

APPLICATION FRAMEWORK FOR *CANVAS* THE VIRTUAL REALITY ENVIRONMENT FOR MUSEUMS

Hank Kaczmarski, University of Illinois, Urbana-Champaign, Illinois USA

hank@isl.uiuc.edu

Camille Goudeuene, Benjamin Schaeffer, Lance Chong, Rose Marshack, Lee Hendrickson, Jim Crowell, University of Illinois, Urbana-Champaign, Illinois USA

cog@uiuc.edu, schaeffr@isl.uiuc.edu, lchong@ncsa.uiuc.edu, rose2000@uiuc.edu,

spiralinaspiral@gmail.com, jimc@uiuc.edu

Abstract

The technical specifications of *CANVAS*, the Collaborative Advanced Navigation Virtual Art Studio, were described in a paper presented at the EVA 2005 conference in Firenze [1]. We take this opportunity to describe the application framework that currently exists to allow artists to develop electronic media projects in *CANVAS* and how the increasing number of 3-D datasets being gathered by museums can be displayed in *CANVAS*. We also describe a case study of the issues involved in creating one narrative media art project *360* within the confines of the existing toolsets available to potential users of *CANVAS*.

When we set out to create an immersive visualization space for museum galleries, our experience with large-scale, complex hardware and software intensive virtual reality displays environments told us that every modification to simplify the system for artists to create applications or stabilize the display for audiences to interact singly or in groups carried significant baggage that would affect multiple aspects of the resultant experience. The cost and complexity of a virtual environment can be reduced in many ways: smaller and fewer displays, simpler or omitted motion tracking, simpler user input devices, simpler audio hardware. Naturally, issues arise when designing a VR application for simplified hardware, or porting one to simpler hardware than what it originally used in legacy applications.

Display hardware can be much of a virtual environment's total cost, so it is an obvious choice for economizing. Omitting screens on the ceiling or floor also simplifies installation in rooms with no oversize ceiling. Applications often do not need ceiling or floor screens, since people find it more comfortable to move neck and eyes horizontally. Horizontal field of view is also more critical than vertical for VR applications. A visualization that lets the user walk entirely around an object, viewing it from all angles, needs to be redesigned for a three-wall 270-degree display. For a single-wall display the redesign may even replace walking around with rotating the object in place "behind" the wall, like a keyboard-and-mouse interface. A physically smaller display, like a two-meter rather than a three-meter room, may suffer more from projector "hot spots" on the screen as the user walks around. This varying brightness may be compensated for by reducing the application's dependence on visual dynamic range. (This may already have been done, since projectors have less dynamic range and color sensitivity than desktop

displays.) However, by reducing the size of the viewable space, one converts what was a room-sized experience into a less interactive experience, a price normally too high to pay.

Motion tracking can be another expensive part of a virtual environment. A tethered system costs less than a wireless system, but in practice it forbids a display that entirely surrounds the user. In other words, it enforces a “front” orientation in the environment. Applications intended for a fully-immersive cube display (astronomy, architectural walkthroughs) will need different user controls when the user generally faces one direction instead of turning freely. Conversely, applications which began on a desktop computer will port more easily. In *CANVAS*, as the design presupposed a multi-user experience, so individual viewer tracking is considered a nuisance as a rule, but is permitted in the basic system design for exhibits which might benefit from individual spatial navigation.

For user controls (or “user interface”) most cave-like environments use a wand. Early wands included only a few controls, i.e., buttons and a joystick. Since then, one trend is to use handheld controllers from video games, “gamepads.” These have several joysticks, sliders, and perhaps a dozen buttons. An application designed for such a rich input device may need serious redesign for a simpler wand. The informal literature on videogame design may be helpful here, particularly discussions of porting games from a desktop computer’s large keyboard to the dozen keys of custom “platform” hardware like PlayStation or Nintendo. Game designers prefer dropping features to adding submenus and other baroque command structure. Another trend in wands is simplification: a motion tracker with only one button (demanded by Macintosh users?). Porting a one-button application to a more powerful motion-tracked gamepad seems trivial. But we find that the gamepad’s extra nonfunctional buttons and joysticks perplex users, particularly new users. It may be worth buying a second, simpler, wand. It may even be worth adding features so the extra buttons do something.

