Project Number: GFP0804

A Virtual Team Room in Wonderland

A Major Qualifying Project Report

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

Joshua Dick

Gerard Dwan

Date: January 12th, 2009

Approved:

Professor Gary F. Pollice, Major Advisor

Abstract

When a team of students working on a large software engineering project needs to collaborate together, holding in-person meetings can be difficult. Various issues could prevent the team from collaborating easily and effectively in person. Our project aims to solve these problems. We developed two software modules for Project Wonderland, an open-source toolkit from Sun Microsystems Labs for creating collaborative 3D virtual worlds. We used these modules, along with preexisting modules, to build a proof-of-concept virtual team room inside Wonderland for use by students taking WPI's Software Engineering class.

Acknowledgments

We would like to thank everyone who has helped, encouraged, or guided us throughout our project.

George Heineman and Gary Pollice organized the Sun Microsystems Project Center. They have put lots of time and effort into making the project center a reality. Professor Pollice, who was also our project advisor, gave us lots of guidance and help along the way, and our experience would have been much less beneficial without it.

We would also like to thank Sun Microsystems for allowing us to utilize their facilities and employees for our project. The patience, help and knowledge exhibited by Jordan Slott, Nicole Yankelovich, and the rest of the Project Wonderland team are truly appreciated.

Table of Contents

List of Illustrations and Figures

Introductions

WPI’s Computer Science department offers an undergraduate course called Software Engineering. When we took the course, the professor who taught it defined a fairly large software development project. Students split into development teams to complete this project. The professor runs the course as if he were running a small software company, and student teams are in turn subjected to issues that could arise at “real” software companies, better preparing them for real software development jobs. Throughout the duration of the course, students are required to meet with a company president, marketing guru, and technical advisor (all characters are realized by the professor or other course staff).

Because each student team develops software separately, but are all trying to realize the same project, there is a large amount of competition between teams. Which team will be able to bring project to fruition most effectively in seven weeks? Inside a team, students work very closely with each other to brainstorm, assess tasks and plan a development schedule, evaluate what progress they’ve already made, and (of course) write code. Although a lot of work is done independently, it is necessary for students to frequently hold team meetings so that the entire team can collaborate.

Unfortunately, holding team meetings is an idea that is easier said than done. Often, scheduling meetings is a nightmare, attempting to find a block of free time common to each team member’s already busy schedule. Team members live at various locations on and off campus, which makes attending meetings difficult for them. Furthermore, there is no consistent, defined space where teams can meet. As a result, students often end up holding meetings in random computer labs on campus, which are not ideal meeting environments, especially if other students share the space with them.

All of the students that were accepted into the Sun Microsystems MQP project center were presented with various project ideas, all to be executed in different groups at Sun. After learning about each available project, each student ranked them by preference and was assigned to a project based on that information. The project that we were assigned to focused on Project Wonderland from Sun Labs, which is open source toolkit written in Java for creating collaborative 3D virtual worlds.

The question then became, what should we do with Wonderland? Our WPI MQP advisor Gary Pollice is one of the professors who teaches Software Engineering. Professor Pollice was well aware of the aforementioned problems students face when trying to hold team meetings, and he wondered if the collaborative Wonderland software could address those problems. Eventually, his thoughts evolved into the idea of a virtual team room in Wonderland, for use by students taking Software Engineering at WPI. If executed correctly, a virtual team room could overcome the issues associated with holding in-person team meetings.

With this goal in mind, we went to work. We wanted our software to be useful for the original virtual team room idea, as well as for future projects by Sun or by the open source community. After brainstorming with our Sun liaison Jordan Slott and the principal investigator for Sun Labs' Collaborative Environments Project, Nicole Yankelovich, we decided that trying to create an entire virtual team room from scratch was not an ideal way to start out. Rather, we decided to create software components (Wonderland modules) that would be useful for both our team room and for project Wonderland in general. The following components help meet our goal because they can be placed in a virtual room and help teams to collaborate.

The first component we worked on was called the “HTML Viewer”. The HTML Viewer is a lightweight Web browser that allows Web pages to be displayed as posters inside Wonderland. In a virtual team room setting, the HTML Viewer can display project tasks or project build statistics on a wall in the team room, so that this information is immediately visible to all team members when they attend a meeting. The HTML Viewer has obvious potential uses outside the scope of our virtual team room project. We lovingly refer to the HTML Viewer as our “training wheels” component, since working on it got us accustomed to developing for Wonderland.

We wanted our next component to take advantage of Wonderland’s 3D space. After more brainstorming with Professor Pollice, Jordan, and Nicole, we came up with the idea for a 3D diagramming and data visualization tool that we decided to call “WonderBlocks”. The idea for WonderBlocks stemmed from an earlier idea involving visualizing software development tasks and their dependencies in 3D. This is just one possible use for WonderBlocks in a virtual team room setting, but WonderBlocks has countless other potential uses.

After developing the HTML Viewer and WonderBlocks, we created a proof-of-concept virtual team room that housed our components as well as other preexisting Wonderland components.

We organize this paper as follows. The “Background” section discusses the concept of virtual worlds, as well as Wonderland itself, in more detail. The “Methodology” section chronicles our work on the two components and our development process. The “Results and Analysis" section examines the outcome of our project, and our accomplishments. We discuss how our project realized our plans at the beginning, and how those plans evolved over time. Finally, the "Future Work and Conclusions" section discusses what Sun Labs, the open source community, and we might do to build upon our work.

Background

In this section, we discuss virtual worlds in general. We explore the different kinds of virtual worlds that are available, as well as reasons why virtual worlds exist. We also compare and contrast popular virtual worlds and Project Wonderland. Finally, we discuss some aspects of the higher level design of Project Wonderland.

Virtual Worlds

Virtual worlds, on a very basic level, are simulated environments. These environments can be represented in any number of ways, such as text-based, two-dimensional, and three dimensional environments. The users, or in many cases, inhabitants, of these environments must have a way to represent themselves. Each user is represented visually in some way. This visual representation is called an avatar. Usually, users have the ability to change the appearance of their avatars as they see fit.

Most of the virtual worlds available today utilize three-dimensional environments. In some cases these environments have the same rules as the real world. Avatars can interact with elements of the world and simulated gravity tethers avatars to the ground. There are many different locations and environments that can be explored by an avatar, and these locations and environments are, at times, analogous to real-life places. Places such as an outdoor terrain or an office building are norms in virtual worlds. Virtual worlds can also include environments that are not analogous to real-life places, such as the environment of "Music in Wonderland"[1], where users can see and interact with a library of music that is visually displayed in a 3D space.

[1] (Sun Microsystems, Inc., 2008)

There are several ways that users can interact with each other through their avatars. Some virtual worlds allow for text-based communication between users. This is similar to having an instant message chat, but an avatar represents the person that one speaks to. Another popular form of communication is via audio, where one can use a microphone and headphones to talk to other users. Many virtual worlds take advantage of directional audio; that is, if an avatar is speaking to the user's left, the left speaker projects the avatar's audio track, and there will be a slight time delay between the left and right speakers (the sound will be in the left speaker first, then move to the right.) In addition, in some cases the volume of the audio is affected by the distance between the user's avatar and another avatar in the virtual world.

There are many virtual worlds available today. One of the more popular virtual worlds is Second Life®. Second Life is a 3D virtual world that was created and is continuously changed by its users (called Residents.)[2] Second Life is "inhabited" by millions of people worldwide. This means that millions of people log in and create homes and/or markets within Second Life. This can be done easily, thanks to Second Life’s in-world scripting language. This in-world scripting language is said to be very easy to use, and whatever each user creates becomes his or her property. Many companies have started using Second Life to advertise and recruit employees. IBM has eleven different locations in Second Life,[3] and Second Life even has a virtual Apple Store.[4]

[2] (Linden Research, Inc., 2008)
[3] (Second Life Business Communicators Wiki, 2007)
[4] (MacBlogz.com, 2008)

Virtual worlds can also be used as educational tools. Professors can give lectures to students across the globe. Sixty schools have real estate in Second Life.[5] This is made possible by tools that are offered in virtual worlds such as virtual lecture halls and presentation viewers. In a similar vein, virtual worlds can be used for business training. In one instance, a virtual world has even been used for flight training for the air force.[6] IBM has also made use of virtual worlds. They have created a Business Center in Second Life. The business center offers tours that walk people through their data centers. They also have speakers that give lectures to people about the up and coming technology being developed at IBM. Additionally, they hold conferences where employees and customers come to collaborate about new ideas, or find ways to solve current problems.[7]

Sun Microsystems

Sun Microsystems (Sun) has a vision in which "The Network is the Computer." Sun has been creating revolutionary processors for years and has gone above and beyond the industry standard with the UltraSPARC processor. Sun has also gotten its name out with Java, a programming language and platform that can be used on almost any computer, and most mobile phones. Sun produces a UNIX-based operating system called Solaris, and has also fostered development of a free, open-source Office software suite called OpenOffice.

Sun Labs is the research and development branch of Sun Microsystems. They typically work on products that are revolutionary and different. There are sub-groups inside Sun Labs that work on different technology areas. For example, one of the groups is working on Sun SPOTs. Sun SPOTs are Small Programmable Object Technology devices.

[5] (Lamb, 2006)
[6] (Sun Microsystems, Inc., 2007)
[7] (IBM, 2008)

They are used in many different ways, from heart and heat monitors to educational tools. Their purpose is open ended. Another Sun Labs group works on Project Wonderland, which we discuss next.

Project Wonderland

Project Wonderland is a toolkit created at Sun Microsystems Labs for creating 3D virtual worlds. Wonderland is written completely in the Java programming language and is entirely open-source. Project Wonderland is built on top of an open-source distributed game server framework—also from Sun Labs—called Project Darkstar. Wonderland's Software Phone is built on top of jVoice Bridge. As mentioned earlier, all of this is written entirely in Java. On top of all of Project Wonderland and the Software Phone are the virtual worlds that can be created using Wonderland software. The following image shows the software stack discussed here.

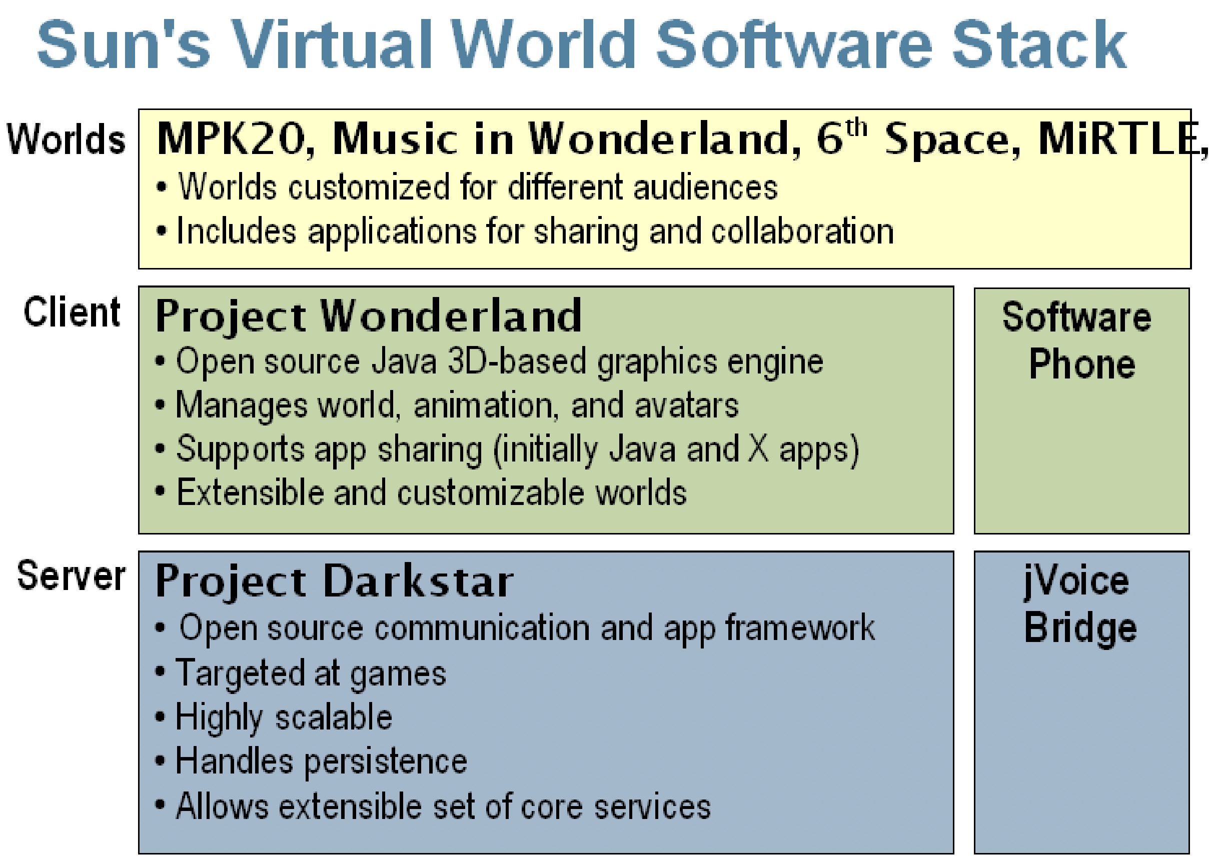

Sun's Virtual World Software Stack
Worlds
MPK20, Music in Wonderland, 6th Space, MiRTLE,
• Worlds customized for different audiences
• Includes applications for sharing and collaboration
Client
Project Wonderland
• Open source Java 3D-based graphics engine
• Manages world, animation, and avatars
• Supports app sharing (initially Java and X apps)
• Extensible and customizable worlds
Software Phone
Server
Project Darkstar
• Open source communication and app framework
• Targeted at games
• Highly scalable
• Handles persistence
• Allows extensible set of core services
jVoice Bridge

Figure 1: The software stack of Project Wonderland.[8]

Another part of the Wonderland architecture is the concept of the cell. A "cell" in Wonderland typically acts as a container for a given Wonderland module. For example, when a developer places a whiteboard into a virtual world, or room, the cell that the whiteboard is placed in contains information such as which direction the whiteboard will face and where the whiteboard should be located. In order to create multiple whiteboards in a given area, there must be multiple cells containing whiteboards. Simple information that a given cell needs to be placed in-world and initialized correctly (such as size and position information) is stored in and read from a Wonderland Cell Descriptor (WLC) XML

[8]https://lg3d-wonderland.dev.java.net/files/documents/5924/119938/Wonderland-InWorldSlides.odp

file. WLC XML files are part of what is called the Wonderland File System (WFS), which includes all files needed to represent a given virtual world.

