FEASIBILITY OF THE NINTENDO DS FOR TEACHING PROBLEM-BASED LEARNING IN KINDERGARTEN THROUGH TWELFTH GRADE STUDENTS

A Thesis

presented to

the Faculty of the Graduate School

at the University of Missouri-Columbia

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

MICHAEL LIPINSKI

Dr. Otho Plummer, Thesis Supervisor

August 2008

The undersigned, appointed by the dean of the Graduate School, have examined the thesis entitled

Feasibility of the Nintendo DS for Teaching Problem-Based Learning in Kindergarten through Twelfth Grade Students

presented by Michael Lipinski,

a candidate for the degree of Masters of Science,

and hereby certify that, in their opinion, it is worthy of acceptance.

Professor Otho Plummer

Professor William Harrison

Professor Michael Hosokawa

Acknowledgements

First off I would like to thank my loving family for all their support over the years. If it was not for them and their sacrifices I would not be writing this page today. I would also like to thank my close friends (anyone who could be considered part of the "Como crew"), my family away from my family. They helped me laugh through the sad times, supported me through the tough times, and entertain me through the boring times. They were always willing to lend a hand and an ear and I love them all dearly for it. That goes double for my best friend and partner in crime, Mike Torpea. I would also like to thank my first advisor, my boss, and my friend Dr. Ali Hussam. I would not have even been in graduate school if not for his "gentle" pushing and the job he gave me to help pay for it. I deeply appreciate all of the opportunities he has given me and will remember the lessons I learned under him forever. Next I would like to thank Dr. Otho Plummer who became my advisor when politics got in the way of an education. Dr. Plummer has been as kind, helpful, and understanding as an advisor could be and that says a lot considering he came in half way through my project. I would also like to thank Dr. William Harrison and Dr. Michael Hosokawa for being on my thesis committee. Finally, and most importantly, I would like to thank my Grandfather, Sam Mandina, who bought me my first computer when I was three years old and helped foster my love of technology. This thesis paper and my Master's degree are dedicated in his memory.

Table of Contents

List of Figures

List of Tables

List of Registers

List of Code Samples

Abstract

Problem-based learning (PBL) is an instructional approach that has been employed successfully since the 1960s and continues to gain mainstream acceptance in many areas of study. PBL is an instructional, and curricular, learner-centered approach that empowers learners to conduct research, integrates theory and practice, and applies knowledge and skills to develop a viable solution to a defined problem (2). In order to gain a wider acceptance in K-12 classrooms, PBL might need to take a technological approach. But with technology, just like education, there are many possible solutions to consider in solving a problem. This paper will discuss a proof of concept prototype built for the Nintendo DS to further PBL education. Because of the built-in Wi-Fi access, massive textual input possibilities, and existence of a large hobbyist development community, the Nintendo DS is the clear choice for this prototype. Many features of the Nintendo DS (and other video game consoles) will be discussed in length in this paper such as backgrounds, sprites, tilesets, input, and Wi-Fi.

INTRODUCTION Chapter 1

Technology is a constantly changing and evolving medium. Because of the increase in advantages and improvements that technology provides us, we must constantly re-evaluate the potential it has to do old things in new exciting ways. In recent years, a new teaching style has started to develop and has proven to provide increased learning benefits to participants. This learning style is called Problem-Based Learning and is starting to garner international recognition of a new way of teaching. Problem-Based Learning, or PBL, is a teaching approach that involves a student-centered instructional strategy in which students collaboratively solve problems and reflect on their experiences. In this process students are encouraged to learn by being asked to solve challenging, open-ended problems while working in small collaborative groups. During the PBL process teachers take on less of an "instructional" role and more of a "facilitator of learning" role. With its collaborative, open-ended, and modular approach to learning, PBL is a perfect candidate for receiving a helping hand from current technology to allow it to spread further and easier. This paper will look briefly at the advantages of PBL, what types of technology could help to make PBL easier to use for the masses, and finally in-depth inspection of the proposed solution of using the Nintendo DS to provide a cheap, economical, and user

friendly interface for PBL for Kindergarten through Twelfth Grade (K-12) students.

Problem-Based Learning (PBL) is a curriculum and instructional innovation first adopted in medical education at McMaster University in the mid-1960s. By 1992, approximately 60 medical schools worldwide had implemented, at least in part, a problem-based learning approach in their curricula (1). PBL has spread to other professional fields such as veterinary medicine, agriculture, natural resources, health related professions (such as occupational therapy), and educational leadership. Furthermore, there is a growing interest in PBL approaches in public schools, especially in the areas of math and science (10). Applications of this method are possible in almost every course of study. This growing support for PBL in professional and educational institutions lends itself to a need for technological support. Currently a majority of PBL systems are done with pen and paper, in which students receive static chunks of information after they achieve certain milestones in a PBL Case. Since PBL has been structured to be a dynamic and individual learning experience, static pen and paper seem to hinder the full vision of PBL. Technology can help this exciting new idea reach its full potential.

To provide PBL to a wider audience, such as Kindergarten through Twelfth Grade (K-12) students, many factors need to be

weighed. Robustness in application, development time, accessibility to teachers and students, and cost are some of the many parameters relevant to a possible PBL technological solution. Some possible solutions that will be discussed in this paper will be individual modules downloaded onto group laptops, students having access to a web application environment, and modules running on the Nintendo DS that communicate with web applications and databases seamlessly from the user's point of view. Each solution presents both positives and negatives, and they will be discussed briefly in this paper.

Finally, the proof of concept project of this paper is the Nintendo DS solution. This paper will discuss both the merits of this solution and a broader understanding of how this solution works. While there is research into the viability of other solutions, this is the solution that actually deserves a proof of concept working prototype. The Nintendo DS is a very sophisticated piece of hardware that has a large "hobby" or "homebrew" community around it. This community has developed many tools that make development on the Nintendo DS simpler and more accessible to many programmers. Programming for the Nintendo DS requires the detailed use of memory registers to store the manner in which the hardware interprets the data that is stored in memory. Both tools and registers of the Nintendo DS will be discussed in detail.

This paper will briefly discuss the procedures and benefits of PBL, how technology could make this new teaching style the norm for many people, and the proof of concept prototype of how PBL could be taught using the Nintendo DS. PBL is a new and exciting way to teach students that encourages them to discover solutions for themselves, which may lead to more knowledge retention. Arguably, the best way to bring PBL to the masses is some type of technological solution. Of all the solutions that were investigated, the Nintendo DS seems the most feasible in the case of teaching K-12. The power of turning self-discovery learning into a video game is very powerful for this age group. The further investigation of how to achieve this solution makes up the bulk of this thesis discussion.

PROBLEM-BASED LEARNING Chapter 2

Problem-based learning (PBL) is an instructional approach that has been employed successfully since the 1960s and continues to gain mainstream acceptance in many areas of study. PBL is an instructional, and curricular, learner-centered approach that empowers learners to conduct research, integrates theory and practice, and applies knowledge and skills to develop a viable solution to a defined problem (2). PBL evolved from innovative health sciences and medical curricula introduced in the 1960s in North America. Medical education, with its intensive pattern of basic science lectures followed by an equally exhaustive clinical teaching program, was rapidly becoming an ineffective and inhumane way to prepare students, given the explosion in medical information and new technology and the rapidly changing demands of future practice (3). Medical faculty at McMaster University in Canada introduced a tutorial process, not only as a specific instructional method, but also as central to their philosophy for structuring an entire curriculum promoting student-centered, multidisciplinary education, and lifelong learning in professional practice (4).

PBL is an instructional method that encourages students, or learners, to conduct research, test theories and practices, and use

knowledge and skills to develop a viable solution to a defined problem. Crucial to the learner's success in PBL is the presentation of "ill-structured" problems, by tutors or educators, which often involve facets of many disciplines. Furthermore, the learners require a tutor who can guide them through the learning process without directly teaching or telling the learners what to learn. When the learning experience concludes, the tutor provides a thorough debriefing, not only to guarantee that learners are exposed to the same knowledge, but to allow learners to reflect on what they have discovered throughout the solution process. These methods used in PBL help to develop specific skills that include the ability to think critically; analyze and solve complex, real-world problems; find, evaluate, and use appropriate learning resources; work cooperatively; demonstrate effective communication skills; and use content knowledge and intellectual skills to become continual learners (5).

There are many ideas behind why PBL is a better way for students to learn. The most important, and probably fundamental, idea behind PBL is that learners must be responsible for their own learning. Learners tackle problems in PBL with only their own knowledge and experience base. In order for learners to overcome new problems presented to them, they are required to learn on their own and also figure out what to learn. Basically, PBL requires learners

to help set their own curriculum. PBL also requires that the problems in the program be "ill-structured". In this way, learners get to practice on problems that are more like the real world and may not have a simple or "clean" solution. By having ill-structured problems, the learners develop critical thinking skills that can be readily applied to the real world. In addition to being ill-structured, problems should also require knowledge from many disciplines. Not only does this help replicate real world problems more thoroughly, but it allows the learner to see the solutions from many perspectives and reinforce knowledge. Collaboration is essential in PBL because it helps emulate the real world even further because outside of school many learners will have to share information and collaborate on a problem to develop a solution. This also helps learners to share information with their classmates so that all learners can benefit from the work of a single learner. All individual work must be shared in this way to help redirect and reanalyze any work the group does.

The closing analysis, or debriefing, of the problem and the solution is crucial. This discussion allows learners to discuss what concepts and principles they discovered did and did not work. At the same time, the debriefing allows the tutor the opportunity to guarantee all the learners covered similar material and gauge the success of the learning done during the problem. Also, this debriefing

encourages personal reflection among the learners and should be reinforced with peer and self assessment. Finally, all assessments and examinations of the learners should be done to measure the learners' progress towards the goals of PBL. Learners need to be assessed regularly both on the dimensions of knowledge and on the learning process to assure that each learner is achieving the full benefit of the PBL process.

While PBL has a relatively long history of successful use in medical and pre-professional schools, it has yet to be widely adopted by K–12 teachers (6). This may be due, in part, to the numerous challenges teachers experience when implementing PBL (6). The first challenge that new tutors and learners will face with PBL is learning how to create a culture of collaboration and interdependence. As outlined above, collaboration is a central part of the PBL process. However, for a collaborative environment, certain structures need to be in place, such as positive interdependence and individual accountability. One strategy teachers have used to help students adjust to the collaborative nature of PBL is the use of "posthole" units (7). Postholes are smaller, or "mini", PBL problems that allow learners to test out the PBL system on a much smaller scale. The use of postholes allows learners and tutors to get acquainted with how the

PBL system works and how to learn inside it before being fully committed to a much larger project.

The next challenge facing PBL in K-12 schools is adjusting to the change in roles, which is a product of the collaborative classroom environment. In the collaborative classroom environment, teachers fulfill a more tutor-like role, instead of the more traditional directive role. One proposal of how to overcome this is the suggestion of "rituals". Rituals are classroom scripts for specific activities that help teachers and students know that practices are appropriate at different times in the project sequence (8). Having automatic or routine practices in the PBL process can help teachers and students feel comfortable and in control.

Another challenge is how to scaffold the learning process and expectations of performance. Scaffolds represent one means of supporting learners in complex or unfamiliar environments (6). Instructional scaffolds refer to the tools, strategies, or guides that enable learners to reach higher levels of understanding and performance than would be possible without them (9). With the intricacies of starting and maintaining PBL in the K-12 environment, teachers and students that are new to their roles in PBL can benefit from many of these tools that structure the necessary PBL tasks while increasing their ability to complete the tasks independently.

TECHNOLOGICAL SOLUTIONS Chapter 3

POSSIBLE SOLUTIONS Section 3.1

In order to gain a wider acceptance in K-12 classrooms, PBL might need to take a technological approach. But with technology, just like education, there are many possible solutions to consider in solving a problem. The solution provided must promote PBL learning, allow for a modular learning process, present an easy to use interface for the K-12 demographic, be affordable, enhance enjoyment of learning and tutoring, and allow for data acquisition for the purposes of assessment. Many technological solutions are available to make PBL more accessible to the K-12 classroom; however, the following three were considered for this project. The benefits and drawbacks of each solution will be discussed in greater detail before the reasons are given for the selected solution. Three alternatives are considered for this paper: 1) have a pen and paper approach to PBL with static content and allow teachers to enter data into a database for evaluation; 2) have a website that both learners and tutors can use for data entry, collection, and assessment; and 3) use a Nintendo DS to have learners interact with the case and save their data through the built-in Nintendo DS Wi-Fi.