Removing motion tracking entirely from the wand is an extreme simplification. Some would argue that this changes VR into just a large display. Porting an application that used a tracked wand to such an environment is almost the same as porting to a desktop PC. Generating 3DOF or 6DOF gestures from 2-D joysticks is awkward—just ask any CAD-software user. *CANVAS* does not require or prevent multiple modalities of device tracking; a laser pointer is supported by aiming video cameras at the rear of the screens to track wavelength-filtered light, traditional pulsed-DC, AC or infrared trackers can be easily integrated if an exhibit benefits from the significant added cost of such devices.

Removing head-tracking can be far more tolerable than removing wand-tracking, though. Head-tracking is missed most when the user manipulates objects rather than, say, navigating through a large world. When several people are viewing a scene, head-tracking may even be undesirable. *CANVAS* permits the tracking of not only the viewer’s head but any number of one or multiple viewers hands, feet or torso if the deployed tracking system is so configured.

Spatialized sound is a significant sensory addition to an immersive space. Audio sources can be tightly attached to positions in the virtual space for a head-tracked user wearing headphones. Giving up head-tracked headphones, or equivalently having multiple persons, practically means that audio can be attached only to broad regions of space. This is acceptable if the primary purpose of audio is confirming user commands or alerting the user to changes. Loudspeakers, even an array of 4 or 8 loudspeakers, also

cannot compete with the precision of tracked headphones. But they are much friendlier for a multi-user installation. Even headphones which do not block outside sounds still discourage users from talking among each other.

Syzygy – Middleware for *CANVAS*

While PC clusters are radically cheaper than special purpose VR hardware, they do have disadvantages, the most important of which is the added complexity of managing the system and writing software for it. Consequently, for a PC cluster to be a viable platform, special middleware must be created that will simplify both of these tasks. This paper describes one such solution, Syzygy, an open source middleware solution for PC cluster VR. While, in its initial incarnation, *CANVAS* runs a Syzygy cluster, we mention other software solutions for such an environment and note that the list grows as the popularity of immersive spaces crosses continents. We focus on three areas: a distributed operating system (Phleet) that helps applications running on the cluster cooperate the hardware and one another, low-level APIs for synchronizing graphics displays running on different computers, and high-level APIs for writing cluster applications.

Syzygy contains the Phleet distributed operating system that portably (across Windows, Linux, Mac OS X, and Irix) coordinates software running on our clusters. In this system, a central szgserver manages all the other Phleet-enabled software. In particular, it centralizes configuration of the system. Instead of having configuration files scattered on each computer, the szgserver stores all the information in a central database, and through the Phleet APIs, programs query that database over the network to configure themselves upon start-up. This is a substantial win for cluster administration, especially for Windows machines that are awkward to remotely administer. This configuration database is stored for each Phleet user, making the cluster a true multi-user environment with each person able to have her own data, executables, and application preferences.

Applications running in a cluster environment are really collections of individual programs (here called application components) that send data to one another over the network. Consequently, in addition to configuration, Phleet must also coordinate how application components connect to one another, for instance telling a component that wants to display graphics information the location of another component offering it. Furthermore, applications must start up and shut down in an orderly fashion. To enable portable remote execution, Phleet includes szgd, a daemon program that launches Syzygy executables in response to Phleet messages produced either by command line tools or one of Syzygy's programming frameworks. As an application launches, it uses the Phleet API to scan the cluster, stopping services that it does not need (or that are incompatible with its operation) and starting services, such as sound players or input device drivers, it does need (using the szgd's running on cluster nodes), along with launching any specific application components of its own. In reverse, Phleet also allows for orderly application shut down, making sure all cluster resources are freed and that each application component exits cleanly.