Project Wonderland also includes superb audio technology. The Wonderland team believes that that audio is important for effective interaction inside a virtual world, and so they have created a unique audio experience that immerses users into Wonderland. For the next iteration (version 0.5 as of this writing), Wonderland developers are working to completely revamp the Wonderland architecture, specifically in the areas of Wonderland modules and 3D graphics.

One of Project Wonderland's primary intended uses is as a business tool that brings employees together from around the globe to collaborate. Wonderland was initially released as a technology demo, and development could have stopped at that point. However, Sun is pursuing further development on Wonderland in order to explore the new and relatively untapped field of the use of virtual worlds in business environments.

Wonderland is open-source; it is very accessible. Because it was developed with the open-source community in mind, it is also extremely extendable. Developers can add functionality to Wonderland, as well as create new models for buildings (from floor to ceiling), and the graphics for new tools. Developers can create components that can be added to the world with ease. These components are known as Wonderland modules. There is a large array of modules already available in Wonderland and there are many more in the works. A few of the modules that can be used today are shared whiteboards, tools to view PDF-formatted documents, and music exploration applications.

How does Project Wonderland compare to other virtual world products? Project Wonderland is not a virtual world in itself; rather, it is a toolkit for creating virtual worlds.

Most virtual worlds are made for leisure or entertainment, whereas Project Wonderland was built more to accommodate meetings between business associates that could not make it to a meeting whether they are sick, injured, or live in a different time zone. While most virtual world software available today is not open-source, Project Wonderland has a strong open-source community contributing to the project. Community contributions become part of Wonderland fairly quickly, allowing the project to grow in ways that the original developers never thought. Anyone can run a Wonderland server and create their own virtual world. This is different then many other virtual world software, which would typically require logging in to a service and using servers run by the company providing the software.

Wonderland is now undergoing a major architectural upgrade. The Sun Labs team is improving the API for developing in Wonderland. Additionally, Wonderland's graphics will be handled by JMonkey Engine[9] instead of using the Java3D library, which was originally used to render Computer Aided Design drawings. JMonkey is an open source API for high performance graphics. JMonkey allows for better shading and three-dimensional rendering than Java3D, which will translate to better looking avatars, objects and environments for Wonderland-based virtual worlds. Wonderland's new avatars will be able to show expressions, which is not possible in the current version of Wonderland. Wonderland's module system will be revamped, making it easier than ever to develop and customize virtual worlds created with Wonderland.

[9] (JME, 2008)

The Virtual Team Room

Software engineering students at WPI can run into problems when conducting necessary formal meetings to collaborate on their team projects. These students cannot always meet at the times scheduled because of other meetings, athletics, or other commitments. Once a meeting time is determined, a room needs to be allocated in which the team will meet and the meeting rooms on campus are oftentimes occupied with clubs, groups, and organizations that reserve the meeting space year round, so booking a meeting room could be out of the question. Classrooms are not typically available for students to have meetings and even so, classrooms are not ideal for holding meetings. In order to accommodate these problems, we decided to create a virtual team room using the Wonderland toolkit.

In our vision, the virtual team room is a place that Software Engineering students can meet any time of the day or night. Students can be in this team room at any time of day they wish, and from anywhere there is a computer or telephone. There is no need to worry about another club or organization using the team room, as it is made specifically for the use of these engineering students.

There are additional benefits of using the virtual team room as opposed to scheduling a time in a regular meeting room. There are ways to keep track of progress in the virtual team room, so that the students in the team will be able to visualize how far along they are and what work needs to be completed before the project is considered complete. Though these resources may also be available in a real-world team room, they are available around the clock in a virtual team room. There is no need to try to find all of the needed materials for the project if one is in the virtual team room, as it is also a place to organize project references and requirements.

Course staff can also take advantage of the virtual team room. They will be able to keep track of metrics involving student participation and project progress. In doing so, they will find if the room is being used to its full potential and give suggestions on how to use the room more efficiently. Course staff can also hold office hours in the virtual team room. This way, even if a student is not able to physically make it to a teaching assistant (TA) for any reason, he or she can still meet with the TA. The professor will be able to go into the team room as well. He or she will be able to assess the progress being made by the teams, and (as a teaching assistant may do) can make suggestions for improvement.

While creating a team room has its benefits for the Software Engineering students at WPI, it does not have as much to offer for the Sun and Open Source communities. After considerable brainstorming, we decided to create modules that could be useful for a virtual team room, rather than creating the team room itself as the main deliverable. This way, our work can be useful for the Wonderland community in the future, instead of being tied to one specific use. We believe that this approach aligns well with Project Wonderland's general goals as well as the goals of the developers at Sun.

A side benefit of this decision is that the virtual team room can make use of preexisting Wonderland modules. Wonderland's PDF Viewer can be used to display slideshows or assignment sheets in-world. Wonderland also has a whiteboard module that can be used to collaborate or make quick diagrams and notes. Wonderland even includes an in-world phone system, so that if a team member cannot attend a virtual meeting on the computer, they can still call in to the meeting using an ordinary telephone and participate that way. Additionally, work is being done on an authentication process for Wonderland. This could allow for individual teams to keep their work private, and prevent it from being tampered with.

With these ideas in mind, we decided to create modules for a virtual team room. Our thought process on the specific components and how we created the components are discussed in the next section of this paper.

Methodology

We spent our first day or two at Sun wondering where to start. We needed to think of a "training wheels project" that would get us comfortable with developing Wonderland modules. We used a three step approach to accomplish this. First, we worked through several Wonderland development tutorials written by Jordan Slott[10] to get a general feel for the structure of (and process of developing) Wonderland modules. Next, we explored the source code of already-existing Wonderland modules to learn about how complex, useful modules are constructed. We then had enough background knowledge to start creating our first Wonderland module.

The HTML Viewer

Conception

We decided that the best way to create our first Wonderland module was to modify an already-existing Wonderland module to fit our needs, rather than start from scratch. We were free (and even encouraged) to modify an existing module. We chose the PDF Viewer module created by Sun developer Nigel Simpson.

There is a different Wonderland module that allows any X11 application to be displayed and manipulated in-world. (X11 or X Window System is "is a highly configurable, cross-platform, complete and free client-server system for managing graphical user interfaces (GUIs) on single computers and on networks of computers."[11] The majority of Linux applications available today use X11.) One of the primary uses of this module is to facilitate in-world Web browsing by using X11 forwarding with a full-featured Web

[10] (Slott, 2008)
[11] (Linux Information Project, 2006)

browser like Mozilla Firefox. This solution works well when a full-featured interactive Web browser is available, but simply displaying Web pages in-world could be done much more easily. The X11 forwarding module can be difficult to set up, requiring a host machine to actually run the application(s) to be forwarded.

These observations gave us the idea to take Nigel's PDF Viewer and modify it to display Web pages in-world. Our module would not be an interactive Web browser. Rather, it would display a Web page as if it were an in-world poster. It would do so in a way that is more lightweight than the X11 solution, because it would be written completely in Java, requiring no external software or Web browser dependencies. Web pages are written in HTML, and since our module would be based on the PDF Viewer, it logically made sense to call our module the HTML Viewer.

We thought of several ways in which the HTML Viewer would be useful for the virtual team room. One could display a Web page containing current project build statistics, or team tasks inside the SourceForge system used by WPI's software engineering classes, or class announcements. Several HTML Viewers could display Web pages that may require a quick glance during a typical team meeting. For instance, if the HTML Viewer displayed a single page from SourceForge, that page would look different every time a meeting was conducted, always showing the most current information. In other words, the HTML Viewer was designed to display information that needs to be quickly reviewed, without requiring user intervention. Without interactivity, the HTML Viewer has an easy "set it and forget it" mechanism; for instance, the SourceForge build statistics page could be posted to the team room wall once, and then it would simply stay there, visible for any team member to see. In addition to being useful for the virtual team room, the HTML Viewer would also be useful for other Wonderland worlds in the future, when a full-featured Web browser was not needed but a lightweight way of displaying a Web page would suffice.

First Steps

Since we knew our module would have to display Web pages without utilizing a traditional Web browser, the very first thing we did was to research methods for rendering Web pages using Java alone. Since the point of creating the HTML Viewer was not to write an HTML parsing and rendering system from scratch, but rather to learn how to develop for Wonderland, we decided to look for preexisting open-source Java HTML rendering engines that we could simply integrate into the HTML Viewer. Most of the available options were either extremely out-of-date or rendered Web pages too poorly to be useful. At the end of our stint of research in this area, the best option seemed to be the open-source Cobra HTML Renderer and Parser[12]. Cobra is part of the Lobo Project[13], which is an open-source Web browser written completely in Java. During our testing, Cobra rendered Web pages more accurately than any of the other open-source Java HTML rendering engines we tried.

The decision to use Cobra inside the HTML Viewer introduced a problem involving licensing. The Cobra library is released under an LGPL license. Cobra utilizes a library from Mozilla called Rhino for its JavaScript functionality, which is released under a dual MPL 1.1/GPL 2.0 license. Wonderland is released under a GPL 2.0 license, but can be dual-licensed for commercial customers[14]. Shortly after we made the decision to use Cobra, Jordan informed us that a potential dual-licensing conflict could occur between Rhino (and

[12] http://lobobrowser.org/cobra.jsp
[13] http://lobobrowser.org
[14] (Sun Microsytems, Inc., 2008)

therefore Cobra) and Wonderland. Only Rhino's GPL license was compatible with Cobra's LGPL license; selecting the MPL license for Rhino would have violated Cobra's LGPL license terms when using it with Cobra. However, Rhino's GPL license has a "viral" effect: all code that "derives" from Rhino must also be licensed under GPL. The definition of "derives from" is unclear and debated in the legal community. The potential conflict stemmed from the fact that using Cobra/Rhino might force Sun to license all of Wonderland under GPL, which would prevent commercial (non-GPL) licenses form being issued to customers.

Jordan said that Cobra would have to undergo a license review by Sun's legal department before it would be allowed to officially be included in Wonderland. As a temporary solution, we simply did not store Cobra in the area of Sun's code repository that houses the HTML Viewer code. To use the HTML Viewer after checking out the code, one currently has to manually download the Cobra library and place its component files in specific areas, although our documentation for the HTML Viewer includes instructions that explain how to do this. Although we later found out that Sun Legal approved the use of Cobra/Rhino, we chose to not make any further repository changes. This way, an administrator that sets up an instance of the HTML Viewer will always be forced to seek out and use the newest version of the Cobra library.

We next had to tie the HTML viewing capability into Nigel Simpson's PDF Viewer that we had selected as our starting point. Nigel's PDF Viewer works by converting PDF pages into Java BufferedImage objects that are eventually painted in-world. Thus, we knew that we'd need a way to render HTML and store the result in a BufferedImage object, rather than display it directly on-screen (like Cobra's sample code had showed us how to accomplish). While researching how to accomplish this, we found a blog post[15] containing Java source code that accomplished that very task. The post's author did not mention anything regarding the licensing of the posted code. However, the code had been made publicly available so we assumed that it could be legitimately adapted for our use, given proper citation.

Armed with this knowledge, we started actually modifying the PDF Viewer. We first wrote a Java method that accepts a URL to a Web site in String form, and returns a BufferedImage of that Web site. We found the point in the PDF Viewer source code where a variable containing a BufferedImage of a PDF page gets painted to the screen. Directly before that point, we changed the value of the variable holding the BufferedImage by calling our new method. At the end of our fourth day at Sun, we were very excited to see Web pages render in-world. (See the figure below.)