In a pen and paper solution, tutors would distribute information to learners when they achieve certain benchmarks or milestones in the problem. Many institutions, including University of Missouri's own School of Medicine, have used this method since the introduction of PBL. This method would be an easy way for learners and tutors to adapt to a PBL system. The differences in PBL and traditional learning are large and would require time and effort from both tutors and learners to adjust. Also, this solution would be the most cost effective in that it uses similar resources that are in classrooms now. Tutors would print out informational sheets to help knowledge discovery and probably worksheets to further facility the teaching and learning process as well as for assessment purposes. Teachers could possibly find these informational and work sheets from either a K-12 PBL website or through a package or library of content that the school purchases. Finally, the teacher would input student data into a database for tracking and assessment purposes.

While this is probably the most cost effective solution, it is also the least technological and also possibly the most work intensive, especially for the tutor. In this solution the tutor would have to find and prepare projects either by hand or from the provided website or database. If a website or database is used, the school would most likely have to purchase some type of license to use this software.

Depending on the cost of the license, this could drastically cut the cost effectiveness and make it just as expensive as other solutions that provide a more hands-on approach or a more technological or multimedia rich approach. Between finding problems and entering data after the problem, this solution is very time intensive for the tutor and requires a lot of time outside of the classroom, which is less than ideal.

Another solution would be to use an interactive website built around PBL. This website would allow learners and tutors to login to the system to do all learning and data entry about the problem. The tutor would release new "assets" to the learners when the tutor believes the group or class to be ready. These assets would be new information that could take any form including text, video, images, audio, or interactive applications. The learners would view these assets and then develop personal and group notes, hypotheses, and learning objectives. A website infrastructure of note taking and asset releasing could be easily specialized towards different types of learning by adding modules that apply themselves to specific facets of a specialized learning environment.

Since all of this work is done on a website and saved in a database, building tools to do assessment would be facilitated because the infrastructure for collecting all the data is already in place. A tutor

could browse a catalog of problems, like in the pen and paper solution; however, these problems would then be used online on the website. The website package could also allow for authoring tools that allow tutors to make their own problems for their class. However, building an interactive website like this would be a rather extensive and expensive undertaking.

While a solution like this will not be the most cost effective for a K-12 school, it would probably yield the best results, but at the same time it may have the highest learning curve for all participants. Learners could interact with each other through the Internet in and outside the PBL package, store all of their data, and refer to it later for studying purposes. The freedom of tutors to share what they want and when they want is still included in this solution, much like in the pen and paper solution. However, these assets are much more rich and do not have to be statically given. The assets could be video, images, audio, or even of an interactive nature to help facilitate the idea of learning discovery behind PBL. The ability for tutors to easily make interactive and rich problems with some assets they gather or create is very powerful. However, with all of this power comes a very high cost. The development cycle for a project like this is very long and requires a large, dedicated team to complete. This is not something a school could easily do individually. It would require

buying a lot of software, computers, and possibly even servers on which to run the solution. The added costs of a desktop or laptop computer for every student is very large. When many schools just struggle financially, this added cost could not be justified to try an experimental way of teaching. In addition to the financial costs, there would be a lot of time spent by tutors and learners learning the system instead of learning their curriculum.

The final solution discussed in this paper is to do a portable version of the website model outlined above but on a Nintendo DS. In this solution, learners would interact with assets on the Nintendo DS (NDS) by "playing" an interactive PBL learning "game." These "games," or interactive modules, would be selected by the tutor from a list of available problems provided by the developer. During the interactive module the learner would select a video game character, or an avatar, to represent them in the digital environment. Then the learner could do a variety of activities, such as answer questions in an interactive quiz, travel to faraway lands to explore, watch videos, listen to audio, look at pictures, type in comments or discussion on topics, and more. All of the learners' interactions will be saved to a database for evaluation later by a teacher. Anything that cannot be done on the NDS itself can be done on a single computer (or computer linked to a projector) to show the entire class either through assets

the teacher has found through a website that is part of the development of the NDS package.

The cost of this solution is less than the interactive website solution with much of the same functionality. The interactive website would require every learner to have at least a moderately powerful computer (laptop or desktop) for them to surf the web, watch videos (possibly in High-Definition), etc. The NDS solution requires each student to have a NDS; however the cost of an NDS is about a fifth to a tenth as much as that of a solid computer. All of the data that the learners enter would be stored in a database much like in the previous two solutions for assessment purposes; this would not be very different than other solutions. For assessment purposes, either website interface tools could be made to allow the tutor to interact with the learner's data or data could be handled manually from the database. Finally, the NDS would be an environment that most K-12 learners would be accustomed to. Many learners in the K-12 demographic play video games regularly and may, in fact, own NDSes of their own. This would greatly decrease the learning curve of the system for the majority of the K-12 learners.

To help facilitate the acceptance of PBL in K-12 schools, three possible technological solutions were discussed: enhancing pen and paper with a database, using an interactive website, or developing

interactive modules for the NDS. PBL seems to be a much more interactive and dynamic type of teaching and learning environment. Because of this, a more technological medium would help to facilitate the interactivity with the data and dynamic possibilities of the content. Receiving static input that does not change based on the user's actions is less than ideal. Whereas the pen and paper case would be helpful in getting acquainted with PBL during the early stages of use in the "mini" PBL, it is less than ideal for a full solution. The PBL website would fulfill the interactive and dynamic requirements of what is needed by K-12 schools. However, the sheer cost of buying a website package for use and all of the computers that every student would need are staggering. In contrast, the NDS is rather economical compared to the price of desktops and laptops. With the NDS system, only one computer would be needed in each classroom in order to demonstrate the few extra things that the NDS cannot do. Also, many children today are well acquainted with websites and video games so both the website or the NDS solution would benefit from this. Unfortunately, the intricacies of the NDS make it difficult to develop programs, so creating tools for tutors to make their own problems is probably not possible. However, as demonstrated by the video game market, a small development company could produce a wide catalog of problems to benefit K-12 PBL.

The Nintendo DS is not the only portable gaming system that could be used to create a PBL learning environment for K-12 learners. Three systems were considered: Nintendo Game Boy Advance (GBA), Nintendo DS (NDS), and the PlayStation Portable (PSP). Each of these systems has its own strengths and weaknesses in features, hardware, and development support, which will be explored in this section. The features that are most important for a project like this are Wi-Fi capabilities, easy text entry, and a large development community.

In order for users to save their data to the database, it is critical that the portable device used utilizes some type of Internet connection. None of the three devices mentioned comes with an Ethernet connection, but two have Wi-Fi capabilities, the NDS and the PSP. The NDS has built-in support for IEEE 802.11b/g compatible Wireless Network Connections with WEP encryption only. The PSP has only IEEE 802.11b compatible Wireless Network Connections with no encryption support. Most Wi-Fi hot spots or school networks have some type of encryption to maintain a secure network, so encryption capabilities are critical. Thus, the NDS's built-in support of Wi-Fi b and g as well as basic WEP encryption support makes it the only viable choice.

PBL requires significant amounts of note taking and free thinking, so a portable device that runs a PBL system requires some type of text entry system. Although none of these devices has keyboards or keyboard peripherals, the NDS has a touch screen. The NDS touch screen uses a stylus for input and can do fairly accurate pixel detection of input to do fine commands. With this touch screen a user can type on a graphical keyboard, draw images, make highlights on assets, move objects around, and even imitate a Graffiti-like letter recognition system that has been popular on PDAs for over 10 years. The NDS also has a microphone that can be used for input and commands.

Hobby development for video game consoles, known as "homebrew," is growing in popularity across the Internet. Homebrewing is legal because it is simply running code that a developer creates on legally purchased hardware. Since there are no development tools for video game systems on a student or personal scale (the only ones that exist are commercially available for tens of thousands of dollars), a good, legal homebrew community is important for learning how to program for the hardware of the device. Programming for video game consoles is not like just opening up any compiler, typing in the language of the developer's choice, and clicking on the build button. Programming for video game consoles requires a

very intricate knowledge of the hardware of the device and its specification. Unless the developer plans on reverse engineering the entire piece of hardware and figuring this out on his or her own, a large online community is necessary to help map out the hardware and figure out ways to work with the hardware.

The GBA was one of the first video game consoles to have a massive homebrew development online. The homebrew community quickly unlocked all of the features of the GBA and mapped all of its registers and their uses. When the NDS came out, the homebrew community surrounding the GBA quickly gravitated towards it and started the process anew. Since the NDS is basically the same hardware infrastructure (doubled the register RAM for the two screens, increased the processor power, and added support for touch screen, Wi-Fi, and microphone) this was not a difficult task for the basics of the NDS system.

During the GBA homebrew cycle, numerous isolated sites developed independently and produced many standards. With the NDS, however, a few of the people that were important in the GBA era collaborated and developed "libnds". Libnds is a standardized low-level library to deal with registers and some basic functions to allow everyone an entry level into NDS homebrew development. Many sites still exist and offer individual tutorials covering many different topics of

NDS development; however, the vast majority of them use libnds as a basis of their work. Furthermore, there are many sites that make libraries that go further to create much more higher-level functions that allow the developer to easily create keyboards and emulate the Graffiti text system. Eventually, libraries were even made to use the NDS's Wi-Fi capabilities, which was the last uncharted part of the NDS hardware. Unfortunately for PSP homebrew, Sony is constantly trying to put a stop to it because they fear the repercussions that could occur due to piracy. Because of this, homebrew development on the PSP is spotty and inconsistent among systems and usually requires a downgrade of the system firmware to do any work.

Obviously the NDS is the clear winner in the criteria of Wi-Fi access, massive textual input, and existence of large homebrew community. While the GBA and the PSP have some advantages in other fields (e.g., the GBA is cheaper and the PSP has a stronger graphical system), neither of them can best the NDS in what would be considered the most important aspects of development for this project. Also, with the GBA and the NDS being very similar, I can use a lot of my knowledge I gained working with the GBA on the NDS only with increased features. Not only will this paper delve deeper into the intricacies of the NDS, its development, and the work done specifically on this project in future sections, but, to a lesser extent, it will also

cover the comparisons and contrasts between the GBA and the NDS hardware.

TECHNICAL SPECIFICATIONS Section 3.3

The forgoing discussion of PBL is not an unambiguous specification of a system implementation, but it does permit the requirements to be understood in a general way. Some facets of the proposed system may not be necessary on some cases or developed at all depending on the needs of the users. First and most importantly, is what will be developed on the NDS. Then how any extra information that cannot be displayed on the NDS will be delivered to the users will be defined. Finally, tools will need to be made for the tutors in order to select or build cases and evaluate learners' progress will need to be specified.

The software on the NDS will need to be defined the most extensively as it is the keystone of how this software package will function. First, cases should allow the selection or building of personal avatars for learners. This is a small task that will just require extensive artwork, but it will allow younger learners to become more invested in the learning process by creating avatars that they can identify with. Of course the avatars will run the gamut of every sex, gender, and body type to allow children to better identify with the character they select to represent them in the video game world.

Allowing the user to imagine themselves in the game world should allow for learners, especially younger ones, to become more invested in the learning process and provide better results. Avatars will not be the only graphics necessary in the NDS cases; sprites of varying characters for the game will need to be made, along with tilesets of different locales (urban, rural, indoors, outdoors, different countries, etc) will need to be made.

Next, the cases on the NDS will need to provide many different forms of input to the user. Some mandatory inputs will be the use of the controller buttons on the NDS unit and simple touch screen functions such as selection and "drag and drop". While many other inputs could be implemented through the touch screen, these are the bare minimum for a functioning system. Further inputs may be wanted, but are not necessarily mandatory, such as keyboard and Graffiti input through the touch screen with a stylus. These two types of input will allow a much broader range of data collection from the learners and allow much more possibilities for learning and evaluation. A keyboard or Graffiti system would even allow for the development of a communication system between users. However, the communication system would most likely be rather slow and could be better implemented through an optional website system so this is left up to the discretion of the developer. Even more inputs may be

capable through the combinations of touch screen and controller use; however any further input capabilities are left up to the discretion of the developer.

Furthermore, a simple mechanic of a quiz system will need to be developed to ask learners multiple choice questions and then record their answers for use later. Developers of the NDS game will require input of cases and assets from experts in the appropriate fields that are building cases for particular subjects. Using these cases and assets developers will have to build different activities to perform and puzzles to solve to help facilitate the learning process. Sound and music development could be done, but is left up to the discretion of the developer if it is necessary. Finally, Wi-Fi interactivity will need to be created between the NDS game and a website to store data in a database.