When using a PC cluster for visualization, one of the most important considerations is how the displays will be synchronized. At the high end, the vertical refresh of each video signal occurs at the same instant using a process called genlocking, which is supported by high-end graphics cards. This is necessary for PC cluster active stereo, where a 3D image

is produced by alternating right eye and left eye images while the users wear LCD shutter glasses that blank their lenses in time with the video signal, making sure that the user's right eye is covered when the left eye image is visible and vice-versa. However, genlock is not required for passive stereo, where the right and left eye images are simultaneously projected onto the same surface, with (say) different polarizations, and users wear cheap filter glasses to make sure that each of their eyes sees only the appropriate image. Using this technology, a coherent stereo image can be seen across the whole cluster using very inexpensive consumer graphics cards.

In addition to genlock, we consider framelock, the property by which given corresponding frames of 3D graphics are displayed at the same time on the cluster nodes. This is important since different views of a scene (which is exactly what we see on the cluster nodes) can be drawn at different rates. Without framelock, upon (say) user navigation, the more quickly drawn views will move before the more slowly drawn, causing an objectionable tearing at the display boundaries. This is an application-level property (since only the application knows when it is going to produce another frame) as opposed to genlock, which concerns the video signal. Here, we can actually do a good enough job in software by creating a synchronization API (as exists in Syzygy) that operates over the cluster network. In this case, once each cluster node is ready to display its new frame, it communicates with the others and they all display them exactly when everyone is ready. This is not a perfect solution without genlock since the new frame will be displayed on the vertical refresh of the graphics card, implying that perfect synchronization requires genlock. However, practice shows that software framelock along with passive stereo produces good enough (actually quite good) results, and, when combined with its inexpensive nature, make this a winning technology combination for PC cluster visualization.

At a higher level, rendering synchronized images across the PC cluster requires synchronizing the data rendered on each PC in coordination with synchronizing when that data is rendered (as has been discussed already with genlock and framelock). This requirement reaches into application architecture. On the one hand, we can try to design an API that is transparent to the programmer. This method's advantage, hiding the data synchronization from the programmer, is also its disadvantage, forcing the programmer to write to a particular API and preventing her from constructing clever (application-specific) methods of sharing data. Consequently, with PC cluster visualization, it is important to have a range of options available: different tools will be more appropriate for different jobs. We first consider different ways data sharing can be made transparent to the programmer. Next, we discuss Syzygy's distributed scene graph, followed by a description of its API for constructing distributed applications that explicitly handle the data sharing they require.

There are several different types of data that an API could transparently share. For instance, the raw pixels comprising the scene could be rendered centrally, sent across the network, and displayed on each cluster node. The APIs to do this are essentially found directly in the OS's windowing subsystem and thus are straightforward to implement. However, this approach is undesirable in many circumstances because of the extremely large amount of pixel data and the very limited networking bandwidth between cluster nodes. It might make sense in special cases, such as distributed ray tracers, where the application is already sending pixels, but makes much less sense with an OpenGL

application, especially when the cluster it runs on has a slower network (as might very well be the case if we are trying to save money in cluster construction).

At a higher level, we could distribute the OpenGL commands themselves. Well-known systems like Chromium take this approach. They modify the system OpenGL dynamic library to capture OpenGL commands, which are buffered and sent across the network for display by a specialized program. The advantage to this approach is that it works with any OpenGL program, but the disadvantage is that OpenGL is a very communications-intensive API, which implies that Chromium has the potential to severely stress the cluster network. Consequently, like in the previous case, this class of APIs demands a fast cluster network and is possibly liable to network bottlenecks.