[15] (Brown, 2008)

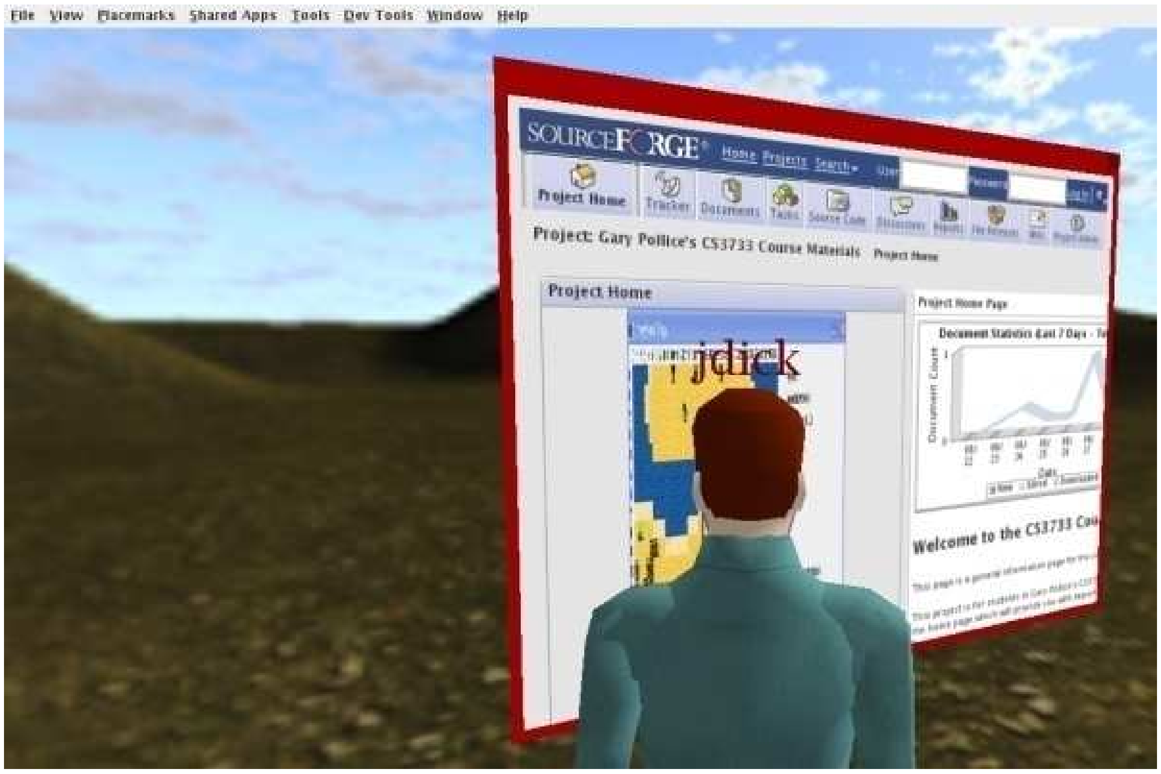

File View Placemarks Shared Apps Tools Dev Tools Window Help
SOURCEFORGE
Project Home
Project: Gary Pollice's CS3733 Course Materials
Project Home
jdick
Welcome to the CS3733 Cou

Figure 2: Web pages rendering in-world at the end of day four

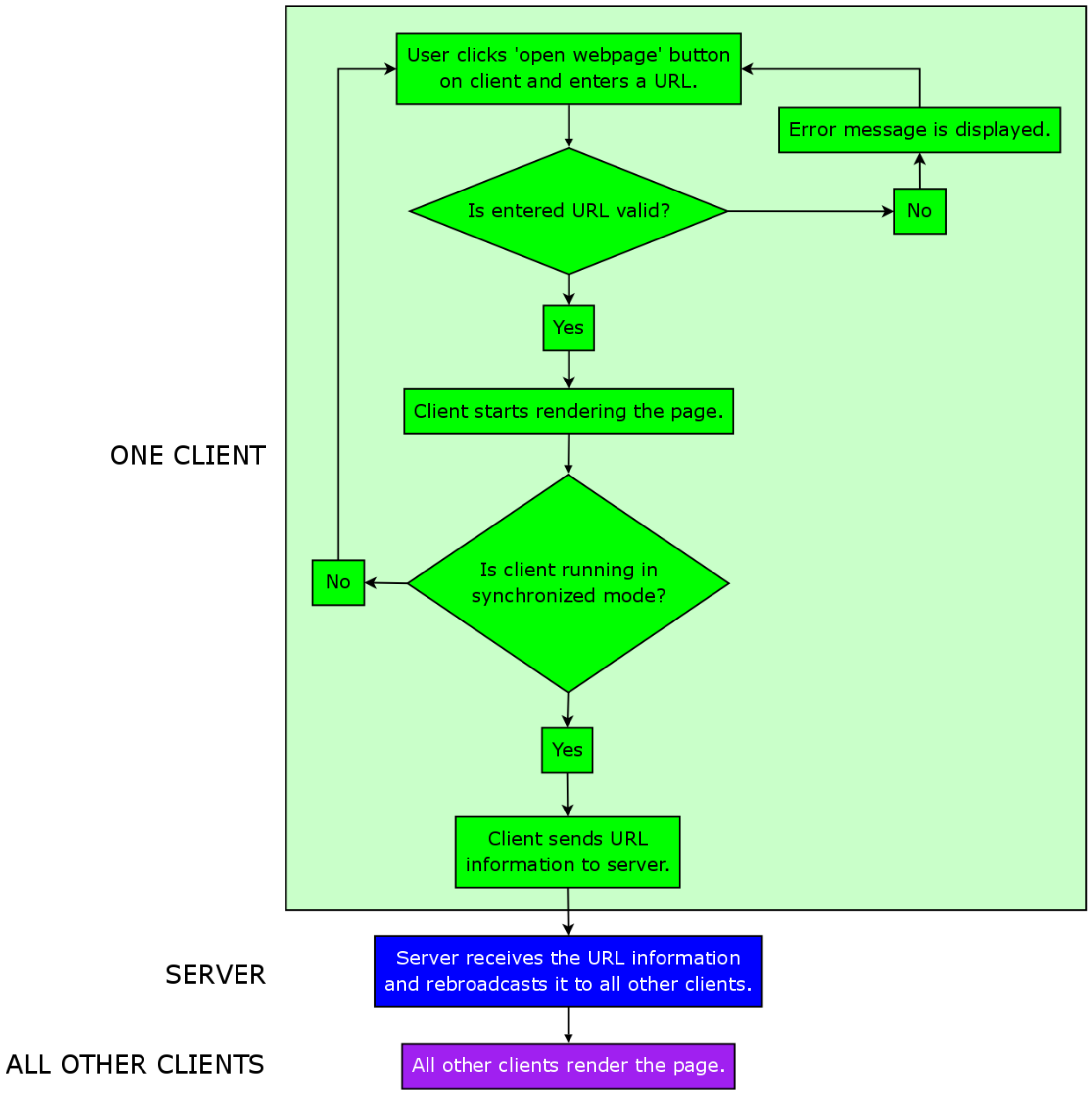
Next Steps

Despite our excitement, there was still a lot of work to be done on the HTML Viewer. At this point, we started referring to the module as the "hacked PDF Viewer", because it still did all of its PDF processing and rendering computations, then ignored the result and substituted a Web page image in its place. A large portion of the subsequent development work on the HTML Viewer involved removing unneeded or redundant code associated with the PDF viewer. Additionally, we had to make design decisions regarding how the finished HTML Viewer should be structured.

The most important decision we made was that the HTML Viewer would be thick-client/thin-server. That is, the Web page rendering is done on each Wonderland client, rather than on the server. This reduces the load on the server in two ways: first, the server would not have to physically render the page (which is computationally intensive,) and second, it does not have to send the rendered image to each client over the network. Instead, the server is used to send messages between clients (for example, "Render the page 'http://www.sun.com'"), and clients act on those messages independently. The next design decisions we made involved the messaging protocol that the server would use to communicate to clients. We ended up modifying the existing message protocol in the PDF Viewer.

Now that the unused PDF Viewer code and GUI elements were removed from the HTML Viewer, we set to work modifying parts we had left intact to better suit our needs. We added relevant error messages/dialogs. We added the ability to zoom in and out on a page using either the mouse wheel or HUD (heads-up display) buttons. We also added the ability to synchronize or un-synchronize the HTML Viewer. When an in-world instance of the HTML Viewer is "synced", all clients that are also synced will affect each other; if one synced client navigates to a new Web page, all other synced clients then display that page. If an HTML Viewer is "un-synced" in a particular client, no other client is affected by it. When an un-synced instance of the HTML Viewer re-syncs, it displays the same page as its synced peers.

When a client is instructed by the user to display a new page, it first checks if the page is valid. If the page is invalid, an error message is displayed. If the page is valid, the client then starts rendering the page. If the client is running in synchronized mode, it uses the aforementioned messaging protocol to notify the server that a synchronized page change should occur. The server then sends a message to all other clients, instructing them to display the new page. This process is outlined in the control flow of the HTML Viewer as depicted in the figure below.



ONE CLIENT
User clicks 'open webpage' button on client and enters a URL.
Is entered URL valid?
No
Error message is displayed.
Yes
Client starts rendering the page.
Is client running in synchronized mode?
No
Yes
Client sends URL information to server.
SERVER
Server receives the URL information and rebroadcasts it to all other clients.
ALL OTHER CLIENTS
All other clients render the page.

Figure 3: Control flow of the HTML Viewer

Final Steps

The final feature we added to the HTML Viewer was a refresh button, allowing clients to individually refresh the currently displayed page. Aside from final general code polishing, we also fixed a major problem where the Wonderland client would become temporarily unresponsive while the HTML Viewer rendered pages in un-synced mode (this was not happening in synced mode). It turned out that the problem involved knowing which thread was calling a time-consuming method that actually rendered Web pages. We discovered that when the HTML Viewer runs in un-synced mode, the Swing Event Dispatcher thread is responsible for this. This is because when the HTML Viewer runs in un-synced mode, it does not listen for any instructions from the Wonderland server. Therefore, threads specific to Wonderland's Darkstar backend do not run while the HTML viewer is in un-synced mode and cannot be used for this purpose. However, when the HTML Viewer runs in synced mode, threads provided by Darkstar can and should be used, because they abstract away the Swing Event Dispatcher thread. The solution was to allow the appropriate thread to call the rendering method, depending on which mode (synced or un-synced) was currently selected.

At the end of our second week at Sun, we decided that we had done enough work on the HTML Viewer, and that we had fulfilled the goals we had hoped to accomplish by developing it. We knew that more features could always be added to the HTML Viewer (see the 'Future Work and Conclusions' section of this paper,) but that it was time to remove our "Wonderland development training wheels" and move on.

WonderBlocks

Conception

Well before we had finished working on the HTML Viewer, we started brainstorming with Professor Pollice, Jordan, and Nicole to come up with ideas for the next Wonderland module that we would develop for use in the virtual team room. Since the

HTML Viewer is essentially a 2D application used inside a 3D space, we wanted our next module to actually utilize the 3D space. After thinking of and rejecting ideas for our new module, one idea seemed promising: a module that visualizes project tasks and their dependencies in 3D. This module would be extremely useful in a virtual team room setting. During a meeting, tasks could be assigned to or traded between team members. If one or more tasks had to be completed before higher-level tasks could be started, this module could clearly show that.

In our vision for this module, tasks are represented by 3D shapes: blocks, spheres, cones, and cylinders. Shapes have several attributes associated with them, such as size and color. A task's shape could represent the area of the project that the task is associated with; for instance, boxes could represent database-related tasks while spheres could represent UI-related tasks. A task's color could represent a task's completion status. Perhaps red tasks are those that have not been started yet or assigned to any team members, yellow tasks are currently being worked on, and green tasks are those that have been completed. A task's size might represent the relative amount of time/work required to complete the task; simple tasks would be represented by smaller shapes while complex tasks would be represented by larger shapes. Perhaps tasks connected together with lines could indicate that those tasks share a dependency.

Once we had the concept of using 3D shapes to represent project tasks, we started thinking of (and sketching) ways that these 'task shapes' could be manipulated and viewed coherently by multiple users in-world.

