universitat Darmstadt, Fachbereich Informatik, 2002, Fraunhofer Publica: Birlinghoven, Germany.

Ridge R, Hudson D and Loach R (2011). The Tao framework. Retrieved 31 October 2011, http://sourceforge.net/projects/taoframework/.

Ruspini DC, Kolarov K and Khatib O (1997). *The Haptic Display of Complex Graphical Environments*. ACM Press/Addison-Wesley Publishing Co: Reading, MA.

SOFA project—(C) INRIA, M., 2008 (2009). SOFA:: Home. Retrieved 09/11/2009, http://www.sofa-framework.org/home.

Terdiman P (2001, May 2009). Memory-optimized bounding-volume hierarchies. http://www.codercorner.com/Opcode.pdf.

Terdiman P (2002). RAPID hack. Retrieved 13 November 2009, http://www.codercorner.com/RAPID_Hack.htm.

Terdiman P (2003). OPCODE—optimized collision detection. Retrieved 17 July 2008, http://www.codercorner.com/Opcode.htm.

Van den Bergen G (1998). Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools* **2**(4): 1–13.

Van den Bergen G (2001). Proximity queries and penetration depth computation on 3D game objects. *Game Developers Conference*.

Van den Bergen G (2004). SOLID—software library for interference detection. Retrieved 17 July 2008, http://www.win.tue.nl/~gino/solid/index.html.

Villard PF, Jacob M, Gould D and Bello F (2009). Haptic simulation of the liver with respiratory motion. *Studies in Health Technology and Informatics* **142**(1): 401.

Received 9 December 2011;
accepted 6 November 2012 after one revision