At a higher level still, we can focus on sharing the semantics of a virtual world via a scene graph API. Here, all the information required to render the world is stored in a database (the scene graph), which the application modifies to manipulate objects, navigate around, or perform other tasks. Because the scene graph can be drawn continuously in a background thread (technically speaking, it is “retained mode” to OpenGL’s “immediate mode”), this style of API is ideally suited for interactive manipulation at an interpreter, like Python (as in the Syzygy distributed scene graph) or Smalltalk (as in Squeak). Freeform manipulation at the interpreter prompt allows rapid prototyping of virtual worlds and, thus, is a very valuable system property. To make a scene graph API transparently distributed, we must simply have the ability to transfer total scene graph state from one application to another and afterwards transfer any update messages. Normal application operation, then, just generates scene graph changes, which are potentially much smaller than the entire scene and, consequently, much easier on less powerful networks. Because of this, several well-known scene graphs APIs have been adapted for distributed virtual environments, where participants each work on a shared scene graph at geographically dispersed locations. For instance, Avango extends SGI’s Performer and Distributed Open Inventor extends (obviously enough) SGI’s OpenInventor.

Syzygy takes these ideas and adapts them to PC cluster graphics. The main difference between the PC clusters and distributed virtual environments is the greater level of synchronization required for the PC cluster case. Here, we must also be worried about when the updates are processed on the cluster nodes in addition to worries in the DVE case that each computer receive the same update stream. At each frame on each cluster node, we want the same updates to have been processed by their scene graphs. Consequently, the Syzygy distributed scene graph application synchronizes itself with the frames produced on the cluster nodes. As the nodes are drawing the next frame, the application buffers its scene graph updates. Once the nodes have all finished drawing the frame, they simultaneously display it and request the application send them the updates it has buffered. Each node now receives the same update sequence, which it applies to its local scene graph before beginning to draw the next frame. In this way, each node draws a view of the same world.

Sometimes it is easier to explicitly add data sharing to an application than to make the application use an available programmer-transparent API. For these cases, Syzygy offers its master/slave framework. Here, an instance of the application runs on each cluster node. These applications operate using a well-defined event loop. At the loop’s start, the master application instance processes data from input devices or other sources and

changes its internal state. The portion of application state that affects rendering is then automatically pulled from memory, packed into a message, and sent to the slave application instances. The slaves then unpack the message and use it to update their internal state. Everybody now draws the scene, synchronizing before displaying the frame, and then repeats the event loop. The programmer explicitly designates what part of application state is shared. This method often allows easy porting of existing applications to a cluster environment, given source code.

Using *CANVAS* for Visualizing 3-D Data

In addition to electronic art created explicitly for CAVE™-like spaces [2], *CANVAS* has as a goal to be able to display and interact with complex 3-D (and higher dimensional) data sets, for example MRI, fMRI, CAT or camera-based scanned image data. Our software for viewing sets of fMRI brain images uses volume rendering rather than a slicer plane, but it still uses texture maps. The 3-D data is rendered on a stack of semitransparent parallel planes. We orient the planes to face the user, no matter where in the immersive space they or the brain are. Visual artifacts appear when the user moves inside the brain itself, but meaningful patterns are hard to see anyways from such viewpoints. To view the Visible Human data [3], we first interpolated and down-sampled the 3-D data so it could fit in a PC's memory. Much of this 600-line C++ program is devoted to building a semitransparent texture map from this data. This texture map is applied to a large square "slicer plane" attached to the wand. Lower-resolution texture maps are computed when moving the wand, to increase frame rate.

We designed a program for painting in space. Moving the wand while holding one button leaves a trail drawn like a ribbon. Holding another button erases any parts of a ribbon near the wand. Its simplicity is compelling because the artist can work from any angle, even inside -- almost sculpting rather than painting. Head-tracking is essential here. We first wrote it in a few hundred lines of C++, using Standard Template Library containers to efficiently split and merge ribbons during erasing and drawing. Avoiding fragmented ribbons is critical for display at high frame rates. A recent port to Python did not shrink the code size, but simplifies maintenance of the software. Most of the CPU use is in the OpenGL libraries, so this port did not reduce performance.