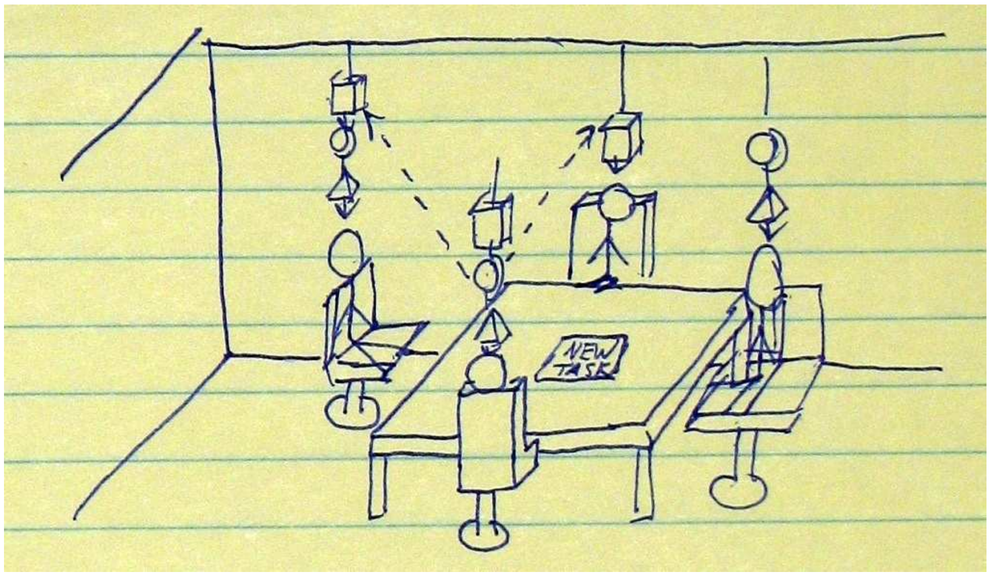


Figure 4: A possible use of "task shapes" in an in-world meeting

The figure above depicts one idea we had in which team members could assign and trade tasks during a meeting. In this vision, tasks assigned to a given team member float above that member's head in a straight line. Tasks are ordered by priority; the most urgent tasks appear closest to each individual's head. If one task depends on another task, this is conveyed by an arrow that appears in midair between those tasks. Any team member can create new tasks, delegate one of their tasks to another team member, change any task's priority, update a task, and remove or complete a task. We believe a tool like this would be a useful visual aid for teams while doing project planning; instead of trying to write everything on a whiteboard or computer, all tasks (and their owners, properties, and relationships) are immediately visible to all team members.

What if a team member enters the virtual team room while a formal meeting is not taking place? Tasks that are assigned during meetings could 'float up' to the ceiling in a "cloud" or "sky" view, as depicted in the figures below.

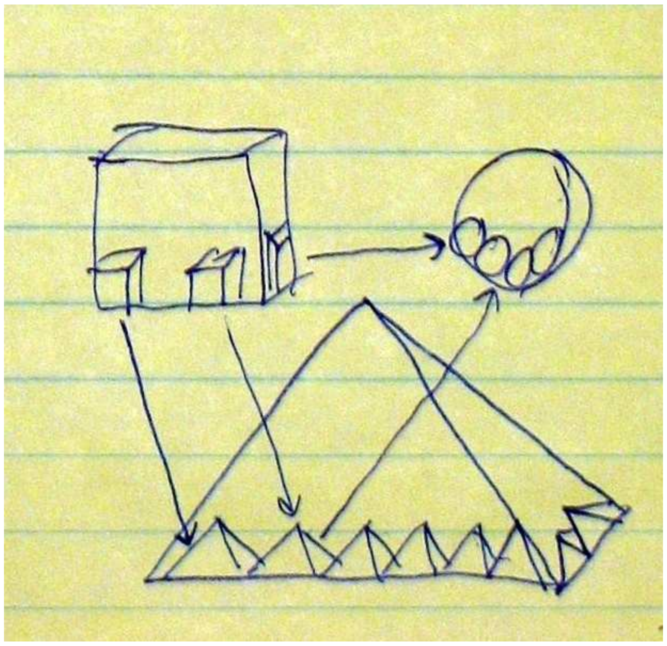


Figure 5: A possible way of grouping "task shapes" and their dependencies

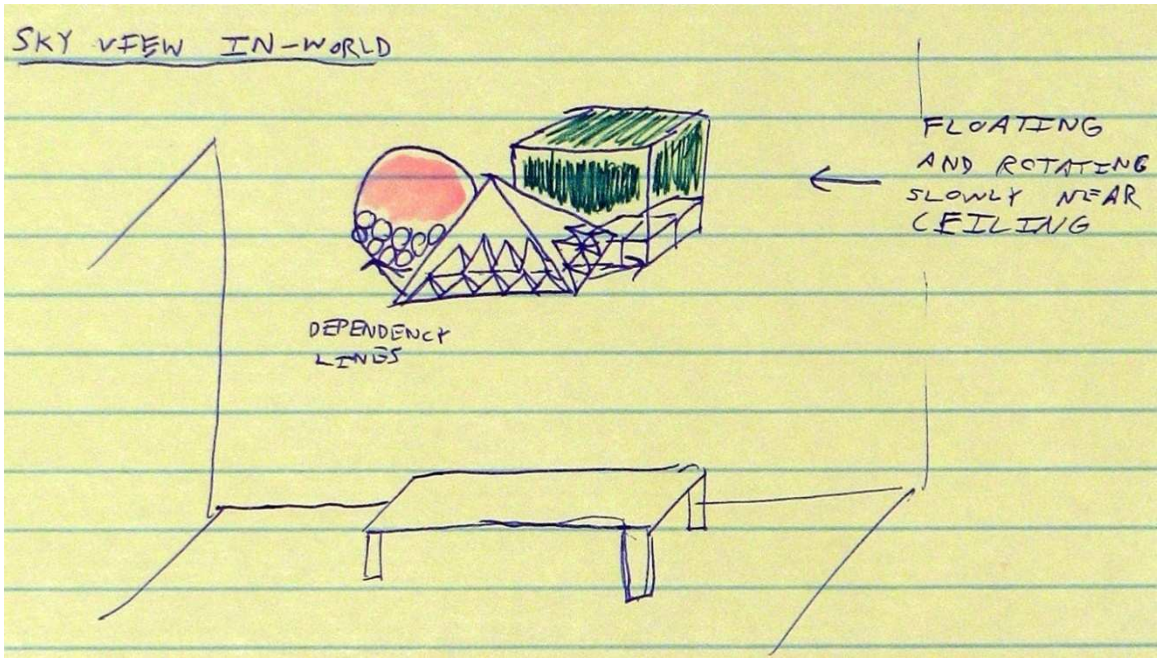


Figure 6: "Task shape sky view", passively displaying all tasks

Regardless of whether a formal team meeting is being held, “sky view” allows all team members to view all tasks (and their dependencies) simultaneously. “Sky view” groups tasks by their shape (which once again represented task category in our vision.) In either of the available views, clicking on a task would show detailed information or notes specific to that task.

After some further discussions with Professor Pollice, Jordan, and Nicole, we realized that our vision of these “task shapes” was slightly narrow. If it were to be completed, a “task shapes” module would be immediately useful for our virtual team room, but perhaps not as useful to developers working on future Wonderland worlds or modules. It was this realization that inspired the idea for our WonderBlocks module.

Instead of doing task planning in 3D, WonderBlocks is much more generalized. It is essentially a 3D diagramming and data visualization tool for Wonderland. WonderBlocks borrows several ideas from our initial “task shapes” vision: objects (tasks, songs, Facebook groups, anything!) are represented by 3D shapes that we decided to call “blocks.” Blocks can be cubic, conic, spherical or cylindrical and can be a number of different colors and sizes. Blocks can optionally have associated text labels that are displayed above them in-world. Blocks can optionally be associated with any number of other blocks, and those associations can be directionless, directional, or bidirectional. Furthermore, multiple users can create, change, and remove blocks and their associations simultaneously.

Using WonderBlocks, groups of users/avatars can collaboratively construct diagrams representing various kinds of information. In a virtual team room setting, WonderBlocks could be used to create diagrams illustrating our original idea of task management, software architecture diagrams, diagrams of database schemas, or any other kind of diagram. It should be clear that WonderBlocks has many potential uses both in a virtual team room setting and in any setting that would benefit from having 3D diagramming functionality.

Now that we had come up with a clear vision for WonderBlocks, it was time to start developing the module.

First Steps

We made our initial design and explored several design considerations. We determined that having the ability to toggle between an "avatar-associated" view and the previously-discussed "sky" view, as in our "task blocks" concept, would be far too complicated to implement in the amount of time we had to complete the project. Instead, we decided that WonderBlocks would operate in a single view that essentially mimics the "sky view" concept, but at avatar level (rather than ceiling level.) In our original "task blocks" concept, we had the idea of associating any type of metadata with a block, and that clicking on a block would display or otherwise convey that metadata. To keep things simple for WonderBlocks, the only metadata associated with a block is a plain text label that would is displayed on that block in-world.

Another design decision we made was that an instance of the WonderBlocks module would occupy a single Wonderland cell when placed in-world. We decided that a Wonderland cell would house all WonderBlocks (as opposed to having one block per cell), and that all interactions with WonderBlocks would occur inside the cell. Information about WonderBlocks cell positioning and size would be stored in a WLC XML file. (Wonderland cells and WLC XML files are described in the "Background" section of this paper.)

Wonderland's persistent data storage capabilities are limited. The Wonderland server does not provide a built-in way of saving information between restarts, so

WonderBlocks diagrams/sessions would be lost and reset to empty diagrams if the Wonderland server was ever restarted. To remedy this problem, we would have to implement our own persistent data storage mechanism that would operate independently of any of Wonderland's built-in functionality. This mechanism would have to persistently store and load the state of an instance of the WonderBlocks module (including information about all blocks and their connections) between Wonderland server restarts. We decided that the module's running state could be saved by using Java's XMLEncoder and XMLDecoder classes to serialize the Java objects representing the state into a single XML file, stored on the computer running the Wonderland server. When the server [re]starts, the WonderBlocks application state is then deserialized back into memory from the XML file. This WonderBlocks state saving and loading process is depicted in the diagram below.

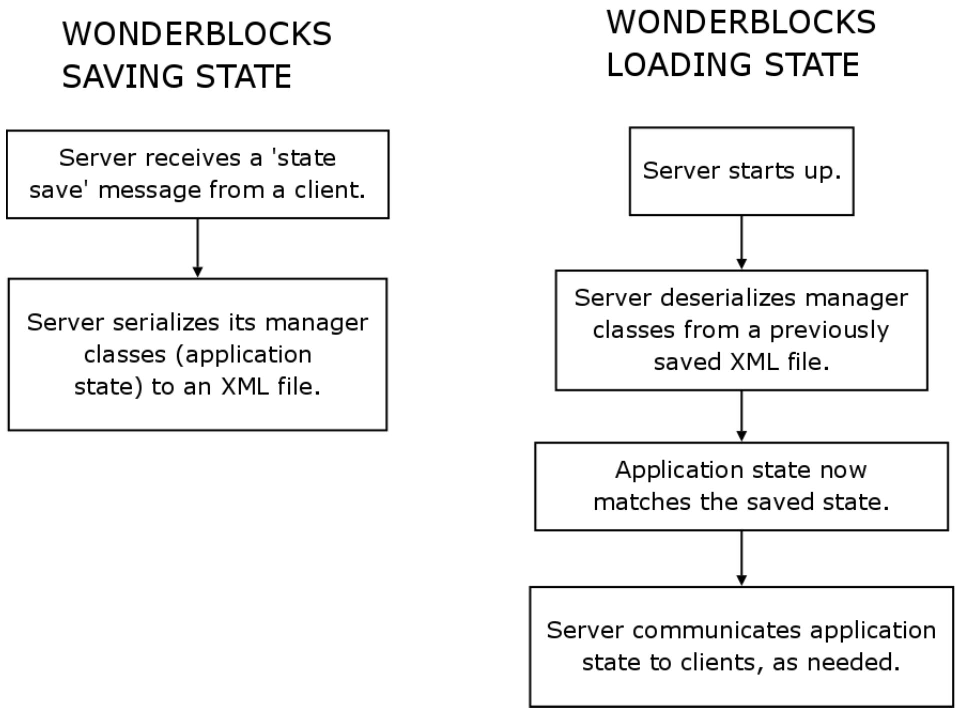

WONDERBLOCKS SAVING STATE
Server receives a 'state save' message from a client.
Server serializes its manager classes (application state) to an XML file.
WONDERBLOCKS LOADING STATE
Server starts up.
Server deserializes manager classes from a previously saved XML file.
Application state now matches the saved state.
Server communicates application state to clients, as needed.

Figure 7: The WonderBlocks state saving and loading process

The XML file mentioned does not use mechanisms from (and is not specific to) Wonderland like the WLC XML file is; rather, it is proprietary and specific to the WonderBlocks module only.

Since we knew that developing WonderBlocks would be a much larger undertaking than developing the HTML Viewer, we split initial development work into two general areas. Code would have to be written concerning the logic and rules governing the behavior of (and relationships between) individual WonderBlocks. We would eventually refer to this code the "WonderBlocks backend". Neither of us had any prior experience doing 3D graphics programming, let alone doing graphics programming with Java3D for Wonderland, as we had only done 2D work for the HTML Viewer. Knowledge of 3D graphics programming (specifically with Java3D) is an obvious requirement for developing WonderBlocks. We would eventually refer to objects that are displayed on-screen and GUI elements as the "WonderBlocks frontend".

The WonderBlocks Backend

As mentioned earlier, we refer to the area of the WonderBlocks code concerning the logic and rules governing the behavior of (and relationships between) individual WonderBlocks as the "WonderBlocks Backend." We made the decision to start developing the backend completely outside Wonderland for simplicity.