There will be some assets that cannot be shown on the NDS like high-definition video, images, or some types of audio. If these assets are important enough to the progression of the case, they must be displayed to the users in some way. A delivery system over the Internet might be ideal, assuming that every classroom has at least one computer with Internet access. The class could crowd around the one computer or the computer could be connected to a television or projector. The decision of how to display these assets could be left up

to the school or the developers. However, to deliver these assets, a website will need to be made to distribute them properly. This website would not need to be incredibly sophisticated, it just needs to allow log in, selection of assets, and tools to either download or display the assets.

Along with an asset website, some type of assessment website would need to be made. The assessment website could be bundled with the asset website if desired. This website would allow tutors to log in and evaluate the work of all the learners. It would need to show the progress of the learners through the case and their answers to all questions and activities that are run on the NDS. Furthermore, the tutor would be shown if the learners answered correctly to multiple choice questions and possible answers for more open ended questions. The tutor could then see statistics and rankings of progress of each individual learner and suggestions on what areas the learners could improve and suggestions on how to help them improve. An evaluation website has limitless options and is completely up to the discretion of the developer. However, at the very least, a rudimentary site that shows learners progress and answers is required to evaluate the job of the learners during a case.

PROTOTYPE SPECIFICATIONS Section 3.4

For the sake of this thesis, a prototype will be constructed that incorporates all of the major features mentioned above in order to prove this concept is feasible. All of the websites mentioned above for asset delivery and assessment are easily built and not necessary for a proof of concept, so for the sake of this thesis, they are ignored in development. The most important part of this concept is the NDS game and the website that allows storage through the NDS over Wi-Fi. All of the graphics (sprites and background tilesets) will be borrowed from the Super Nintendo game Chrono Trigger (copyright Square-Enix Company). This means that a wide range of graphics are not necessary for this proof of concept as they are easily made by artists on a development team that would build a final product. The proof of concept prototype will include all types of input mentioned such as controller input and touch screen inputs that include basic touch screen selection, keyboards, and a Graffiti text recognition system. This prototype will not include any type of messaging system for users of the system; this can easily be built for a website or proven through the use of saving to a database through the NDS. The quiz system will be fully developed to allow users to answer multiple choice questions. Besides a basic quiz, no other activities or puzzles are planned for this prototype. A combination of examples from other

video games and what is built into this prototype for other functions is sufficient to provide proof of this capability. No sound or music development will be done in this prototype as it is some what out of the range of what is necessary to prove the core of this idea. Finally, Wi-Fi interactivity is crucial to this project working and this will be built into the prototype to test communication between the NDS and a website to store data into a database.

NINTENDO DS DEVELOPMENT Chapter 4

NINTENDO DS HARDWARE Section 4.1

The NDS's hardware is rather powerful for its small size. The standard NDS measures 148.7 x 84.7 x 28.9 mm (5.85 x 3.33 x 1.13 in) in size and the newer NDS Lite measures 133 x 73.9 x 21.5 mm (5.24 x 2.9 x 0.85 in) which is about 42% less volume than the original NDS. Both versions of the NDS have the same internal and external hardware, but one is smaller than the other. Even with its small size, the NDS has enough processing power from its dual processors to create graphics on par with the Nintendo 64. The Nintendo 64, which was released in 1996, is considered to be only two video game generations older and was a larger, non-portable home system. The NDS runs on two ARM processors, an ARM946E-S main CPU and an ARM7TDMI co-processor, that run at clock speeds of 67 MHz and 33 MHz respectively (14). This is a significant hardware improvement from the GBA's (predecessor to the NDS) single ARM7TDI that clocked at 16.8 MHz (11). Each one of the NDS's processors can devote an entire 2D engine to one of the NDS's two screens. These 2D engines work very similarly to the 2D engine on the GBA, but the NDS's are more powerful and there are two of them.

This allows the NDS to have one of the most advanced 2D rendering systems ever seen on a console system.

Both of the NDS's screens measure 62.5 x 47.0 mm (2.45 x 1.81 in), 76.2 mm (3 in) diagonal, run at a resolution of 256 x 192 pixels, and have a dot pitch of .24 mm. These screens are about 22 mm (0.87 in) apart or 90 imaginary pixels away from each other. The lower screen on the NDS is overlaid with a resistive touch screen, which registers pressure from one point on the screen at a time. This screen can also average multiple points of contact, if necessary.

As stated before, each screen can have one whole 2D engine dedicated to it. However, the NDS is only capable of one 3D engine that can only render on one screen at a time. The NDS has a transform and lighting chip as part of its 3D hardware. This allows for better graphics while alleviating some of the processing power from the core processor. The NDS's 3D hardware is also capable of texture-coordinate transformation, texture mapping, alpha blending, anti-aliasing, cell shading, and z-buffering. However, it uses point (nearest neighbor) texture filtering, which results in some tiles having a blocky or pixelized appearance. Unlike most 3D hardware however, the NDS has a set limit on the number of triangles it can render as part of a single scene. The maximum amount of vertices is about 6144, or about 2048 triangles per frame.

All of this hardware stores its data locally on 4 MB of RAM. While many files and instructions can be stored on a game card, these 4 MB are what the processors use directly for current processing. In addition to this 4 MB of RAM, there are 656 KB of Video RAM (VRAM) and additional RAM for BIOS, fast RAM (IWRAM), fast shared RAM (WRAM), Virtual Video RAM, and a few other odds and ends. Refer to Figure 4.1 for a graphical representation of this.

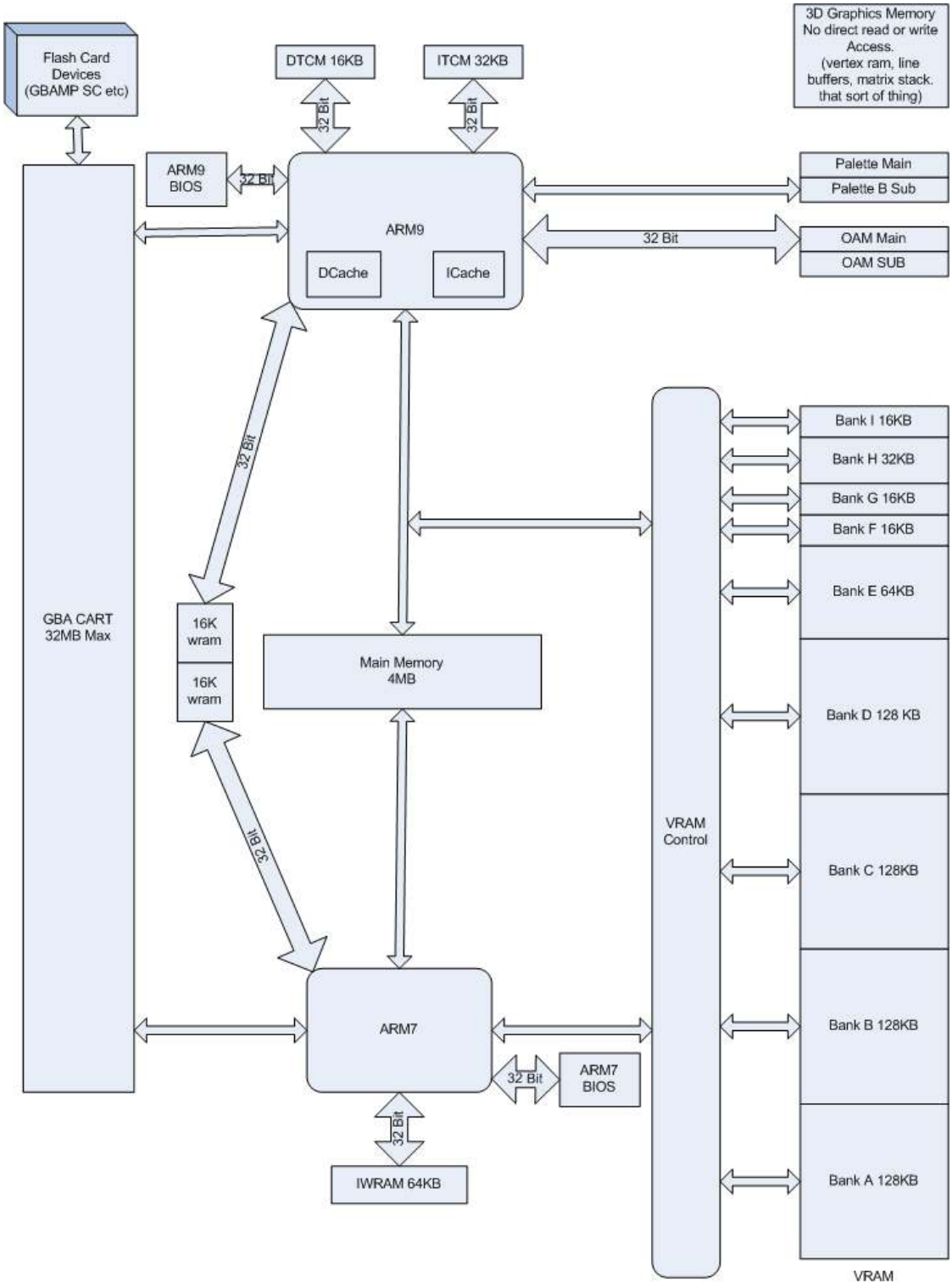

Figure 4.1: Nintendo DS Memory Map (20). Unless otherwise stated, the data width for each bus is 16 Bit.

Memory Map

ARM 9			
Name	**Start Address**	**Stop Address**	**Size**
Main	0x02000000	0x023FFFFF	4MB
BIOS	0xFFFF0000	0xFFFF7FFF	32KB
ITCM	0x00000000	0x00007FFF	32KB
DTCM	0x0B000000	0x0B003FFF	16KB
Shared WRAM Bank 0	0x03000000	0x03003FFF	16KB
Shared WRAM Bank 1	0x03004000	0x03007FFF	16KB
ARM 7			
Main	0x02000000	0x023FFFFF	4MB
BIOS	0x00000000	0x00003FFF	16KB
IWRAM	0x03800000	0x0380FFFF	64KB
Shared WRAM Bank 0	0x03000000	0x03003FFF	16KB
Shared WRAM Bank 1	0x03004000	0x03007FFF	16KB
Video RAM			
Main OAM	0x07000000	0x070003FF	1KB
Sub OAM	0x07000400	0x070007FF	1KB
Main Palette	0x05000000	0x050003FF	1KB
Sub Palette	0x05000400	0x050007FF	1KB
Bank A	0x06800000	0x0681FFFF	128KB
Bank B	0x06820000	0x0683FFFF	128KB
Bank C	0x06840000	0x0685FFFF	128KB
Bank D	0x06860000	0x0687FFFF	128KB
Bank E	0x06880000	0x0688FFFF	64KB
Bank F	0x06890000	0x06983FFF	16KB
Bank G	0x06894000	0x06897FFF	16KB
Bank H	0x06898000	0x0689FFFF	32KB
Bank I	0x068A0000	0x068A3FFF	16KB
Virtual Video RAM			
Main Background	0x06000000	0x0607FFFF	512KB
Sub Background	0x06200000	0x0621FFFF	128KB
Main Sprite	0x06400000	0x0643FFFF	256KB
Sub Sprite	0x06600000	0x0661FFFF	128KB

Table 4.1: Nintendo DS Memory Addresses

Table 4.1 lists all parts of RAM and include addresses and size of each part. Some of the parts of RAM are self explanatory, like BIOS; however, some could use a little more explanation. ITCM, or Instructions Tightly Coupled Memory, is a 32 KB section of the ARM9 memory that is faster than the main RAM. ITCM is ideal for small, preferably 32-bit, functions that are either computation-intensive or frequently called. For example, libnds uses ITCM to store its interrupt dispatcher. DTCM, or data tightly coupled memory, is another section of the ARM9 memory that is used for fast data memory retrieval. Since DTCM is only 16 KB, it is mostly used as a location for the programs stack and for local variables. Because of the small size of DTCM it is advisable not to use too many local variables (especially arrays) or deep recursion. Too much of either of these things could cause overflow problems very quickly. Both ITCM and DTCM use 32 byte cache lines thanks to the Direct Memory Access (DMA) buses and are directly contained in the ARM9 CPU core.

There are also two small 16 KB banks of fast, 32 bit RAM that both the ARM7 and ARM9 can use. These are referred to as the Shared Windows RAM Banks or WRAM. WRAM has nothing to do with the Windows Operating System and is, in fact, a faster version of Video RAM or VRAM. These blocks of RAM can only be accessed by one processor at a time. If both processors try to access this RAM, the

ARM7 has priority. By default, these two banks will be mapped to the ARM7 as they form a continuous chunk of RAM with ARM7's IWRAM, or Internal Working RAM. IWRAM is the ARM7's personal 64 KB of fast, 32 bit RAM. The combination of the two Shared WRAM banks at 16 KB a piece and the 64 KB of IWRAM effectively gives the ARM7 96 KB of fast, 32 bit RAM.