Design in *CANVAS* versus Design for *CANVAS*

In fields such as product design, interior design, and architecture, the so far well-known form of computer visualization, commonly called "computer-aided design" or CAD has been indispensable for more than two decades. As the computer hardware and software gets cheaper and better, and consumers, designers, manufacturers, and distributors get more savvy and sophisticated, the definition of CAD is about to take on new meanings. A successful design on one hand is a vessel of bringing users designer's ingenuity in aesthetics and functionality, on the other hand is a good solution in finding a balance among manufacturing, marketing and consumers. It is a very complicated process for one designer to work on, and is a major communication effort for a team or multiple teams in a cooperation setting. In design education, helping students to understand and practice in such a complicated process is also a major challenge. This is a

strong market demand that calls for variety of truly useful and efficient computer powered facilitating systems to be developed.

Because of its advantage in display and computing power, *CANVAS* as a visualization tool has remarkable potential. In a virtual environment, the interpretation of CAD could be extended to using computer visualization and simulation as a tool to assist most, if not all, aspects of the design process. In other words, computer visualization can be used not only to display the 3D appearance model of a design, but it can be helpful in cultivating a successful design concept starting from its research stage, option analysis, ideation, concept selection, manufacturing and costs evaluation, to usability and ergonomics simulation, marketing planning, and finally in the future, but certainly not the last step or the least important factors, social and environmental impact outlook. A virtual environment such as the *CANVAS* can be a perfect workbench for such a job. It will allow information from multiple links of the chain to be visualized and presented to a much larger audience/user group, and further more, it will enable every user to contribute input. It could potentially increase the speed of R&D process dramatically by bringing together the art and science of intuition and problem solving in design into every team member's arm's reach.

Other design related peripheral uses which could be seamlessly integrated into such an environment include remote collaboration, concept demonstration, and even visualized project management such as event scheduling and resource management. It is a time and effort efficient way to exchange information and ideas within a design team, or to a student even an outsider such a client. A Syzygy-driven *CANVAS* allows users to navigate through 3-D models created in popular CAD programs. In order to meet the needs of our design community, *CANVAS* will be constantly evolving, and being updated and redesigned by both the designers in this case who are also part of the end user group, as well as the computer scientists and hardware engineers, and thus ensure it is sound in functional, operational, and usability design.

VRML - Virtual Reality Modeling Language in *CANVAS*

VRML is a 3D format especially well suited for CAVE-like environments, from the most elaborate six-surface, fully-immersive spaces to the purposely simplified 3-wall environment of *CANVAS*. An artist is able to use VRML to create not only complex objects but also the world those objects inhabit as well as interactions between objects [4]. Compared to many other 3D tools, VRML was specifically designed to aid in the creation of and interaction with virtual environments [5]. Further adding to its usefulness is the fact that many pre-existing data sets can be easily converted for use in VRML-ready environments [6].

There is a long history (in technology standards) of VRML usage in the museum sciences. Examples range from virtual specimens to virtual museum tours and exhibitions [7,8,9,10]. Archival projects have produced models which can be readily explored using VRML [11]. Data visualization is another task VRML handles well [12, 13]. Thus, in a move towards the practical virtualization of a museum's assets, VRML is an attractive option.

Taking VRML from its 'native' environment of a web browser on a desktop PC into the immersive environment of a CAVE only increases VRML's effectiveness. In the

same way that a CAVE enhances many other 3D applications, so does the immersion increase the presence one experiences when interacting with VRML worlds or models. Furthermore, a CAVE provides powerful methods for interacting with and navigating VRML datasets that are simply not available on desktop PCs.