The very first part of the backend that was developed was a simple JavaBeans class representing a single block, including information such as block size, shape type, color, and more. A second JavaBeans class representing a connection between two blocks was then created. JavaBeans classes are "reusable, platform-independent components…that can be changed or customized."[16] Utilizing the JavaBeans architecture for blocks and connections ensures that additional features/functionality can be easily added to blocks and connections without needing to make significant changes to preexisting WonderBlocks code.

Initially, the connection class included fields that referred to block objects, and we realized this would cause issues relating to the way we had chosen to implement persistent application state storage (namely, that block and connection objects are serialized to an XML file.) If connection objects directly 'stored' block objects, the blocks inside the connections would be serialized to XML *in addition to* the original (actual) block objects. In other words, using this scheme, redundant copies of block objects would be stored.

To work around this problem, we created two singleton manager classes, one for blocks and one for connections. The manager classes map blocks and connections to unique ID numbers. This way, connections can store blocks' ID numbers, rather than the blocks themselves. Connections reference a given block by telling the block manager to expose a block with a given ID number. All block objects are only accessible via the block manager class, while all connection objects are only accessible via the connection manager class. This separation ensures that no redundant data is stored when each class is serialized to an XML file.

Because the backend code was initially developed outside Wonderland, we were able to write JUnit tests for it. The tests actually helped us find and fix a minor bug. After the backend code had evolved enough, we thought that it was ready to be tied with the WonderBlocks frontend that was being developed at the same time, and to work inside

[16] (Sun Microsystems, Inc., 2008)

Wonderland. At this point, the decision to develop the backend code completely outside Wonderland caused unforeseen problems that will be discussed later.

Note that design notes detailing the workings of the WonderBlocks backend appear in this paper's appendix.

The WonderBlocks Frontend

The WonderBlocks frontend consists of everything users see and manipulate when interacting with WonderBlocks: the blocks and connections as displayed in-world, the heads-up display buttons, and various popup dialog boxes and status notifications. The frontend is essentially the user interface for WonderBlocks. To start developing the WonderBlocks frontend, we experimented with the very simple Wonderland module that had been created as a result of following Jordan Slott's previously-mentioned Wonderland development tutorials. This module placed a textured 3D primitive in-world that would change between a cube and a sphere when clicked. Although this module was certainly very simple, it lent itself naturally to be a starting point for WonderBlocks, as WonderBlocks also involves working with 3D primitives in-world. On our fifteenth day at Sun, we started experimenting with manually adding untextured, solid-colored primitives in-world, as pictured below.

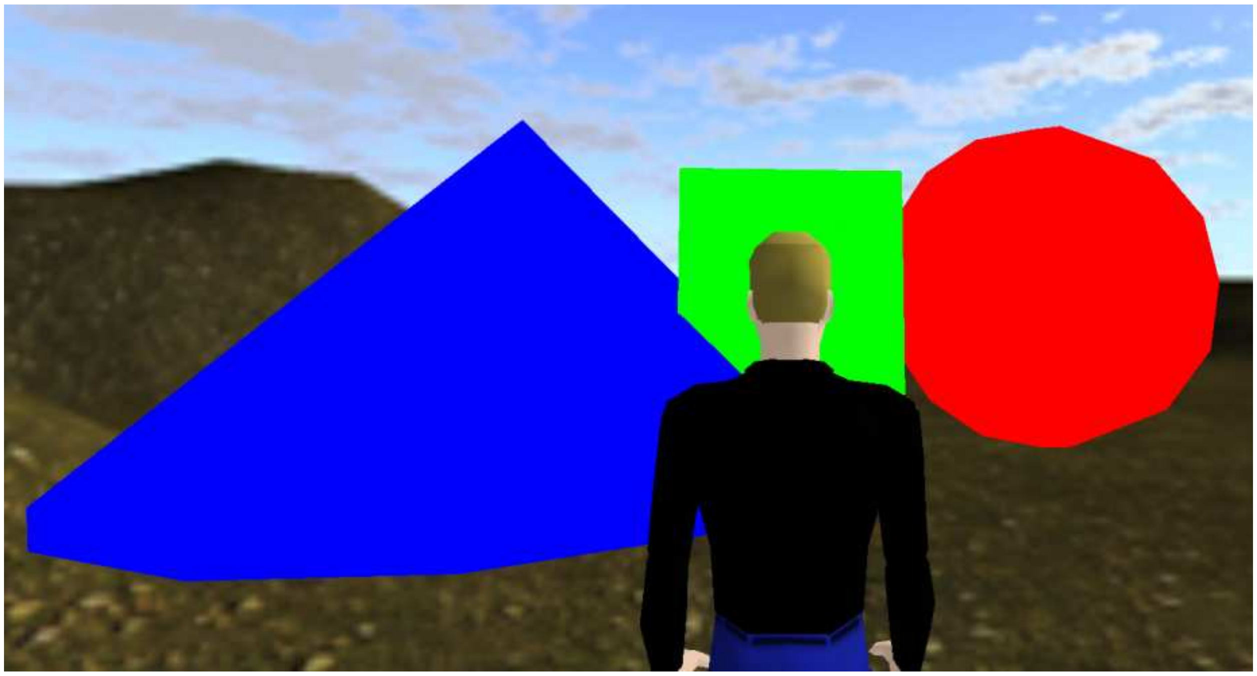


Figure 8: An early experiment with placing colored 3D primitives in-world

We faced some interesting problems during these initial experiments. First, the primitives shown in Figure 7 certainly do not look very smooth or three-dimensional. We did some experimenting with polygon counts, which effectively control how many 2D triangles are used to draw a 3D object, and therefore control how 'smooth' a primitive appears. For each type of primitive, we chose polygon counts that struck a good balance between graphics performance and primitive smoothness.

Because light reflects off of 3D objects, and our primitives looked very flat and not three dimensional, we spent a good deal of time investigating 3D shading techniques in Java3D. After doing a lot of Internet research and talking to various Wonderland team members at Sun, we were able to add appropriate shading and specularity to our primitives, making them 'pop' and appear much more three-dimensional.

Another challenge we faced was actually breaking out of Wonderland's built-in abstractions for placing objects in-world. Normally, Wonderland automatically stores and reads an object's positions and orientation to/from a WLC XML file, and automatically places that object in-world accordingly. Because we did not want each and every block to have their own individual XML files to store positioning and placement information, we had to peel back Wonderland's layers of Java3D abstractions, and to manually position primitives with pure Java3D code.

The largest challenge we faced in developing the WonderBlocks frontend concerned how connections between blocks are drawn in-world. Originally, we had thought to use long, thin, bar-like Java3D cylinders to represent connections. These cylinders were to be placed in-world using this algorithm:

1. The origins of the blocks to be connected are averaged together to find their 3D midpoint, which becomes the connection's origin.
2. The distance between the origins of the blocks to be connected is calculated using the Pythagorean theorem, and this distance becomes the connection's length.
3. A thin cylinder with the calculated length is drawn at the calculated origin. Angles of rotation for the cylinder in all three 3D axes are calculated using basic trigonometric functions, and the cylinder is then rotated into the appropriate position to appear as though it is connecting the two blocks.

It turned out that this algorithm worked much better in theory than in practice. We experimented with this algorithm, and never got the third part of it to work correctly. We drew complex diagrams and experimented with trigonometry, but no matter what we did, the cylinders would simply not rotate into place correctly. After the experimentation, Jordan helped us realize that the problem occurred because our algorithm was trying to do rotations compositionally (rotating in one axis, then another, and finally a third). The problem with this approach is that rotations around one axis may affect the other two axes, so the compositional rotation approach we were using could never work correctly 100% of the time. We discovered a mathematical device to solve this problem called Euler angles[17,18]. However, the math involved is extremely complex and we had already spent two days working on connection rotation with no results, so we decided to move on. Rather than using shaded 3D cylinders to represent connections between blocks, we decided instead to draw connections using a native Java3D function that simply draws an unshaded line that connects two 3D points (in our case, the origins of each block.)

Connections between blocks are directional, and we had to think of a way to represent those directions visually. Our original plan was to place tiny 3D cones at the ends of a directional connection that are parallel to that connection, so that the cones would look like arrows. We knew this would not be an option because of the compositional rotation problem; we would not be able to make the arrows point away from a block or towards a block. Instead, we decided to represent directional indicators with tiny spheres rather than tiny cones. Because spheres are symmetrical in all directions, rotation of spheres is not an issue.

At this point, we had done several experiments with hard-coding blocks and connections, and after we were confident in the techniques involved, we started writing methods to draw blocks and connections dynamically. These methods accept objects from the WonderBlocks backend (blocks and connections) as parameters, and draw them in-

[17] (Weisstein, 2008)
[18] (Wikipedia, 2008)

world. At this point, it became necessary to start formally linking the WonderBlocks frontend and backend code together.

Next Steps

The WonderBlocks backend and frontend code were now evolved enough that they were ready to be linked together into the first version of the WonderBlocks module. To do this, we had to design a messaging protocol allowing the Wonderland server to communicate with all clients running WonderBlocks. After our initial work on the messaging protocol, the Wonderland server was able to deserialize some information about blocks and connections from an XML file and then send that information to each client running WonderBlocks. Each client would interpret the received information and draw blocks and connections accordingly.

Along the way, we gradually added more features to the WonderBlocks frontend. We were able to figure out how to get block and connection labels to rotate in 3D and always face a given user's avatar (each connected user always sees the labels rotate to face them as they move around in-world.) This feature is useful because this way, labels are always readable for all clients, no matter where avatars are positioned; avatars cannot walk 'behind' labels causing them to appear backwards, because they automatically rotate to face the avatar.

WonderBlocks was finally starting to look as we had originally envisioned it, sporting shaded blocks, labels, and connections with directional indicators (see the figure below.) However, there was still a lot of work to be done on WonderBlocks. The module still needed a user interface, and its client/server messaging protocol left a lot to be desired.

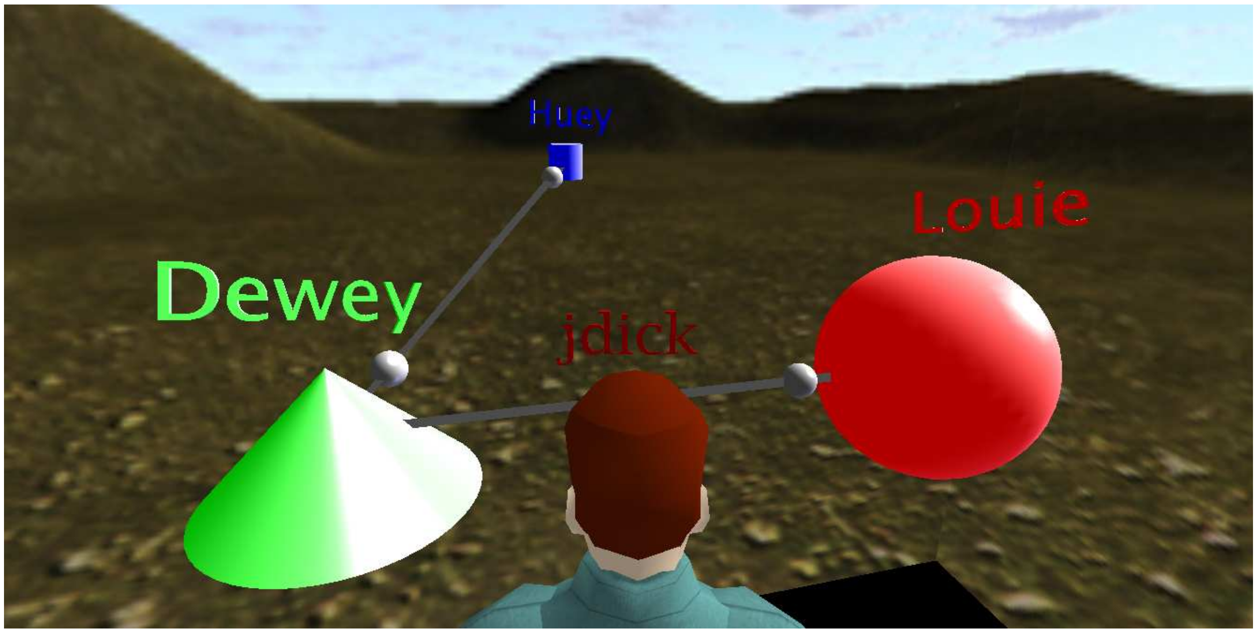


Figure 9: WonderBlocks shading, directional connections and auto-rotating labels

WonderBlocks Frontend, continued

For the WonderBlocks user interface, we decided to use the same type of heads-up display (HUD) menu buttons that the PDF Viewer and HTML Viewer both use. We spent a lot of time deciding how the user should be able to trigger and dismiss this HUD menu. After some experimenting, we decided that the HUD should be triggered two ways, either by clicking on any block or connection, or by simply walking in the near vicinity of the WonderBlocks in-world. Initially, the HUD automatically dismissed itself whenever the user walked away from the WonderBlocks in-world, but we realized this created a usability problem; if a user was editing a WonderBlocks diagram and stepped back to get a better look, the HUD would close. Because of this, we decided that the HUD should always be manually dismissed by the user, upon clicking a 'close' button. In essence, the HUD can be automatically or manually triggered, but it is always manually dismissed.