A rather large, and important, section of memory is the Object Attribute Memory, or OAM. The OAM stores all of the data for how every graphical object, or sprite, is rendered to the screen. This data includes pixel location on screen, the sprite location on the current map, and other attributes of the sprite like height, shape, and palette type. The area of memory following this is reserved for the Palettes. This stores all of the color information currently in use by the Video RAM for backgrounds and sprites. There is a Main and Sub version of both OAM and Palette, to indicate to which 2D engine the OAM and Palette belong.

Next are the 9 different VRAM banks of varying sizes. These banks, when mapped correctly, will contain any type of tilesets and maps for the requested amount of backgrounds and sprites. For the engine to know what VRAM bank to use, it must be assigned to the Virtual Video RAM registers. The Virtual Video RAM is a map of what backgrounds and sprites need to be displayed on both of the screens.

Finally the 256 KB of Serial Flash Memory is used for system settings, BIOS, and update-able Firmware.

Finally, attention must be paid to the input devices for the NDS. As stated before, the lower screen on the NDS has a touch screen that allows input from one point on the screen at a time. This input can be administered through touch or a stylus. The screen can even average multiple points of contact to gather one touch point from many. This touch screen can be used for many applications including a "point and click" interface, a virtual keyboard, emulating a Graffiti system, and even dragging and dropping. The NDS also has a directional pad, two shoulder buttons, and four face buttons for many combinations of input. Some games even utilize multiple styles of input to allow users to pick a favorite. Not only are there controls on the NDS for input, but there is a standard microphone too. This microphone can be used to control objects through just noise or voice recognition commands. Finally the NDS is Wi-Fi enabled to connect to any Wi-Fi b or g hot spot that has up to, and including, WEP encryption security. Wi-Fi on the NDS allows for wireless multi-player games through either a LAN or through the Internet. Also, it can be used for limited web surfing and HTTP posting in order to save to a database.

The strength of the ARM hardware for the present application derives from the provision of hardware elements which can carry out

key functions such as graphics, direct memory access, sprites, touch screen input, and Wi-Fi. Conversely, there is no hardware for many operations common on general purpose computers, including even multiplication and division. This hardware concept provides the speed and memory efficiency necessary for a powerful portable device; however it does require the programmer to directly control these hardware units through their associated registers.

LIBNDS Section 4.2

In order to program to the hardware, a library of registers and a compile toolchain are necessary. Libnds and devkitPro are the library and toolchain of choice for NDS programming. Libnds, formerly ndslib, is a library of register definitions and some low level functions that was created by Michael Noland (joat) and Jason Rogers (dovoto) (12). Libnds was developed to be an open source alternative to Nintendo's official SDK for the NDS. Noland and Rogers were both heavily involved in the homebrew scene for the GBA. Both Noland and Rogers reverse engineered the GBA very early on to discover all of its registers and memory map. They also created many of the first programs and tutorials for development of homebrew on the GBA. When the NDS came out, they decided to try to make an all encompassing library of definitions so there was a standard that everyone could work with.

In the days of GBA homebrew development, sources of register and memory definitions were scarce and it seemed everyone created their own by scratch and had different naming conventions. Understandably, this led to many problems when sharing code with other developers and hindered a lot of possible progress on GBA homebrew development. But with the advent of libnds, Noland and Rogers were able to provide a common library that everyone could build from and use as a point of discussion. Not only that, but their register and memory location naming convention in libnds was much more logical and orderly than the majority of naming conventions in GBA definitions that floated around the Internet. Currently libnds is maintained and updated by Dave Murphy (WinterMute) through the devkitPro project.

DevkitPro is as the developers put it a "toolchain of choice for homebrew game development." DevkitPro currently supports development for the GBA, GP32, PSP, NDS, and GameCube. DevkitPro is an all-in-one tool that features many libraries for homebrew development like libnds, tools to help developers make data for their project (i.e. image and audio converters), and provides a development environment. Even though devkitPro helps configure a computer for development, it is not an Integrated Development Environment, or IDE. DevkitPro installs path variables onto an OS for the development

packages selected, ties the packages into an open source compiler (gcc and g++), and allows developers to build projects using this compiler and make files. All the things that devkitPro does could be done manually, and all the programs it installs could be installed separately. However, devkitPro makes it extremely simple to set up a homebrew development environment and allows a developer to start programming sooner and easier. DevkitPro also allows the developer to select their personal favorite IDE since it is compatible with Microsoft Visual Studio, UltraEdit, Programmer's Notepad, and even Eclipse.

Though libnds is essential to any NDS homebrew development, it is not the only user built, open source library out there; the most popular of these libraries is PALib. PALib is a very large open source community project to build basic functions that have a lot of utility for NDS development. The wiki, tutorial, functions, and documentations have been translated into at least 7 different languages. While PALib does not come included with devkitPro, it can be installed to integrate with devkitPro very easily. Another smaller and more specialized library is dswifi. Dswifi is a library of tools that allows the use of the NDS's built-in Wi-Fi. This was one of the last features of the NDS to be unlocked by the homebrew community. It was locked for so long that some websites started offering “bounties” for a programmer who

could document all of the Wi-Fi registers and the first developer to make homebrew code that would actually send and receive from websites. This bounty went unclaimed for over a year until Steve Stair created the dswifi library and released it to the public. Since then, dswifi has been modified and built upon to be added into devkitPro and PALib. It is not difficult to use libnds, dswifi, and PALib in conjunction with each other; in fact most projects like this one use a little bit of all of them. Through the course of this project the various functions that are used from Libnds and PALib will be recognized while explaining how coding for the NDS works and how this project in particular works.

BACKGROUNDS Section 4.3

The NDS has a very robust and full featured 2D engine in it (two 2D engines actually). Some of these features include frame buffer backgrounds, rotational backgrounds, tiled backgrounds, sprites, rotational sprites, transparencies, importing of graphics through Direct Memory Access (or DMA), graphic effects (like windowing, blending, and mosaic effects), text output, and many more. In addition, the NDS is capable of sound, Wi-Fi, and user input through controllers, touch, and microphone.

Some background knowledge of how the NDS renders to the screen is necessary before discussion of how to render out any graphics to the screen can begin. Each of the LCD screens on the NDS functions in the same way that any other screen does. The hardware draws the picture on the screen from left to right and from top to bottom. Both NDS screens are entirely refreshed 59.8 times every second or 60 frames per second. However, it is not nearly as simple as that. Every row of pixels is called a scanline, and while a scanline is being drawn the hardware is in the HDraw period. The HDraw period lasts for 256 pixels on every scanline. After a scanline is drawn there is a brief pause before the next scanline can be drawn. This brief pause in between scanlines is called the HBlank period and lasts for 91 pixels. In between the HDraw and HBlank period there is a brief 8 pixel buffer where the NDS's registers read as if the hardware is still in HDraw, but there are no pixels visible on the screen for this time.

There is a scanline for every pixel high the screen is; in the case of the NDS, there are 192 scanlines and the drawing of all of these scanlines makes up the VDraw period. At the end of VDraw there is a brief pause before the screen starts drawing again at the top, left corner while it continues to draw 70 scanline blanks; this is called the VBlank period. In between VDraw and VBlank is another buffer of 1

scanline where the NDS's registers read as if the hardware is still in VDraw, although there is not a visible scanline on the screen at this time. VBlank is longer than HBlank because many HBlanks occur during the VBlank as shown in Figure and Table 4.2. To avoid graphical glitches like tearing, positional data and graphical updates are usually made during the VBlank period. As a result, most GBA and NDS games run at 60 or 30 fps because they are synched to the VBlank period and not to any type of processor cycle clock.

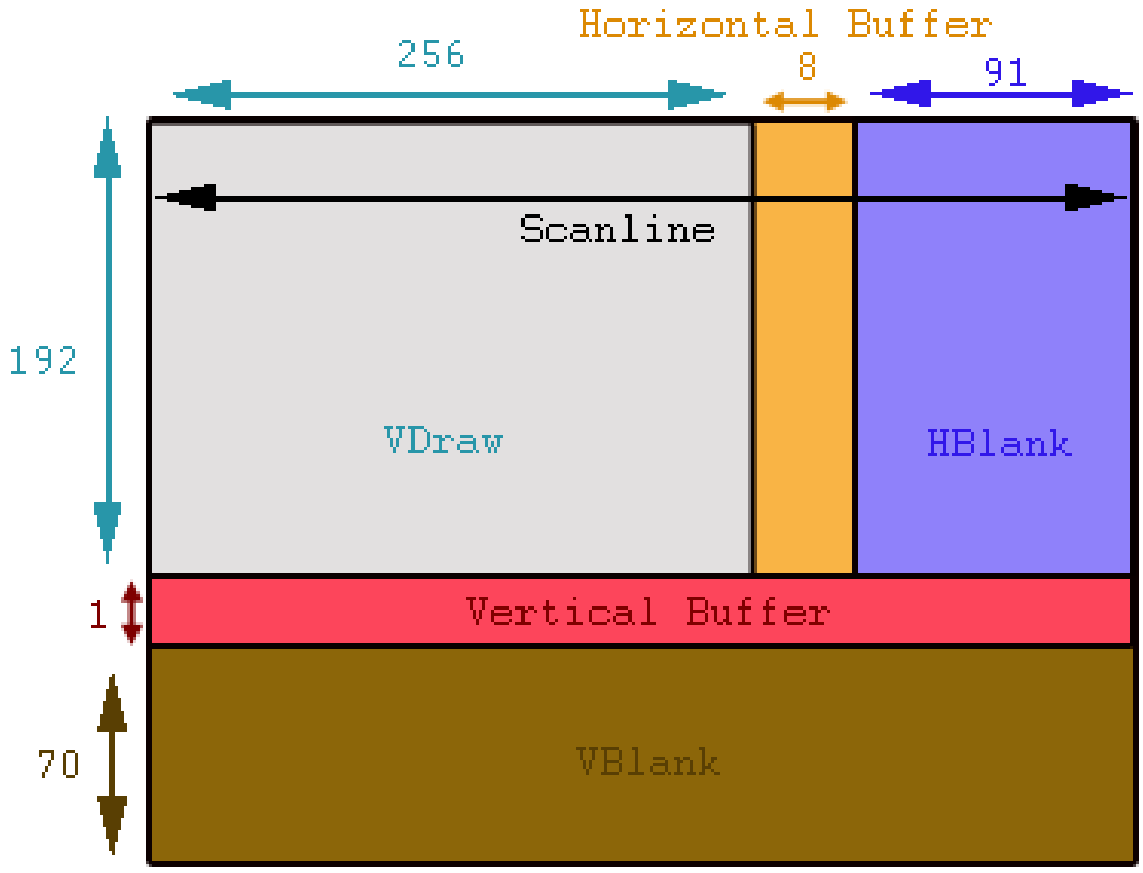

Figure 4.2: A diagram showing the ranges of VDraw, VBlank, and HBlank on the NDS screens.

Subject	Length	Cycles
Pixel	1	6
HDraw	256px	1536
HBuffer	8px	48
HBlank	91px	546
Scanline	HDraw + HBuffer + HBlank	2130
VDraw	192 * Scanline	408960
VBuffer	1 * Scanline	2130
VBlank	70 * Scanline	149100
Refresh	VDraw + VBuffer + VBlank	560190

Table 4.2: Display timing details

GRAPHIC MODES Section 4.5

Backgrounds are a basic building block for video games. The majority of video games put the player in control of some type of character (usually represented by a sprite) and the player moves this character around on some type of background. These backgrounds come in many types and sizes and display differently depending on the render mode the NDS's hardware is in. As mentioned previously, there are two separate 2D graphic cores in the NDS. These two cores are commonly referred to as the Main and Sub graphics cores. Only one 2D graphics core can be displayed on each screen at a time.

When set to the same mode, each core will act the same and have similar features. However, there are some minor differences between the cores. These differences are:

- The Main core has two extra modes which the Sub core does not have. These modes are capable of rendering large bitmaps.
- The Main core can use one of its backgrounds in the 3D engine, which can only be displayed on the Main core's screen.
- The Main core can also choose not to use the 2D engine at all and render directly from Virtual Video RAM. Doing this is usually referred to as the “frame buffer mode.”