In *CANVAS*, a CAVE powered by Syzygy, VRML duties are handled by VRMLView, an application developed to visualize data in the VRML (and related OpenInventor) format [14]. VRMLView allows the user to configure how the VRML data is loaded, both in terms of visual representation and spatial orientation, and additionally how the user wishes to interact with the world or object. Navigation of the VRML world is handled by Syzygy's Scene Graph architecture [15]. A user can interact with VRML objects in terms of rotating, translating or scaling the object freely. VRMLView also supports the playback of any animations that may have been created in or for the world, such that a user could be automatically taken through a VRML scene.

Thus, VRML is a versatile tool for the creation of virtual worlds or models and *CANVAS* using Syzygy and VRMLView is a compelling environment in which to explore such worlds.

PYTHON AND MORE

Scripting languages like Python [16], Perl [17], and Ruby [18] are easier to learn and faster to program in than traditional C or C++. With these, we find that students without years of programming experience can prototype and design VR visualizations far more quickly. In one semester, a student can then learn details about their problem domain and about VR, instead of mastering the intricacies of C++ programming. Even our seasoned programmers benefit from the Python bindings we have wrapped around Syzygy's C++ interface. Faster programming improves the quantity of code, of course, but quality improves too. Inevitably, the initial prototype changes dramatically over a few weeks as new ideas occur and the final design emerges. Since less programming time is invested initially, the programmer is less reluctant to greatly rework or even abandon early designs. A better design results. Of course programs written in C and C++ tend to run faster (at a higher frame rate). But CPUs have become so fast in recent years that this is rarely a practical concern. CPU time is valuable, but not nearly as much so as programmer time.

360, A Case Study

As described in [1], *360*, a narrative art exhibit, is a three-walled, gallery-centered immersive room presentation of the life-stages of the romantic life of a couple. In its initial showing at the Krannert Art Museum in August 2004, *360* contained a timeline of the couple meeting in a café, with the gallery viewer wearing stereo glasses seated at a real table within eye and earshot of the couple at a virtual table nearby. Navigation through the life of the couple was afforded through a joystick. The seated position of the viewer played off of the decision to have the three graphics computers draw all images and spatialized sounds from a fixed head position. This initial exhibit involves MPEG-compressed video within a series of viewer-selectable computer-generated 3-D scenes, but the guiding principal of this project was to not make decisions that limited the ability

to expand the scope of the project over time. Even in the simple early stages of planning, it became obvious that this project would be a collaboration between artists and computer programmers.

At the first meeting of the multidisciplinary team formed to create *360*, the artists presented ideas about the "look and feel" of the project, pointed out some of the aesthetic ideas they liked about previous immersive space applications – the textured, layered, somewhat murky backgrounds of the dance application, and presented a movie of a fly-through of a computer-generated world; this world was full of interesting graphic icons, checkerboards, flying objects; it was a canned world, not generated on the fly, probably used Maya, and swept a camera through the model. The computer programmers in attendance noted that their world would be computed "on the fly" and thus could be much more dynamic than this world, more easily configurable, and more interactive, since our camera will be controlled in real time.

The presentation of graphics materials was most helpful in determining what the look and feel of the project should be. Next, the accompanying narrative was explained, and then how the story may be presented was discussed, both in a logical and storytelling sense, and then in a more technical sense; i.e., the mechanics of displaying an animation and how the programmers will technically interact with the world. At the end of the first set of meetings, media collection was deemed to be the responsibility of the artists; some would collect the graphics and textures, others would collect and edit sound and videos. Meanwhile, computer programmers would develop a framework for this world to sit in. Challenges later occurred with this approach as it was discovered that what an artist's eyes sees in a finished video is significantly different than what appears in the 3-D world after going through the hardware and software that integrate that video into the virtual world, necessitating re-shooting of video to pre-distort color balance and brightness/contrast/gamma levels.