We decided that the HUD would work using a modal interface. The HUD includes buttons for ‘block creation mode’, ‘connection creation mode’, ‘editing mode’, ‘deletion mode’, a save button, and a close button. Modes can be switched simply by clicking any mode button. Modes can be exited by either clicking on any mode button twice (to toggle it,) or by closing the HUD entirely. Whenever a single client adds, deletes, or edits a connection or block, the change is visually reflected on all connected clients.

In block creation mode, a dialog box pops up for the user to enter information specific to the new block like block size, color, etc. After clicking ‘OK’, the block is added in-world. In connection creation mode, clicking on one block and then another block triggers a dialog box that pops up for the user to enter information about the connection. After clicking ‘OK’, the connection is added in-world. Connections can optionally have a label, and can optionally be directional, bidirectional, or non-directional. Directional connections always point from the first block clicked to the second block clicked. To reverse a connection’s direction, it has to be deleted and then recreated by clicking the blocks to be connected in the opposite order. This obviously is not an issue for bidirectional and non-directional connections. In editing mode, clicking on any block or connection will display a dialog box allowing the user to change the properties it was last assigned. After clicking ‘OK’, the changes made are reflected in-world. In deletion mode, clicking on any connection instantly removes it from the world. Clicking on any block instantly removes it and all connections associated with it from the world. Clicking the ‘Save’ button in the HUD instructs the Wonderland server to serialize the current WonderBlocks diagram/state to an XML file that will be reloaded when the server restarts. Whenever the ‘Save’ button is clicked on any client, all connected clients display a status message that notifies the user that the diagram was saved.

We believe that in general, WonderBlocks' modal interface is easy to learn and use. We realize that this modal interface does have some downsides. For instance, if a user is in "deletion mode" and does not realize it, and then clicks on a block, that block and all of its connections will be deleted and irrecoverable. An undo feature (discussed in the "Future Work" section of this paper) would help to make this less of an issue.

WonderBlocks Backend, continued

The WonderBlocks client/server messaging protocol code constantly evolved as the features mentioned above were added and implemented. The messaging protocol is at the heart of WonderBlocks. Messages are sent between the client and server for any user-initiated action. For instance, when a client is in connection creation mode and blocks are clicked, the client sends the server a message informing the server that a connection should be created. This message contains all of the information necessary to make the connection, including which two blocks should be connected, and the connections name and direction, if specified. The server then inspects its own internal state of the running WonderBlocks application. If the two blocks specified in the message are already connected in the server's internal state, or if the two blocks clicked on actually happen to be the same block, the server will realize this and will then send the requesting client a message instructing that particular client to display an appropriate error message. If the connection was successfully created in the server's internal state, it sends a message to all connected clients (including the requesting client) instructing those clients to draw the new connection locally.

All interactions with WonderBlocks work this way; messages always originate at one client and are sent to the server. Then, the server will either generate an appropriate message to be sent to that particular client (usually in the case of an error) or to all clients including that particular client (usually in the case of a valid operation.) A user's actions trigger clients to generate messages and send them to the server, but clients never independently act on a user's actions (only on messages received from the server). We chose this approach to avoid synchronization issues.

For example, in the case of deleting a block, assume that a client deletes a block locally, before sending a message to the server saying "tell all other clients to also delete this block". Meanwhile, another client has requested that two blocks be connected, and one of the blocks involved in the connection is the one that the first client has just deleted locally. The first client would not be able to create the connection if instructed to by the server, because it has already locally deleted a necessary block, but all other clients will be able to. This would mean that the clients would be in an inconsistent state, and this is the exact reason that we did not implement the WonderBlocks messaging protocol in this fashion. Because all clients request state changes from the server when instructed by a user, but only perform those changes when instructed to by the server, all clients stay in the same synchronized state. The Wonderland server automatically handles request concurrency; if two clients make conflicting requests simultaneously, the server will process those requests in a transactional way and will ensure that a consistent state is still maintained. See the diagram below for a depiction of the client → server → client messaging process just described.

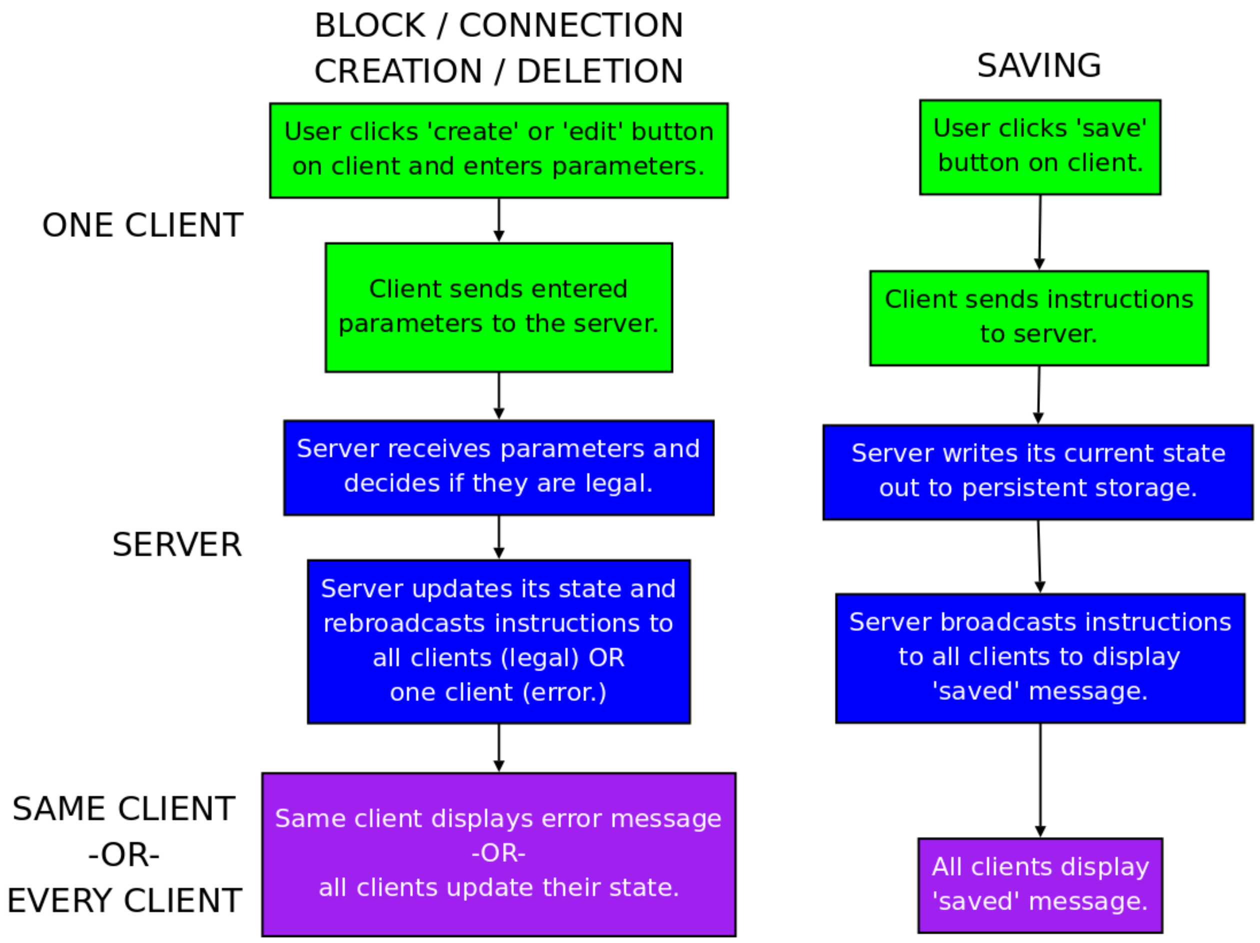

BLOCK / CONNECTION CREATION / DELETION
SAVING
ONE CLIENT
User clicks 'create' or 'edit' button on client and enters parameters.
Client sends entered parameters to the server.
User clicks 'save' button on client.
Client sends instructions to server.
SERVER
Server receives parameters and decides if they are legal.
Server updates its state and rebroadcasts instructions to all clients (legal) OR one client (error.)
Server writes its current state out to persistent storage.
Server broadcasts instructions to all clients to display 'saved' message.
SAME CLIENT -OR- EVERY CLIENT
Same client displays error message -OR- all clients update their state.
All clients display 'saved' message.

Figure 10: Two parts of the WonderBlocks messaging protocol

Implementing the messaging protocol this way requires the server and all clients to independently run their own respective copies of the WonderBlocks backend code. Otherwise, clients would not know what to do when receiving messages like 'delete the block with this id'. Even so, the messaging protocol ensures that the running state of the application is consistently synchronized across all clients and the server. Each client's backend state is independently updated and each client independently reflects visual changes in-world upon receiving a message from the server.

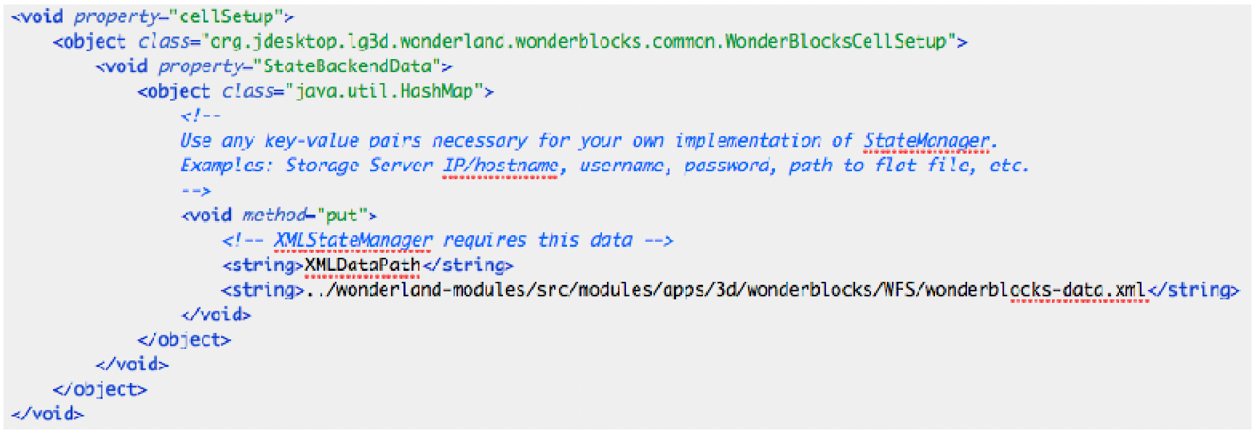
Since the initial development of the WonderBlocks backend occurred completely outside of Wonderland, a few unanticipated problems with the backend code relating to synchronization and concurrency came up after running the same code inside Wonderland. We realized that using singletons for the manger classes creates unnecessary redundancy and complications when the backend code runs inside of Wonderland. It turned out that Wonderland's Darkstar backend automatically provides the same protection for the manager classes that was previously provided by the singleton design pattern. Keeping the manager classes as singletons was now not only redundant, but also caused random crashes because of conflicts with the Darkstar backend. Naturally, we removed the singleton functionality from the manager classes. (For simplicity, the WonderBlocks backend design notes in this paper's appendix still assume the backend classes are still singletons.)

We also found another interesting concurrency-related bug in the backend code once it was running inside Wonderland that had not surfaced when it was running independently. During functional testing, we noticed that some operations would be randomly ignored, the client-side and server-side manager states would go out of sync, and a ConcurrentModificationException would be thrown by both the Wonderland client and server. This problem appeared in two places in the backend code: a single HashSet in memory was being iterated through in one class, while another class simultaneously removed elements from it. Operations like that are not thread-safe, which is why that problem surfaced. The solution turned out to be simple: a copy of the HashSet to be altered is created, and then the copy is iterated through while the original is altered. A more elegant solution would have been to switch from using foreach loops to using Java Iterators with Iterator.remove(), etc, but doing so would have required extensive backend refactoring. Although creating copies of the HashSets in question is less memory-efficient than using Iterators, doing so was a quick and acceptable solution to the problem. Interestingly, the functional tests that revealed the problem involved block deletion.

Final Steps

At this point, our seven weeks at Sun were nearly over, and WonderBlocks was nearly complete. A few more features were added to the WonderBlocks backend. On the server side, WonderBlocks always serialized its state data to a static XML file on the Wonderland server. What if a developer would rather use a database to store state data, or another method entirely? To give developers this freedom, we added an abstract class called StateManager that allows for persistent data storage in the WonderBlocks backend. This abstract class contains abstract methods allowing for serialization and deserializtion (storage and retrieval) of StateContainer objects, which encapsulate the two manager classes. Together, the two manager classes encapsulate the entire running state of an instance of the WonderBlocks module. We then wrote a class called XMLStateManager that implements the StateManager class. It performs the functionality we had already written of serializing and deserializing the manager classes to/from XML files, while also utilizing the abstract class/"generic" data storage code we wrote.