Main 2D Engine				
Mode	**BG0**	**BG1**	**BG2**	**BG3**
Mode 0	Text/3D	Text	Text	Text
Mode 1	Text/3D	Text	Text	Rotation
Mode 2	Text/3D	Text	Rotation	Rotation
Mode 3	Text/3D	Text	Text	Extended
Mode 4	Text/3D	Text	Rotation	Extended
Mode 5	Text/3D	Text	Extended	Extended
Mode 6	3D	-	Large Bitmap	-
Frame Buffer	Direct VRAM display as a bitmap			
Sub 2D Engine				
Mode	**BG0**	**BG1**	**BG2**	**BG3**
Mode 0	Text	Text	Text	Text
Mode 1	Text	Text	Text	Rotation
Mode 2	Text	Text	Rotation	Rotation
Mode 3	Text	Text	Text	Extended
Mode 4	Text	Text	Rotation	Extended
Mode 5	Text	Text	Extended	Extended

Table 4.3: This table shows how all of the different Graphics Modes work on the different cores.

EXTENDED ROTATION BACKGROUNDS Section 4.6

Extended rotation backgrounds are often referred to as the bitmap modes. Since there is a direct relationship between what is in the Virtual Video RAM and what is drawn on the screen, bitmap modes are a good example of how graphics work on the NDS. Figure 4.3a shows a bitmap representation of a sprite. A sprite is used here just for simplification purposes, but what happens on the bitmap background works the same way, just on a larger scale. A bitmap in the NDS is exactly like a bitmap on a computer; it is composed of a

grid of colored pixels. In order to use bitmaps in a program and correctly render them on the screen, the developer and the program need to know how they are arranged in memory. Figure 4.3b is a zoomed in copy of the sprite found in Figure 4.3a and demonstrates how these pixels are blocked off in a grid.

Since a bitmap is just a matrix of color-indices that can be described by ***w***x***h***, where ***w*** is the width of the bitmap and ***h*** is the height of the bitmap, each individual pixel can be referenced by a coordinate pair. In the Virtual Video RAM though, the lines of the bitmap are laid out in one long array so that the pixel (***x***, ***y***) in the bitmap ***w***x***h*** would be the ((***w*** * ***y***) + ***x***)-th address from the starting point designated for this entry in memory. Figure 4.3b demonstrates how this works by labeling the offset in memory with the little yellow numbers. This figure illustrates a ***w***=24 by ***h***=24 bitmap which is stored in RAM at 8bpp, or 8 bits per pixel which is equal to 1 byte. Since the y-axis in the NDS points down, the origin is at the top left corner. This entire math is dependent on an 8bbp; if a change is made to another bit depth, all of the addresses change, as well. For example, if another common bit depth of 16bpp is used, this would require 2 bytes per pixel, and all pixel numbers would need to be multiplied by two.

Text backgrounds are used in the homebrew community to describe what are also known as "tiled" backgrounds. Tiled backgrounds work much like bitmaps but on a larger scale. A tiled background requires a tileset that consists of multiple 8x8 pixel tiles. These tiles work in similar ways to large colors in a bitmap, but instead of painting a single pixel one color, it paints an 8x8 pixel tile onto the screen. These tiles are then used to create a map of what the screen will look like. This map works much like the bitmap in Figure 4.3c, in which the numbers represent which tiles get painted at which tile location on the screen. Because the size of the screen is reduced by a factor of 8, the NDS is only concerned with rendering 32 x 24 tiles on the screen at any time. Obviously this requires much less overhead in processing power when updating a background as opposed to updating 256 x 192 pixels. This is the reason that the vast majority of GBA and NDS games use tiled backgrounds instead of drawing the backgrounds by hand with bitmaps.

Figure 4.3a: A sprite composed of a 16x32 bitmap.

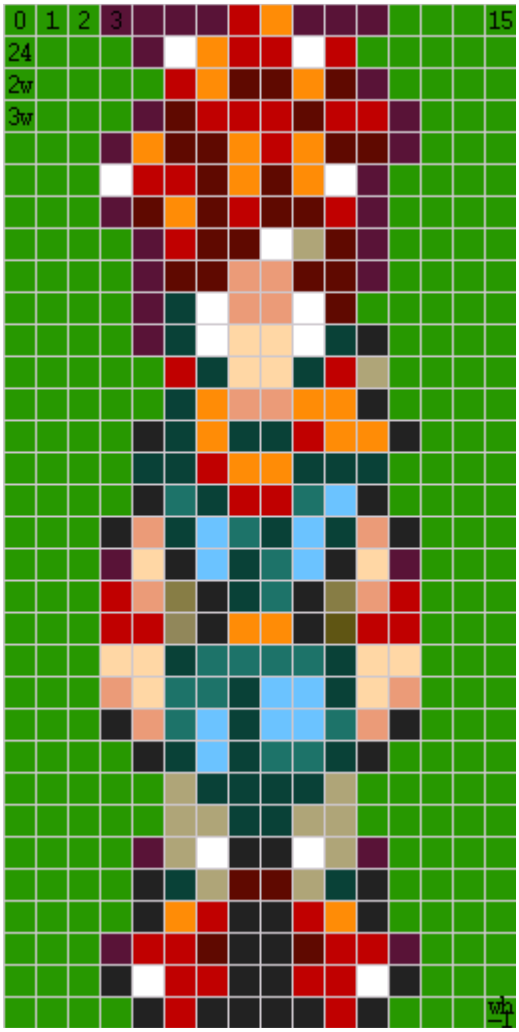

Figure 4.3b: Zoom in on Figure 4.3a, with pixel offsets.

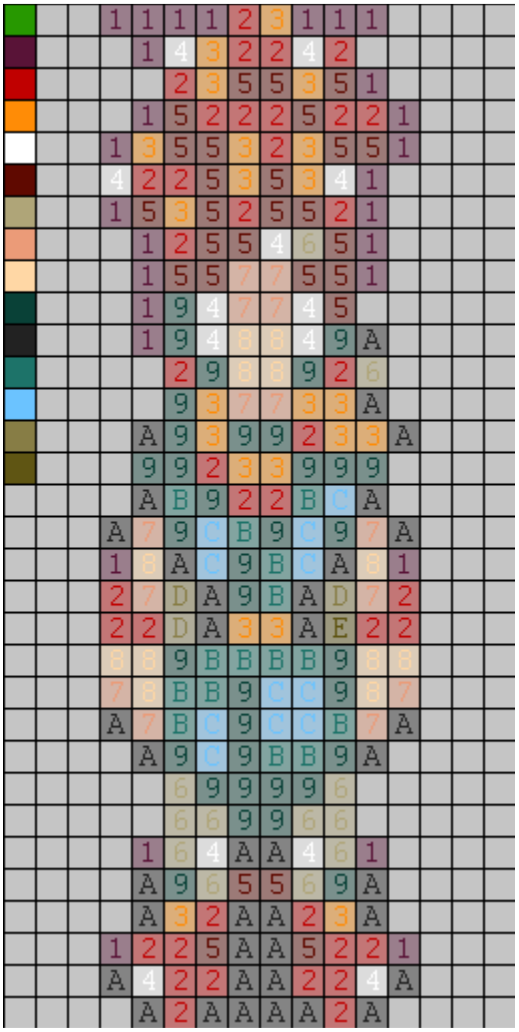

Figure 4.3c: Zoom in on Figure 4.3a, with pixel values. Zero omitted for clarity. Palette on the left-hand side.

For concrete, mathematical proof of this, consider Figure 4.4a, which illustrates a 256 x 256 pixel image; even at 8bpp this would require over 65 KB of VRAM. Recalling how our VRAM is divided (refer to Table 4.1), there is not a VRAM bank larger than 128 KB. Even if the largest VRAM bank was used to store backgrounds, there is only

room for two of the four backgrounds that can be used. Most normal game levels get up to 1000 x 1000 pixels, which is not at all practical. In addition to storing the screen in RAM, there is the need to be able to scroll around the map, which would mean updating all of the pixels, every frame. Even if the code required to scroll was optimized to its full potential, it would take too long to run almost 60 times per second. Closer inspection of the map would yield the observation that the majority of the map is composed of repeated shapes and elements.

Normally the NDS divides tiles into 8x8 pixel tiles; however for simplicity the map and tileset in Figures 4.4a-c are divided into 16x16 pixel tiles. On closer observation of Figure 4.4a, it can be noticed that the image is made up of these repeating 16x16 pixel tiles. These blocks would be the tiles of the map as represented in the tileset of Figure 4.4b. There are only a few unique tiles that compose the tileset. However, to render these tiles to the screen a tilemap is needed that works in much the same way a bitmap renders pixels to the screen. The image that is to be rendered on the screen is divided into a matrix of tiles. Each element of the matrix has a tile index which represents the tile that is to be displayed in that location. In Figure 4.4c such a tilemap has been overlaid upon the image to demonstrate how this works. Suppose that both the tileset and the

map used 8bit entries: the sizes of 11 * (16 * 16) = 2816 bytes for the tileset and 16 *16 = 256 bytes for the tilemap. That makes for a grand total of 3072 bytes for the whole scene instead of the 65 KB that were necessary for the bitmap version of the scene. Rotation backgrounds, also known as affine backgrounds, work exactly the same was as normal tiled backgrounds when rendering to the screen. However, they have the added benefit of the ability to be rotated and scaled according to an affine matrix.

The tile mapping process is demonstrated by using the tileset of Figure 4.4b and the tile map of Figure 4.4c; the end-result is Figure 4.4a.

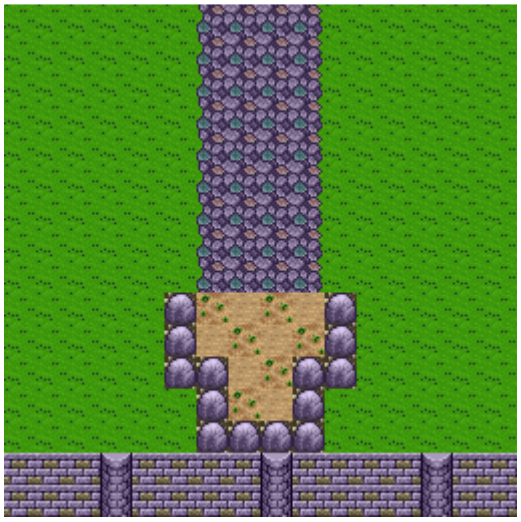

Figure 4.4a: The image on the screen.

Figure 4.4b: The tileset.

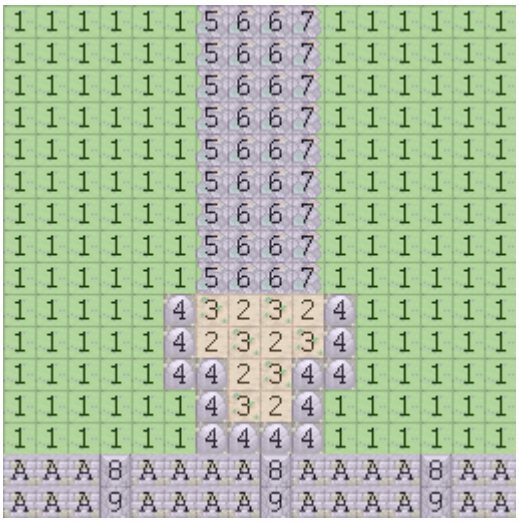

Figure 4.4c: The tile map (with the proper tiles as a backdrop).

DISPLAY CONTROL REGISTER Section 4.8

In order to put this knowledge to use, a developer needs to store bits into the NDS's memory register to control the hardware. The first register that a developer must control is the DISPLAY_CR register, which is the primary control of the Main core engine. A SUB_DISPLAY_CR controls the Sub core engine. The bit layout of these registers can be found in Register 4.1 and the uses of all of these bits can be found in Table 4.4. All representations of registers or register-like data in this paper will follow the general format where the libnds library uses the libnds name, the Sub name is the register name for the Sub core's version (if it exists), the Address and Sub Address is where the registers reside in memory, the Engine is which core or 2D engine it works on (M for Main and S for Sub), the Bit is

which bit in the register is being discussed, the Name is how the bits are represented in the register map, the definition is how it is defined by libnds, and the description explains what the bit(s) do.

Libnds name: DISPLAY_CR, **Address:** 0x04000000,
Sub name: SUB_DISPLAY_CR, **Sub Address:** 0x04001000

F	E	D	C	B	A	9	8	7	6	5	4	3	2 1 0
SW	W1	W0	SPR	BG3	BG2	BG1	BG0	FB	SBM	SBD	STM	3D	Mode

1F	1E	1D 1C 1B	1A 19 18	17	16	15 14	13 12	11 10
SE	BE	SB	CB	HB	SBB	STB	DMB	DM

Register 4.1: Register map for the Display control register.