Navigational Issues

The biggest problem in communicating between techies and artists was trying to explain the necessity of consistency in navigational systems. For instance, the joystick was either going to move us through the space, or move objects toward our position. It could not do both without a mode-switching mechanism, and there was a desire to keep the user interface as simple as possible, so mode-switching was out of the question. In addition, there was the problem of "picking." How shall we decide when an object should be "activated?" One suggestion was a proximity activator, which could allow a user interface which flew us through the space. As we approached a threshold near an object, the object could become activated and tell us its story, take us to another screen, etc. As we left the object, life would return back to normal (for some reason, this suggestion did not take.) Any other object picking modality required a button, or some sort of picking device, and many more complications, and it had to be explained that once a interface coding system had been developed, it needed to stay consistent, for programming sake and also for ease of use for the audience.

A Helpful Solution for Artists

A proposal was made to develop a text configuration file, where the artists may write along the lines of:

Shape (color, size, x-coordinate, y-coordinate, z-coordinate, texture.jpg)i.e.,

Box (Red, 20, 5, 10, -3, myTexture.jpg)

Sphere (blue, 10, 3, 6, 10, anotherTexture.jpg)
Movie (white, 10, 4, 8, 22, myMovie.mov)

This would allow non-programmers to place objects in the file, render them, and test the project as an interior decorator might test out a room in a 3D modeling program. The program would be written to read in this text configuration file at run-time, so no compile or debugging would be necessary. At the initial stage of the *360* project, this idea was scrapped due to time and monetary constraints and also, because the OpenGL code is quite simple and similar to the above syntax; it was thought that the artists could just as easily learn the coding syntax. What we did not consider, however, was that the artists would be required to learn a new editor, learn to compile and debug their code, and learn how to run it on a complicated distributed-graphics, grid operating system. In the future, the configuration file route will be taken.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the groundbreaking work done at the Electronic Visualization Laboratory at the University of Illinois, Chicago, Illinois, USA on room-sized immersive visualization and are grateful to Kathleen Harleman, Director of the Krannert Art Museum, University of Illinois, Urbana-Champaign, USA for her continued faith in the *CANVAS* project. Work on *CANVAS* is a result of development of the Beckman Institute VR-Cube [19], a project funded by the National Science Foundation under grant 0079800 and the Beckman Institute. CAVE™ is a trademark of the Board of Trustees, University of Illinois.

REFERENCES

- [1] EVA 2005 Florence Proceedings pp.160-165
- [2] <http://www.evl.uic.edu/art/index.php3>
- [3] Banvard, Richard A., [The Visible Human Project® Image Data Set From Inception to Completion and Beyond](#), Proceedings CODATA 2002: Frontiers of Scientific and Technical Data, Track I-D-2: Medical and Health Data, Montréal, Canada, October, 2002
- [4] <http://www.virtualrealms.com.au/vrml/tute01/tutorial.htm>
- [5] http://www.siggraph.org/education/materials/siggraph_courses/S98/18/vrml97/vrml97.htm
- [6] <http://vrmlworks.crispen.org/convert.html>
- [7] <http://www.nhm.ac.uk/interactive/vrml/>
- [8] <http://arrhythmia.hofstra.edu/vrml/museumn/museum.html>
- [9] http://www.narrativerooms.com/pogany/vr/index_a.html
- [10] http://www.sciencemuseum.org.uk/wellcome-wing/splash_ns.html
- [11] <http://graphics.stanford.edu/projects/mich/>
- [12] <http://www.vrmlsite.com/oct96/spotlight/qa/qa.html#use>
- [13] <http://www.cnn.com/SPECIALS/multimedia/vrml/hurricane/>
- [14] <http://www.isl.uiuc.edu/Cube%20Projects/VRMLView/vrmlview.htm>
- [15] <http://www.isl.uiuc.edu/szg/doc/SceneGraph.html>
- [16] <http://www.python.org>
- [17] <http://www.perl.org>
- [18] <http://www.ruby-lang.org/>
- [19] http://www.isl.uiuc.edu/Labs/room_b650.htm