A feature we added to the "generic" data storage system is the ability to store any kind of textual key-value pairs in the Wonderland WLC XML file. This way, per-installation information can be stored in and read from the WLC XML file, and easily modified by a Wonderland server administrator who does not necessarily have Java programming knowledge. For example, if a developer was to write a MySQLStateManager class, necessary information specific to MySQL like "hostname=mysql.myserver.com, dbname=wonderland, username=sqluser, password=sqlpass," etc, can be stored in the Wonderland WLC XML file and changed easily. For our XMLStateManager class, the following information needs to appear in the WLC XML file:



```
<void property="cellSetup">
    <object class="org.jdesktop.lg3d.wonderland.wonderblocks.common.WonderBlocksCellSetup">
        <void property="StateBackendData">
            <object class="java.util.HashMap">
                <!--
                Use any key-value pairs necessary for your own implementation of StateManager.
                Examples: Storage Server IP/hostname, username, password, path to flat file, etc.
                -->
                <void method="put">
                    <!-- XMLStateManager requires this data -->
                    <string>XMLDataPath</string>
                    <string>../wonderland-modules/src/modules/apps/3d/wonderblocks/NFS/wonderblocks-data.xml</string>
                </void>
            </object>
        </void>
    </object>
</void>
```

Figure 11: Information in a WLC XML file needed by the XMLStateManager class

As you can see, the XMLStateManager class relies on a piece of information called "XMLDataPath", which specifies the pathname that the WonderBlocks state XML file should be serialized to/deserialized from. Because any kind of textual key-value pairs can be read from the WLC XML files, and because of our abstract StageManager class, we believe we've given future WonderBlocks developers a lot of freedom to be able to implement persistent data storage in whatever way they may prefer.

One final feature we added to the WonderBlocks backend was consistent logging functionality through the use of Java's built in Logger class. Whenever any WonderBlocks state change occurs on the clients or server, detailed information about the change is logged. All log entries are written in a consistent way that should be able to be easily parsed using regular expressions, etc. This way, if team members or a supervisor (in the case of the virtual team room, a professor) want(s) to see how a particular instance of the

WonderBlocks module is being used, detailed logs could be exposed (and then potentially parsed) by intercepting the output of the Logger object.

At this point, after flattening a few minor last-minute bugs, we had finished developing WonderBlocks. However, there was still one more task to complete before our time at Sun was over.

The Virtual Team Room

The original project vision was to build a virtual team room inside Wonderland. At this point, we had created components that we thought would be useful for a virtual team room, as well as for Wonderland in general, but we had not spent a lot of time working on creating the virtual team room itself. We quickly created a team room using Wonderland World Builder that showcases both of our modules, as well as the preexisting PDF Viewer and whiteboard modules. Our team room has a main area (for all teams), three smaller meeting areas (for individual teams), and an outside "deck" area to showcase the WonderBlocks component. After completing our work on the virtual team room, our time at Sun had come to a close.

Results and Conclusions

Results

Our original project vision was to create a virtual team room using Sun's open-source virtual world toolkit, Project Wonderland. After completing our work, we feel that the accomplishments that we made can contribute to the team room concept as well as to Wonderland's open source community as a whole.

Our sponsors at Sun were satisfied with the work that we did during our stay at Sun. We were able to adapt quickly to the technology and to the 3D mindset, and our sponsors took notice. The code we wrote at Sun is now in a repository that is accessible by Sun employees and the Wonderland development community, for anyone to use and/or improve upon. This allows for plenty of possibilities for future work on (or applications of) our modules.

The size of our project is large considering the short duration of the project. The HTML Viewer consists of a total of 1,267 lines of Java code, and WonderBlocks consists of 2,782 lines of Java code.[19] These statistics do not incorporate other files (such as WLC XML files) needed by the Wonderland server to actually place the HTML Viewer and WonderBlocks modules in-world. In the Sun Labs environment, we were able to run extensive functional tests on the modules as we wrote them, and did so at a comfortable pace thanks to the performance capabilities of the lab machines we were provided with.

[19] Measured using David A. Wheeler's 'SLOCCount' program.

Future Work

We are satisfied with the work that we have done in the seven weeks we spent at Sun Microsystems. However, there are certainly some things that could be done to expand upon the work we have completed.

We created the HTML Viewer not only to expand the Project Wonderland toolkit, but to also get accustomed to the project. Here are some improvements and features that could be made/added to the HTML viewer in the future:

- Scrolling/panning around a zoomed-in page using the keyboard and mouse.
- Automatically refreshing a given page using a timer.
- Browser-like features, such as clickable links, forward/back buttons, and an address bar rather than an address dialog.

The WonderBlocks module also has many possibilities for expansion:

- Re-drawing *parts* of WonderBlocks diagrams when those parts change, as opposed to always re-drawing entire diagrams
- Positioning blocks with mouse dragging, instead of entering 3D coordinates in dialog boxes
- Continuing to display the mode indicator in the HUD after status messages such as "State Saved" are displayed
- Making connections appear more 3D by using cylinders instead of Java3D lines
- Custom metadata for each block

- "Sky view": In a meeting, if someone is assigned a task, that task could float up to the ceiling and all of the tasks combined would give the group an idea of what their progress is and what to expect in the near future
- An 'undo' feature to revert accidental modifications

There is also future work for WonderBlocks that is already lined up for other students. The title of the work is "3D Brainstorming Tool Based on WonderBlocks"[20] and the concept behind the project is to make WonderBlocks act like 3D sticky notes, and to export the organization of the blocks to a spreadsheet.

Finally, our project focused on modules for a virtual team room in Wonderland. Although we created a prototype team room to show off these modules, the team room itself could be significantly expanded and improved upon. Another Major Qualifying Project group at WPI is currently doing work on the virtual team room. The virtual team room will consist of the Wonderland modules that we have created as well as preexisting modules (such as Wonderland's Software Phone). The virtual team room might also contain new modules currently being developed by various WPI MQP groups.

Conclusion

We have contributed modules to Wonderland's open-source community that are useful and include many possibilities for future expansion. The modules we created also helped enforce our initial project vision of creating a virtual team room in Wonderland.

The work we have done at Sun Microsystems did not only benefit us, but it benefitted our advisors as well. We were able to help our WPI advisor by creating tools that can be used to augment one of the courses he teaches at WPI. Our WonderBlocks module

[20] (Sun Microsystems, Inc, 2008)

will be expanded in some of the ways described in the “Future Work” section of this paper, as part of a separate WPI MQP project. Through our work on WonderBlocks, our Sun advisors acquired a tool that shows off Wonderland’s capabilities and utilizes the 3D space to visualize data in a way that has not been done before. WonderBlocks can also act as a foundation for future work to be done in Sun’s virtual world toolkit.

We have learned a great deal during our time on this project. One of the concepts we were able to become familiar with was what it was like to have a career in the software industry. We commuted to Sun Microsystems every day to work in an office environment, while collaborating with team members and taking suggestions from a manager. We also enjoyed the Sun campus and the corporate culture at Sun. We learned a great deal about adapting to develop on software platforms that we have not seen or used before. This is the first time that either of us has developed software that works with 3D graphics, and now we are both open-source contributors.

Overall, the work we have done in creating these modules for a virtual team room was a success. While the end result does not exactly match our initial project vision, we were able to keep all of our advisors satisfied, both at WPI and at Sun Microsystems. We hope that our work will continue to be expanded upon in the future, as it is part of Sun's expansive open-source community, as well as of WPI Major Qualifying Projects that are currently in progress. Through all of the hard work we have done, we have learned a great deal, and will not soon forget the lessons we have learned while completing this project.

Appendix

WonderBlocks Backend Design Notes

Background Information

This section of the appendix details the design of the WonderBlocks backend, which executes on the Wonderland server. It should be noted that every Java class in the WonderBlocks backend follows the JavaBeans convention, and can be placed into one of three categories:

1. Classes representing objects that will be displayed in-world (Block, BlockConnection)
2. Classes that manage in-world objects (BlockManager, BlockConnectionManager)
3. Classes that assign IDs that managers can map in-world objects to (BlockID, BlockConnectionID)

Architecture

IDs

Each object that can be displayed in-world (Blocks and BlockConnections) has an associated ID number that is represented by a separate object (a BlockID or a BlockConnectionID.) This way, only one copy of each Block and BlockConnection object exists in memory, but they can be referenced multiple times by other objects via their IDs. It is the responsibility of the manager classes (BlockManager and BlockConnectionManager) to actually map IDs to the objects they represent. The ID classes statically store the next ID number to be assigned, so that each time a new ID object is instantiated, the ID number stored inside it is automatically incremented. Separate ID classes exist for each object type so that BlockIDs and BlockConnectionIDs can increment independently.

Blocks

The Block class represents a single WonderBlock. This class stores information such as WonderBlock shape, size, color, and attribute/value pairs (metadata.) Blocks can also be connected together, but they do not retain connection information directly; connections are actually represented by a separate BlockConnection class. To that end, each Block maintains a list of BlockConnectionIDs, where each BlockConnectionID is associated with the BlockConnection that includes that particular Block.

BlockConnections

A connection between two Blocks is represented by the BlockConnection class. There are special rules governing connections.

Namely:

1. No Block can be connected to itself.
2. No two Blocks can share more than one connection between them.
3. A Block can have any number of connections to other Blocks, provided that rule 2 is not broken.
4. A connection between any two Blocks can be directionless, directional, or bidirectional.

These rules are handled by the Block Connection Manager, which will be discussed later.

A BlockConnection object stores information about the two Blocks that share that connection, as well as the direction in which the Blocks are connected. An advantage of having distinct BlockConnection objects is that it is possible for connections to have metadata associated with them that is independent of the two Blocks sharing the connection. (For example, each connection could store a weight, which would allow one to create a weighted graph.)

It was mentioned before that each Block maintains a list of BlockConnectionIDs that are associated with connections involving that Block. In actuality, only one Block in a given connection actually keeps track of that connection's ID in its list of BlockConnectionIDs; the other Block stores no information about that particular connection and does not 'know' it is part of a connection. The BlockConnection class employs a loose parent/child analogy to distinguish between the two Blocks sharing the connection. For a given connection, the parent Block is the one that stores that connection's ID.

This analogy makes the most sense for directional connections; a parent Block always points to a child Block, so the parent Block keeps track of that connection's ID. The analogy works similarly but is less logical for directionless and bidirectional connections. Although there is not a parent/child relationship between blocks that are connected bidirectionally or with no direction, the connection is still represented as a parent/child relationship internally. Regardless of the connection's direction, only the parent Block 'knows' that it is part of a connection.

See the figure below for a diagram depicting the relationships between Blocks and BlockConnections that were just described.

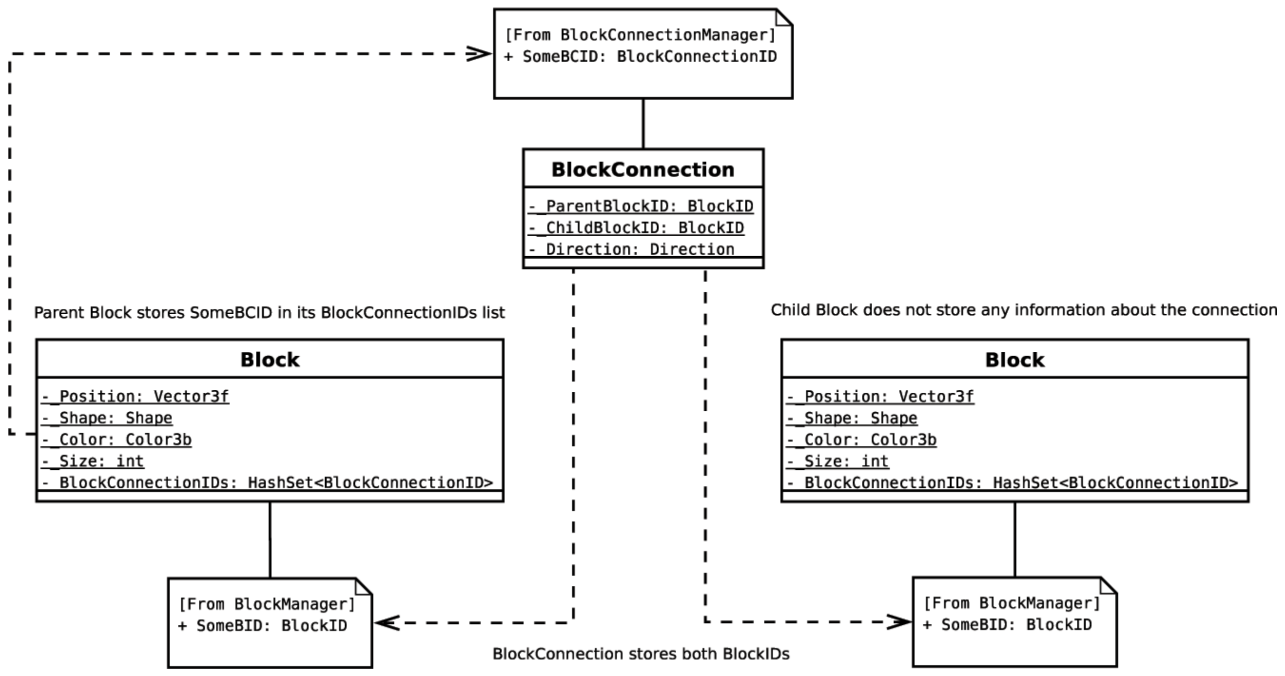
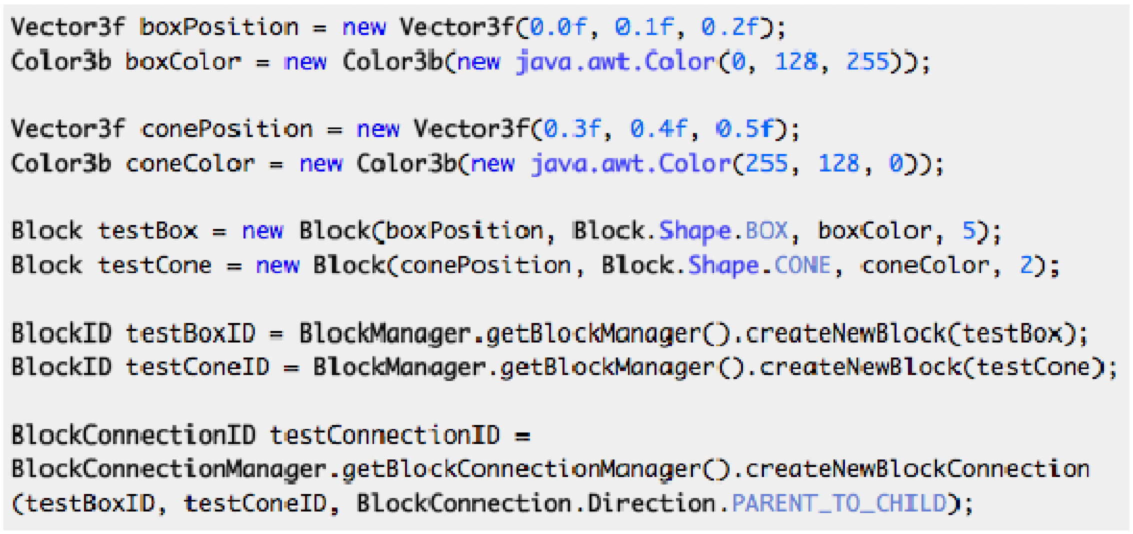