Engine	Bit	Name	Definition	Description
M & S	0-2	Mode	MODE_x_2D	Allows for setting graphic modes 0-5.
M	3	3D	ENABLE_3D	Sets BG0 to 2D or 3D mode (0 = 2D; 1 = 3D)
M & S	4	STM	DISPLAY_SPR_1D DISPLAY_SPR_2D	Sprite Tile Mapping (0 = 2D, max 32 KB; 1 = 1D, max 32 KB - 256 KB)
M & S	5	SBD	DISPLAY_SPR_2D_BMP_128 DISPLAY_SPR_2D_BMP_256	Sprite Bitmap 2D-Dimension (0 = 128 x 512 pixels; 1 = 256 x 256 pixels)
M & S	6	SBM	DISPLAY_SPR_1D_BMP	Sprite Bitmap Mapping (0 = 2D, max 128 KB; 1 = 1D, max 128 KB - 256 KB)
M & S	7	FB	DISPLAY_SCREEN_OFF	Force a screen blank
M & S	8	BG0	DISPLAY_BG0_ACTIVE	Enables rendering of BG0
M & S	9	BG1	DISPLAY_BG1_ACTIVE	Enables rendering of BG1
M & S	A	BG2	DISPLAY_BG2_ACTIVE	Enables rendering of BG2
M & S	B	BG3	DISPLAY_BG3_ACTIVE	Enables rendering of BG3
M & S	C	SPR	DISPLAY_SPR_ACTIVE	Enables rendering of Sprite
M & S	D	W0	DISPLAY_WIN0_ON	Enables the use of Window0
M & S	E	W1	DISPLAY_WIN1_ON	Enables the use of Window1

M & S	F	SW	DISPLAY_SPR_WIN_ON	Enables the use of Sprite Window
M & S	10-11	DM	MODE_FIFO	Display Mode (Engine A: 0 - 3; Engine B: 0 - 1; GBA: Green Swap)
M	12-13	DMB		Display Mode VRAM block (0- 3 = VRAM A - D) (DM >= 2)
M & S	14-15	STB	DISPLAY_SPR_1D_SIZE_32 DISPLAY_SPR_1D_SIZE_64 DISPLAY_SPR_1D_SIZE_128 DISPLAY_SPR_1D_SIZE_256	Sprite Tile 1D-Boundary (see Bit4)
M	16	SBB	DISPLAY_SPR_1D_BMP_SIZE_128 DISPLAY_SPR_1D_BMP_SIZE_256	Sprite Bitmap 1D-Boundary (see Bit5 - 6)
M & S	17	HB	DISPLAY_SPR_HBLANK	Sprite Processing during H-Blank (was located in Bit5 on GBA)
M	18-1A	CB	DISPLAY_CHAR_BASE(n)	Character Base (in 64 KB steps) (merged with 16 KB step in BGx_CR)
M	1B-1D	SB	DISPLAY_SCREEN_BASE(n)	Screen Base (in 64 KB steps) (merged with 2 KB step in BGx_CR)
M & S	1E	BE	DISPLAY_BG_EXT_PALETTE	BG Extended Palettes (0=Disable; 1=Enable)
M & S	1F	SE	DISPLAY_SPR_EXT_PALETTE	Sprite Extended Palettes (0=Disable; 1=Enable)

Table 4.4: Listing of bits in DISPLAY_CR and SUB_DISPLAY_CR

The first 16 bits act much in the same way that REG_DISPCNT did on the GBA. REG_DISPCNT was the name of the same address space in the GBA's memory that was used to control the main display. As shown in Table 4.4, Bits 0-2 control the various graphic modes that were discussed in Table 4.3. Bit 4 tells the NDS to store the Sprite data in RAM as either a one dimensional array or a two dimensional array of tiles. Knowing how these tiles are stored in memory can drastically change how they are accessed and will be discussed in

Section 4.11. Bits 8 through B turn on the various backgrounds. Each 2D engine can display up to four different backgrounds; however it is required to enable these backgrounds through the DISPLAY_CR register or else they will not render to the screen. Bit C allows the use of sprites. Bits D through F turn on the various windows that can be used in the 2D core. Windows work similarly to graphical masks and allow various backgrounds or sprites to be seen through. Windows will be described in greater detail in Section 4.20.

BACKGROUND CONTROL REGISTERS Section 4.9

Once the backgrounds are enabled through DISPLAY_CR, they need to be configured in their respective registers. As stated previously, each graphics core has four backgrounds that can be rendered to the screen at a time. Each one of these backgrounds has a memory register that controls how the background behaves. All of these registers act in the same way; their register map can be found in Register 4.2, and the bit lists can be found in Table 4.5.

Libnds name: BGx_CR, **Address:** 0x04000008 + 2x,
Sub name: SUB_BGx_CR, **Sub Address:** 0x04001008 + 2x

F E	**D**	**C B A 9 8**	**7**	**6**	**5 4 3 2**	**1 0**
Sz	Wr	MBB	CM	Mos	TBB	Pr

Register 4.2: Register map for the background control register.

Bit	Name	Definition	Description
0-1	Pr	BG_PRIORITY(n)	Priority of the background.
2-5	TBB	BG_TILE_BASE(base)	The address of the tile base block that stores the tiles for the BG.
6	Mos	BG_MOSAIC_OFF BG_MOSAIC_ON	Enables mosaic effect.
7	CM	BG_16_COLOR BG_256_COLOR	Color mode of the background (0 = 16 colors (4bpp); 1 = 256 colors (8bpp))
8-C	MBB	BG_MAP_BASE(base)	The address of the map base block that stores the map for the BG.
D	Wr	BG_WRAP_OFF BG_WRAP_ON	If set, allows affine/rotational backgrounds wrap around their edges. Has no effect on regular backgrounds because they wrap around by default.
E-F	Sz	(See Tables 4.6a-e)	Sets the size of the background. Regular and affine backgrounds have different sizes depending on settings. These sizes can be found in Tables 4.6a-e.

Table 4.5: Listing of bits in BGx_CR and SUB_BGx_CR registers.

Sz-flag	Definition	Tiles	Pixels
00	BG_32x32	32x32	256x256
01	BG_64x32	64x32	512x256
10	BG_32x64	32x64	256x512
11	BG_64x64	64x64	512x512

Table 4.6a: Regular Background map sizes.

Sz-flag	Definition	Tiles	Pixels
00	BG_RS_16x16	16x16	128x128
01	BG_RS_32x32	32x32	256x256
10	BG_RS_64x64	64x64	512x512
11	BG_RS_128x128	128x128	1024x1024

Table 4.6b: Affine Background map sizes.

Sz-flag	Definition	Pixels
00	BG_BMP8_128x128	128x128
01	BG_BMP8_256x256	256x256
10	BG_BMP8_512x256	512x256
11	BG_BMP8_512x512	512x512

Table 4.6c: 8 bit Bitmap Background map sizes.

(Since extended affine backgrounds do not include 16-color modes, Bit 7 can be used for mode selection. For these four settings Bit 7 needs to be set)

Sz-flag	Definition	Pixels
00	BG_BMP8_1024x512	1024x512
01	BG_BMP8_512x1024	512x1024

Table 4.6d: 8 bit Large Bitmap Background map sizes.

Sz-flag	Definition	Pixels
00	BG_BMP16_128x128	128x128
01	BG_BMP16_256x256	256x256
10	BG_BMP16_512x256	512x256
11	BG_BMP16_512x512	512x512

Table 4.6e: 16 bit Bitmap Background map sizes.

(Since bitmaps do not have any tile base blocks used, Bits 2-5 can be used for mode selection. For these four settings Bits 2 and 7 need to be set.)

The first two bits of the background control register are the priorities of the background. The priority of a background represents the order that the background is rendered to the screen. Because there are four backgrounds for each core, there are four levels of priority. A background that has a priority of zero means that the background is rendered on top of all other backgrounds; a background that has a priority of three means that the background is rendered beneath all other backgrounds. Because of windows (see Section 4.20) and transparencies, some backgrounds can be seen through other backgrounds. It is because of this that we have priorities so that the backgrounds render in the correct order. In case of a tie in priority or no priorities being assigned to the backgrounds, the backgrounds are rendered out in the order of their names. So if all backgrounds

were given a priority of zero, then BG0 would render out first and BG3 would render out last.

To map a background to its tileset, the background needs to be mapped to a place in memory to store the tiles called the tile base blocks, sometimes referred to as character base blocks. The tile base blocks are 16 KB offsets of memory in the Main and Sub Background Virtual Video RAM. As stated in Register 4.2 and Table 4.5, the tile base block map bits of a background are bits 2-5 in the background's control register. This section of the register tells the NDS where to find the tiles for the background memory. This data is first stored in one of the nine VRAM banks and then is mirrored to the Virtual Video RAM. As can be seen from Table 4.1, the Main core's background data is stored in between 0x06000000 and 0x0607FFFF giving the Main core 512 KB of background storage. Also, the Sub core's background data is stored in between 0x06200000 and 0x0621FFFF giving the Sub core 128 KB of background storage. This difference in storage area is because the Main core can do 3D graphics, which require more space, and the Sub core cannot.

Since each background control register has four bits dedicated to tile base blocks, this allows each background to map to 16 different tile base blocks. Also with the DISPLAY_CR bits 18-19 all of the Main core's addresses in Main Background memory can be offset by

increments of 64 KB; however, these settings do not affect the Sub core. The exact offset of the memory being used can be calculated using these formulas:

Main Tile Base Block: BGx_CR.Bits2-5 * 16 KB + DISPLAY_CR.Bits18-19 * 64 KB

Sub Tile Base Block: SUB_BGx_CR.Bits2-5 * 16 KB + 0

Tile base blocks are where the tilesets are stored, but the maps that show how those tiles are displayed on the screen are stored in the map base blocks, sometimes referred to as screen base blocks. The map base blocks are mapped over the same parts of memory as the tile base blocks; the only difference is the map base blocks are in offsets of 2 KB and the tile base blocks are in offsets of 16 KB. Because tile and map base blocks occupy the same space, it is possible to accidentally store a map over a tile, or vice versa, which would cause a corruption in some of the graphics. To prevent this it is a general practice to try to store maps early in memory and tiles late in memory and have them both grow towards each other. The background control registers dedicate Bits 8-C for map base blocks, which allows up to 32 offsets for maps. The exact memory offset inside Virtual Video RAM can be calculated with these formulas:

Main Map Base Block: BGx_CR.Bits8-C * 2 KB + DISPLAY_CR.Bits18-19 * 64 KB

Sub Map Base Block: SUB_BGx_CR.Bits8-C * 2 KB + 0

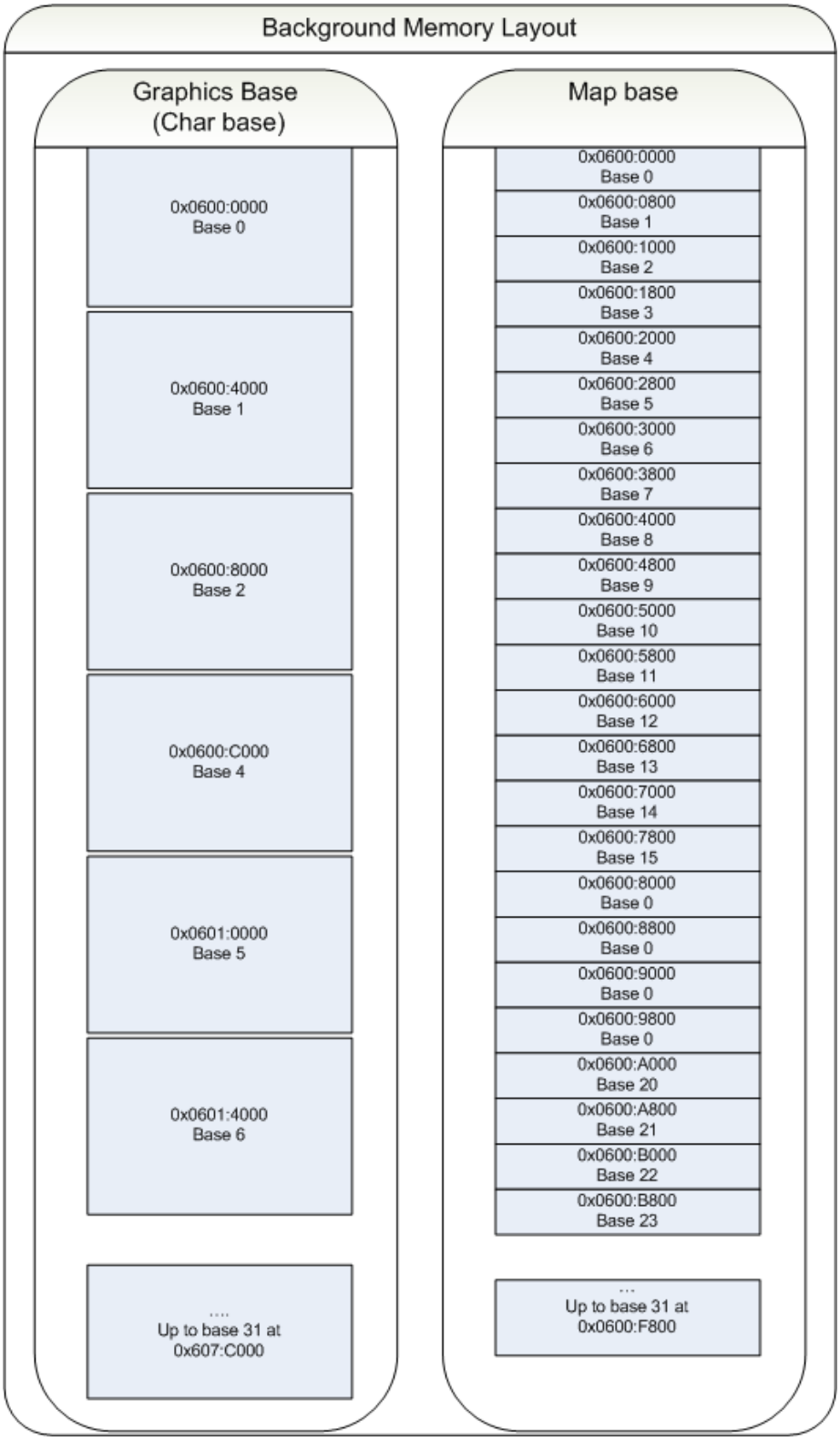

Figure 4.5: Tile and map base block offsets in the first VRAM (21).

With the tile and map base block filled with data and the correct registers set, graphics should be rendered to the screen. When looking at Tables 4.6a-e it becomes quickly apparent that some maps will have data that renders larger than the 256x192 screens that are in the NDS. In order to display the parts of the map that are off screen, a program needs to make use of two more registers. Each background has a horizontal and a vertical offset to show part of these maps. It is best to consider the map a large 2D image and the screen is a camera that shows part of that image. To move the camera around the horizontal and vertical offsets of the background the map needs to be manipulated through two registers. These registers can be seen in Register 4.3 and 4.4. Figure 4.6 demonstrates how these offsets can move the "camera" over the map. Oddly enough, these registers are write only. Since the registers cannot be read from, no calculations can be made with the values in the registers themselves. To keep track of the values in the registers, variables will need to be managed. A possible solution to this will be demonstrated in a Background struct that was used in this project.

Libnds name: BGx_X0, **Address:** 0x04000010 + 4x,
Sub name: SUB_BGx_X0, **Sub Address:** 0x04001010 + 4x

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Horizontal offset															

Register 4.3: Background horizontal placement register.

Libnds name: BGx_Y0, **Address:** 0x04000012 + 4x,
Sub name: SUB_BGx_Y0, **Sub Address:** 0x04001012 + 4x

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Vertical offset															

Register 4.4: Background vertical placement register.

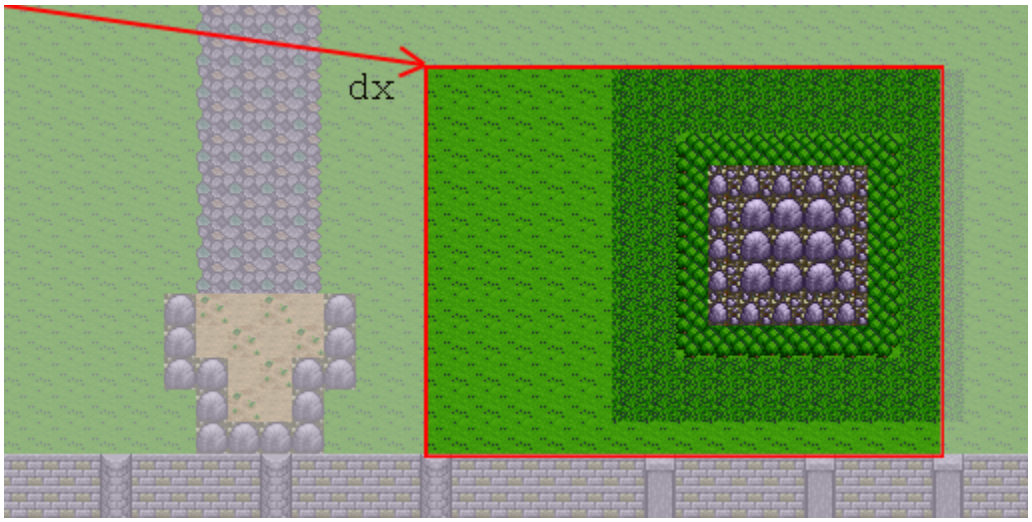

Figure 4.6: Scrolling offset **dx** is the position of the screen on the map. In this case, **dx** = (212, 32).

BACKGROUND FUNCTIONS Section 4.10

After learning how the NDS reads graphics from memory and displays it on the screen, code is written. Since there are many variables to keep track of while working with backgrounds and some of them are write only, this project uses a custom struct to interact with backgrounds. Refer to Code 4.1 to view this struct. TileData and

MapData are the points in memory where the tile and map data are stored in order to be transferred to the tile and map base blocks that TileBlock and MapBlock are set to. ScrollX and ScrollY are the camera offsets that are stored in BGx_XO and BGx_Y0. The rest of the variables just store the relevant constant to the setting desired (i.e. Bg.Mosaic = BG_MOSAIC_ON). In order for this struct to be able to do anything, three functions are necessary to do the following: enable Main backgrounds (Code 4.2), enable Sub backgrounds (Code 4.3), and update the backgrounds (Code 4.4). The only difference between EnableBackground() and EnableSubBackground() is that EnableBackground() uses the Main core registers and EnableSubBackground() uses the Sub core registers.

```
//defines the background structure that makes it easier to manipulate backgrounds
typedef struct Bg
{
   u16* TileData;          //the tile data for the background
   u16* MapData;           //the map data for the background
   u8 Mosaic;              //the mosaic setting for the background
   u8 ColorMode;           //which color mode the background is set to
   u8 Number;              //which number background it is
   u16 Size;               //the size of the background
   u8 TileBlock;           //the char block set to the background
   u8 MapBlock;            //the screen block set to the background
   u8 Wraparound;          //the wraparound setting for the background
   s16 ScrollX, ScrollY;   //the x and y offset of the background
   s32 DX,DY;              //the zoom on a rotational background
   s16 PA,PB,PC,PD;        //the rotational data of each corner of the background
   bool Main;              //Boolean flag to tell if the background is a Main
                           //background.  false == sub, true == main or normal

}Bg;
```

Code 4.1: Bg struct for keeping track of background data in Backgrounds.h.

```
// void EnableBackground(Bg* bg)
// This function takes the background sent to it and sets all of the
// appropriate registers in the display register and the
// appropriate background registers.
// Input: pointer to the background you would like to enable
// Output: None (just updates registers)

void EnableBackground(Bg* bg)
{
   u16 temp;    //temp pointer for info of background

   //load the tile data into the appropriate Char Block
   bg->TileData = (u16*)BG_TILE_RAM(bg->TileBlock);
   //load the map data into the appropriate Screen Block
   bg->MapData = (u16*)BG_MAP_RAM(bg->MapBlock);

   //make use of the temp pointer for all other info
   temp = bg->Size | bg->ColorMode | BG_MAP_BASE(bg->MapBlock) |
               BG_TILE_BASE(bg->TileBlock) | bg->Mosaic;


   //check which background was sent to the function
   switch(bg->Number)
   {
      //if case 0 set BG0 register equal to the info in temp
      case 0: BG0_CR = temp;
         //set the display register ORed to BG0 enable
         DISPLAY_CR |= DISPLAY_BG0_ACTIVE;
         break;

      //if case 1 set BG1 register equal to the info in temp
      case 1: BG1_CR = temp;
         //set the display register ORed to BG1 enable
         DISPLAY_CR |= DISPLAY_BG1_ACTIVE;
         break;

      //if case 2 set BG2 register equal to the info in temp
      case 2: BG2_CR = temp;
         //set the display register ORed to BG2 enable
         DISPLAY_CR |= DISPLAY_BG2_ACTIVE;
         break;

      //if case 3 set BG3 register equal to the info in temp
      case 3: BG3_CR = temp;
         //set the display register ORed to BG3 enable
         DISPLAY_CR |= DISPLAY_BG3_ACTIVE;
         break;
   }
}
```

Code 4.2: The EnableBackground() function in Backgrounds.cpp.

```
// void EnableSubBackground(Bg* bg)
// This function takes the Sub background sent to it and sets all of the
// appropriate registers in the display register and the
// appropriate background registers.
// Input: pointer to the background you would like to enable
// Output: None (just updates registers)

void EnableSubBackground(Bg* bg)
{
   u16 temp;    //temp pointer for info of background

   //load the tile data into the appropriate Char Block
   bg->TileData = (u16*)BG_TILE_RAM_SUB(bg->TileBlock);
   //load the map data into the appropriate Screen Block
   bg->MapData = (u16*)BG_MAP_RAM_SUB(bg->MapBlock);

   //make use of the temp pointer for all other info
   temp = bg->Size | bg->ColorMode | BG_MAP_BASE(bg->MapBlock) |
               BG_TILE_BASE(bg->TileBlock) | bg->Mosaic;
   //check which background we are dealing with
   switch(bg->Number)
   {
      //if case 0 set BG0 register equal to the info in temp
      case 0: SUB_BG0_CR = temp;
         //set the display register ORed to BG0 enable
        SUB_DISPLAY_CR |= DISPLAY_BG0_ACTIVE;
         break;

      //if case 1 set BG1 register equal to the info in temp
      case 1: SUB_BG1_CR = temp;
         //set the display register ORed to BG1 enable
        SUB_DISPLAY_CR |= DISPLAY_BG1_ACTIVE;
         break;

      //if case 2 set BG2 register equal to the info in temp
      case 2: SUB_BG2_CR = temp;
         //set the display register ORed to BG2 enable
        SUB_DISPLAY_CR |= DISPLAY_BG2_ACTIVE;
         break;

      //if case 3 set BG3 register equal to the info in temp
      case 3: SUB_BG3_CR = temp;
         //set the display register ORed to BG3 enable
        SUB_DISPLAY_CR |= DISPLAY_BG3_ACTIVE;
         break;
   }
}
```