[From BlockConnectionManager]
+ SomeBCID: BlockConnectionID
BlockConnection
- ParentBlockID: BlockID
- ChildBlockID: BlockID
- Direction: Direction
Parent Block stores SomeBCID in its BlockConnectionIDs list
Child Block does not store any information about the connection
Block
- Position: Vector3f
- Shape: Shape
- Color: Color3b
- Size: int
- BlockConnectionIDs: HashSet<BlockConnectionID>
Block
- Position: Vector3f
- Shape: Shape
- Color: Color3b
- Size: int
- BlockConnectionIDs: HashSet<BlockConnectionID>
[From BlockManager]
+ SomeBID: BlockID
[From BlockManager]
+ SomeBID: BlockID
BlockConnection stores both BlockIDs

Figure 12: Depiction of the representation of a connection between two Blocks.

The most complex classes in the WonderBlocks backend are the two manager classes, BlockManager and BlockConnectionManager. To facilitate easy sharing of the backend's state between its component classes, the WonderBlocks manager classes are singletons. They should be invoked accordingly by calling *Manager.get*Manager(). Normally, the singleton design pattern mandates that singleton classes should have protected constructors, so that the classes cannot be manually instantiated (which would cause data synchronization problems.) The WonderBlocks manager classes DO NOT have protected constructors. Instead, these classes have public constructors, which are necessary for them to follow the JavaBeans convention. Therefore, be very careful when working with the manager classes. They should NEVER be manually instantiated.

The manager classes are the backbone of the WonderBlocks backend. Each manager abstracts away Block or BlockConnection objects so that they can be referred to and manipulated by ID number anywhere in the code, any number of times, while always maintaining a single copy of a given Block or BlockConnection object in memory. See the figure below for a simplified code listing demonstrating how Blocks, BlockConnections, and their respective manager classes work together to represent the overall state of an instance of the WonderBlocks module. For more specific information regarding how the manager classes should be used and manipulated, please see the WonderBlocks API documentation.



```
Vector3f boxPosition = new Vector3f(0.0f, 0.1f, 0.2f);
Color3b boxColor = new Color3b(new java.awt.Color(0, 128, 255));

Vector3f conePosition = new Vector3f(0.3f, 0.4f, 0.5f);
Color3b coneColor = new Color3b(new java.awt.Color(255, 128, 0));

Block testBox = new Block(boxPosition, Block.Shape.BOX, boxColor, 5);
Block testCone = new Block(conePosition, Block.Shape.CONE, coneColor, 2);

BlockID testBoxID = BlockManager.getBlockManager().createNewBlock(testBox);
BlockID testConeID = BlockManager.getBlockManager().createNewBlock(testCone);

BlockConnectionID testConnectionID =
BlockConnectionManager.getBlockConnectionManager().createNewBlockConnection
(testBoxID, testConeID, BlockConnection.Direction.PARENT_TO_CHILD);
```

Figure 13: Blocks, BlockConnections and the managers working together.

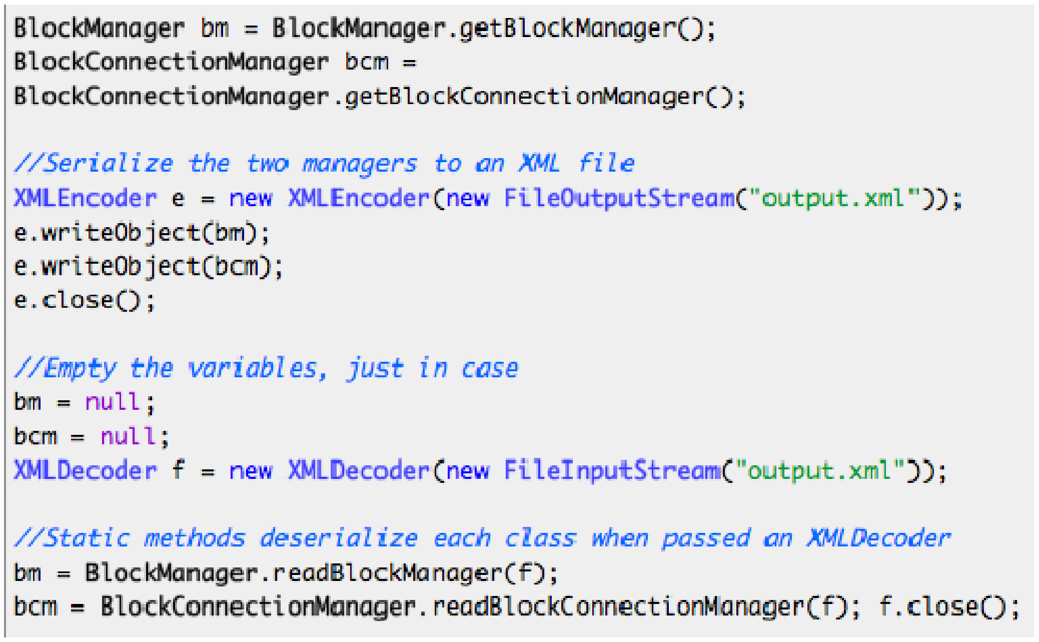
Data Persistence

Wonderland is built on top of Sun's Project Darkstar game framework, which includes a Berkeley DB database for persistent storage. Unfortunately, the Wonderland 0.4 server erases this database every time the server is restarted, so the WonderBlocks component uses a different approach for persistent storage. Because every WonderBlocks class follows the JavaBeans convention, each of them can be serialized and stored on disk.

At a later time, they can then be read back from the disk and be deserialized back into Java objects.

Remember that the manager classes abstract away Blocks and BlockConnections, and that Blocks and BlockConnections themselves refer to each other internally via ID numbers that are assigned by the managers. What this means is that serializing the manager classes to disk stores the entire running state of an instance of the WonderBlocks component, without including any redundant information (multiple copies of the same object.) Upon deserializing the saved data, WonderBlocks execution can resume as if the application had never terminated.

Note that the two managers have to be serialized to and deserialized from disk in the same order. In our current implementation of the WonderBlocks backend, the two managers are [de]serialized to/from XML files using Java's built-in XMLEncoder and XMLDecoder classes. See the figure below for a simplified code listing that demonstrates [de]serialization of the two manager classes to/from XML files. We've created generic interfaces to allow for [de]serialization, so that developers using the WonderBlocks backend will have more options for implementing data persistence. In other words, rather than saving all data to XML files, a developer may wish to [de]serialize data to/from a database, etc.



```
BlockManager bm = BlockManager.getBlockManager();
BlockConnectionManager bcm =
BlockConnectionManager.getBlockConnectionManager();

//Serialize the two managers to an XML file
XMLEncoder e = new XMLEncoder(new FileOutputStream("output.xml"));
e.writeObject(bm);
e.writeObject(bcm);
e.close();

//Empty the variables, just in case
bm = null;
bcm = null;
XMLDecoder f = new XMLDecoder(new FileInputStream("output.xml"));

//Static methods deserialize each class when passed an XMLDecoder
bm = BlockManager.readBlockManager(f);
bcm = BlockConnectionManager.readBlockConnectionManager(f); f.close();
```

Figure 14: Example of [de]serialization of the manager classes to/from XML.

Works Cited

Brown, M. (2008). *Images of Websites*. Retrieved 12 12, 2008, from Martin Brown: http://mwbrown.org/webshot.shtml

CollegeOTR.com, T. W. (2008, December 11). *College OTR: FCC Commissioner: World of Warcraft Makes College Kids Drop Out*. Retrieved December 11, 2008, from College OTR: http://www.collegeotr.com/college_otr/fcc_commissioner_world_of_warcraft_makes_college_kids_drop_out_16679

IBM. (2008, July 21). *IBM Business Center*. Retrieved January 2, 2009, from IBM Business Center: http://www.ibm.com/3dworlds/businesscenter/us/en/

JME. (2008). *JMonkeyEngine.com*. Retrieved December 11, 2008, from JMonkeyEngine.com web site: http://www.jmonkeyengine.com

Lamb, G. M. (2006, October 5). *At colleges, real learning in a virtual world - USATODAY.com*. Retrieved December 11, 2008, from USATODAY.com: http://www.usatoday.com/tech/gaming/2006-10-05-second-life-class_x.htm

Linden Research, Inc. (2008). *Second Life | What is Second Life?* Retrieved December 11, 2008, from Second Life Web site: http://secondlife.com/whatis/

Linux Information Project. (2006, March 29). *The X Window System: A Brief Introduction*. Retrieved 1 12, 2009, from Linux Information Project: http://www.linfo.org/x.html

MacBlogz.com, A. o. (2008, May 14). *Second Life Apple Store. | MacBlogz - One Stop Apple News*. Retrieved December 11, 2008, from MacBlogz - One Stop Apple News: http://www.macblogz.com/2008/05/14/second-life-apple-store/

Second Life Business Communicators Wiki. (2007, 12 31). *Brands with a Second Life Presence List*. Retrieved 12 12, 2008, from Second Life Business Communicators Wiki: http://slbusinesscommunicators.pbwiki.com/FindPage?RevisionsFor=Companies+in+Second+Life

Slott, J. (2008, 07 31). *Extending Project Wonderland by Creating New Cell Types*. Retrieved 12 12, 2008, from Project Wonderland: http://wiki.java.net/bin/view/Javadesktop/ProjectWonderlandNewCell

Sun Microsystems, Inc. (2008, December 01). *TWiki . Javadesktop . StudentProjects*. Retrieved December 11, 2008, from TWiki . Javadesktop web site: http://wiki.java.net/bin/view/Javadesktop/StudentProjects

Sun Microsystems, Inc. (2008, February 14). *Lesson: JavaBeans Concepts*. Retrieved January 12, 2009, from The Java Tutorials: http://java.sun.com/docs/books/tutorial/javabeans/whatis/index.html

Sun Microsystems, Inc. (2007). *lg3d-wonderland: Project Wonderland*. Retrieved December 01, 2008, from lg3d-wonderland: Project Wonderland Web site: https://lg3d-wonderland.dev.java.net/

Sun Microsystems, Inc. (2008). *Project Looking Glass*. Retrieved December 11, 2008, from Sun Microsystems web site: http://www.sun.com/software/looking_glass/

Sun Microsystems, Inc. (2008, October 7). *Sun Labs - Project Wonderland - Go ahead, make a scene*. Retrieved January 2, 2009, from Sun Microsystems: http://research.sun.com/spotlight/2008/2008-08-19_project_wonderland.html

Sun Microsytems, Inc. (2008, 8 30). *Project Wonderland User FAQ (Frequently Asked Questions)*. Retrieved 12 12, 2008, from java.net: http://wiki.java.net/bin/view/Javadesktop/ProjectWonderlandEndUserFAQ

Weisstein, E. W. (2008). *Euler Angles*. Retrieved 12 15, 2008, from Wolfram MathWorld: http://mathworld.wolfram.com/EulerAngles.html

Wikipedia. (2008, 12 10). *Euler Angles*. Retrieved 12 15, 2008, from Wikipedia: http://en.wikipedia.org/wiki/Euler_angles