Code 4.3: The EnableSubBackground() function in Backgrounds.cpp.

```
// void UpdateBackground(Bg* bg)
// This function takes the background sent to it and sets the new x
// and y offsets and the rotation of the background.
// Input: pointer to the background to be updated
// Output: None (but updates the registers of the background)

void UpdateBackground(Bg* bg)
{
   if(bg->Main == true)
   {
      switch(bg->Number)
      {
         //if case 0 set the Hor and Ver offsets = the backgrounds's x & y scrolls
         case 0: BG0_X0 = bg->ScrollX;
            BG0_Y0 = bg->ScrollY;
            break;

         //if case 1 set the Hor and Ver offsets = the backgrounds's x & y scrolls
         case 1:    BG1_X0 = bg->ScrollX;
            BG1_Y0 = bg->ScrollY;
            break;

         case 2:    //if case 2
            if(!(DISPLAY_CR & MODE_0_2D))//it is a rot background
            {
               BG2_CX = bg->DX; //set the x and y settings for the background
               BG2_CY = bg->DY;

               BG2_XDX = bg->PA;//set the rotational info for the background
               BG2_XDY = bg->PB;
               BG2_YDX = bg->PC;
               BG2_YDY = bg->PD;
            }
            else           //it is a text background
            {
               //set the Hor and Ver offsets = the backgrounds's x and y scrolls
               BG2_X0 = bg->ScrollX;
               BG2_Y0 = bg->ScrollY;
            }
            break;

         case 3: //if case 3
            if(!(DISPLAY_CR & MODE_0_2D))//it is a rot background
            {
               BG3_CX = bg->DX; //set the x and y settings for the background
               BG3_CY = bg->DY;

               BG3_XDX = bg->PA;//set the rotational info for the background
               BG3_XDY = bg->PB;
               BG3_YDX = bg->PC;
               BG3_YDY = bg->PD;
            }
```

```
            else            //it is a text background
            {
               //set the Hor and Ver offsets = the backgrounds's x and y scrolls
               BG3_X0 = bg->ScrollX;
               BG3_Y0 = bg->ScrollY;
            }
            break;
      }
   }
   else
   {
      switch(bg->Number)
      {
         //if case 0 set the Hor and Ver offsets = the backgrounds's x & y scrolls
         case 0: SUB_BG0_X0 = bg->ScrollX;
            SUB_BG0_Y0 = bg->ScrollY;
            break;

         //if case 1 set the Hor and Ver offsets = the backgrounds's x & y scrolls
         case 1:    SUB_BG1_X0 = bg->ScrollX;
            SUB_BG1_Y0 = bg->ScrollY;
            break;

         case 2:    //if case 2
            if(!(SUB_DISPLAY_CR & MODE_0_2D))//it is a rot background
            {
               SUB_BG2_CX = bg->DX; //set the x and y settings for the background
               SUB_BG2_CY = bg->DY;

               SUB_BG2_XDX = bg->PA;//set the rotational info for the background
               SUB_BG2_XDY = bg->PB;
               SUB_BG2_YDX = bg->PC;
               SUB_BG2_YDY = bg->PD;
            }
            else            //it is a text background
            {
               //set the Hor and Ver offsets = the backgrounds's x and y scrolls
               SUB_BG2_X0 = bg->ScrollX;
               SUB_BG2_Y0 = bg->ScrollY;
            }
            break;

         case 3: //if case 3
            if(!(DISPLAY_CR & MODE_0_2D))//it is a rot background
            {
               SUB_BG3_CX = bg->DX; //set the x and y settings for the background
               SUB_BG3_CY = bg->DY;

               SUB_BG3_XDX = bg->PA;//set the rotational info for the background
               SUB_BG3_XDY = bg->PB;
               SUB_BG3_YDX = bg->PC;
               SUB_BG3_YDY = bg->PD;
            }
```

```
            else            //it is a text background
            {
               //set the Hor and Ver offsets = the backgrounds's x and y scrolls
               SUB_BG3_X0 = bg->ScrollX;
               SUB_BG3_Y0 = bg->ScrollY;
            }
            break;
      }
   }
}
```

Code 4.4: The UpdateBackground() function in Backgrounds.cpp.

So after building functions to enable and update backgrounds, backgrounds are ready to be used in the main program. All of the background enabling and the initialization code are placed in one function in main.cpp called InitBackgrounds(), which can be seen in full in Code 4.5. In this function many constants are used. All constants are built-in parts of libnds except for SPRITE_DMA_CHANNEL which is declared in Sprites.h to be 3. Also note the use of right bit shifting. The GBA and NDS do not have any type of mathematical logic or operations unit, so there is no support for hardware multiplication or division. While multiplying is just emulated by multiple additions of numbers, divisions are emulated by a very costly algorithm. A bit shift in either direction takes one clock cycle, so it is much more efficient to do bit shifts if dividing by a power of two. In this occurrence, division is by 8, with a resultant bit shift to the right 3 times to make the calculations more efficient.

```
void InitBackgrounds()
{
   int index;        //counting variable for a for loop
   int NumberTiles;//variable to calculate how many tiles are in the map so to
                     //    determine how much data to transfer

   bg0.Number = 0;                  //background number 0
   bg0.TileBlock = 0;               //sets background 0 to Character Base Block 0
   bg0.MapBlock = 31;               //sets background 0 to screen base block 31
   bg0.ColorMode = BG_COLOR_256;   //sets background 0 to 256 color mode
   //sets background 0 to a Text background of 256x256
   bg0.Size = TEXTBG_SIZE_256x256;
   bg0.Mosaic = 0;                  //turns off mosaic
   bg0.ScrollX = ScreenX;           //sets background 0 to the x screen offset
   bg0.ScrollY = ScreenY;           //sets background 0 to the y screen offset
   bg0.Main = true;                 //sets background 0 as a main background

   //enables the background and sets all the proper pointers
   EnableBackground(&bg0);

   //Sets the priority of this background to zero (highest priority)
   BG0_CR |= BG_PRIORITY(0);

   //loads tile graphics for Alphabet using libnds function
   dmaCopyHalfWords(SPRITE_DMA_CHANNEL, FontData, bg0.TileData,
        FontDataSize);

   bg1.Number = 1;                  //background number 1
   bg1.TileBlock = 1;               //sets background 1 to Character Base Block 1
   bg1.MapBlock = 30;               //sets background 1 to screen base block 30
   bg1.ColorMode = BG_COLOR_256;       //sets background 1 to 256 color mode
   //sets background 1 to a Text background of 256x256
   bg1.Size = TEXTBG_SIZE_256x256;
   bg1.Mosaic = 0;                  //turns off mosaic
   bg1.ScrollX = ScreenX;           //sets background 1 to the x screen offset
   bg1.ScrollY = ScreenY;           //sets background 1 to the y screen offset
   bg0.Main = true;                 //sets background 1 as a main background

   //enables the background and sets all the proper pointers
   EnableBackground(&bg1);

   //sets BG1 to priority 2 (second to lowest priority)
   BG1_CR |= BG_PRIORITY(2);

   //loads MapTile's palette into the Main background palette using libnds function
   dmaCopyHalfWords(SPRITE_DMA_CHANNEL, MapTilesPalette, BG_PALETTE,
        MapTilesPaletteSize);

   //loads MapTile data using libnds function
   dmaCopyHalfWords(SPRITE_DMA_CHANNEL, MapTilesData, bg1.TileData,
        MapTilesDataSize);

   //Take the width and the height of the map, divide both by 8 (>>3)
```

```
    //  since tiles are 8x8 pixels,
    //  and then multiple them together to get all the tiles for the map.
    NumberTiles = (Screens[CurrentMap].MapHeight >> 3) *
         (Screens[CurrentMap].MapWidth >> 3);

    //load map data for the current screen
    for(index = 0; index < NumberTiles; index++)
       bg1.MapData[index] = Screens[CurrentMap].MapData[index];


    SubBG0.Number = 0;              //background number 0
    SubBG0.TileBlock = 0;           //sets background 0 to Character Base Block 0
    SubBG0.MapBlock = 31;           //sets background 0 to screen base block 31
    SubBG0.ColorMode = BG_COLOR_256; //sets background 0 to 256 color mode
    //sets background 0 to a Text background of 256x256
    SubBG0.Size = TEXTBG_SIZE_256x256;
    SubBG0.Mosaic = 0;              //turns off mosaic
    SubBG0.ScrollX = ScreenX;       //sets background 0 to the x screen offset
    SubBG0.ScrollY = ScreenY;       //sets background 0 to the y screen offset
    SubBG0.Main = false;            //sets background 0 as a sub background

    //enables the background and sets all the proper pointers
    EnableSubBackground(&SubBG0);

    SUB_BG0_CR |= BG_PRIORITY(3);    //sets SubBG0 to priority 3 (lowest priority)

    //loads tile graphics for Alphabet using libnds function
    dmaCopyHalfWords(SPRITE_DMA_CHANNEL, FontData, SubBG0.TileData,
         FontDataSize);

    //loads MapTile's palette into the Sub background palette using libnds function
    dmaCopyHalfWords(SPRITE_DMA_CHANNEL, MapTilesPalette, BG_PALETTE_SUB,
         MapTilesPaletteSize);

    return;
}
```

Code 4.5: Function InitBackground() in main.cpp that initializes all background data to be used.

After running these functions and waiting for a VBlank state, and then using the libnds built-in function swiWaitForVBlank(), backgrounds will finally be rendered out to the screens. It is very easy to update these screens through the Bg struct and then wait for

another VBlank to update the background registers. Most applications on the NDS follow this loop of updating variable states, waiting for the next VBlank, and then updating all of the register states so the next VDraw cycle will have the new data.

SPRITES Section 4.11

Graphics in console video game systems are comprised of backgrounds and sprites. Any type of static object that comprises the environment, background, or menu system will most likely be a background. Anything else, be it a player, character, enemy, gun fire, health gauge, etc., are sprites. Video game sprites are more commonly defined as an animated object that can move freely from the background; this is why sprites are also commonly referred to as objects or OBJ. The NDS has hardware specifically built to handle sprites and backgrounds, which is why it has such a powerful 2D engine. Most video game systems do not have powerful hardware to render sprites and backgrounds directly. Those types of video game systems usually "paint" the 2D image manually onto a 3D plane.

Sprites are similar to backgrounds in many ways; the following is a list of the most important similarities. Both sprites and backgrounds are rendered through hardware on a tile system. As described in the previous section about backgrounds, tiles are a way to shrink down the amount of pixels that get updated every screen refresh by building

graphics out of 8x8 pixel squares. Sprites also have the option of being 16-color or 256-color. Just like backgrounds, sprites have their own palette space that be either comprised of one 256-color palette or sixteen 16-color palettes. Another similarity between sprites and backgrounds is that they both render out based on a priority system. Just like backgrounds, sprites have priorities zero (the highest) to three (the lowest). A higher priority sprite will render on top of a lower priority sprite. In the event of the same priority, the one with the higher position in Object Attribute Memory (OAM) will be rendered on top. Also, sprites will render above any background of equal or lower priority. This allows sprites to be able to go behind certain background, but stand "on top" of the majority of them.

While sprites and backgrounds share much in common, they are also very different. The first difference that is encountered when working with sprites is how the sprites are mapped into memory. Sprites are not limited to a single tile and, in fact, the majority of sprites will be a combination of a few tiles (see Table 4.10 for different size options). These tiles are mapped to tile blocks in VRAM and Virtual VRAM just like background tiles. However, the programmer has the option of storing the tiles into memory as a 1D or 2D array. To view how to control the 1D or 2D memory array, refer to the explanation of Bit 4 in DISPLAY_CR as explained in Register 4.1 and

Table 4.4. For example, there is a full sprite in Figure 4.7a; Figure 4.7b shows how this would be stored in 2D mapping in which each tile row of the sprite has a 32-tile offset. However, if the tile base block is considered as a large array of tiles, then the tiles of every sprite are consecutive by rows of the sprite; as in Figure 4.7c.

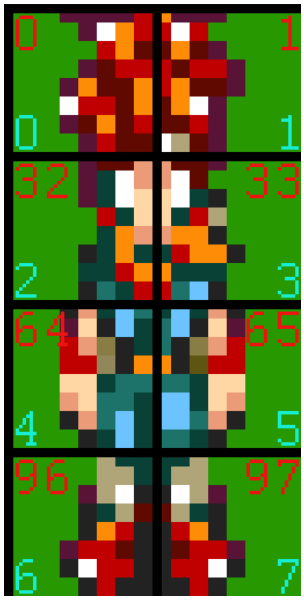

Figure 4.7a: A full sprite divided into its tiles. Colored numbers indicate mapping mode: red for 2D, cyan for 1D

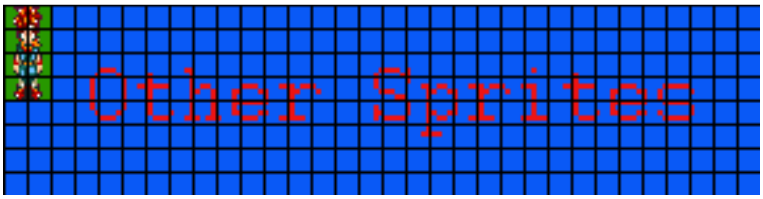

Figure 4.7b: How Figure 4.7a would be stored in memory using 2D mapping.

Figure 4.7c: How Figure 4.7a would be stored in memory using 1D mapping.

The major difference between sprites and backgrounds is how they are controlled through registers. While each background has only one control register, sprites use a few in what is known as the Object Attribute Memory (OAM). The OAM starts at address 0x07000000 (with OAM_SUB at 0x07000400) and is 1024 bytes long. A sprite uses two basic structures that a sprite uses to render itself on the screen; libnds defines these as a SpriteEntry and a SpriteRotation. The SpriteEntry structure contains what are commonly referred to as the "Object Attributes" in which there are three attribute registers that contain all the details of rendering a sprite. The GBA also has an OAM that works in the exact same manner as the NDS. On the GBA, the sprite attributes were just defined as u16 registers that stored hexadecimal numbers just like any register (see Code 4.6). Fortunately, programmers of libnds made an extremely in-depth definition of SpriteEntry so it can be used as a more fully optioned struct (see Code 4.7). The libnds definition still handles all of the three attributes; fortunately it makes controlling all the various bits much more intuitive.

The other part of OAM is the SpriteRotation data or what is also referred to as "Object Affine." Sprites can be rotated just like backgrounds and are known as either rotational or affined sprites. The

SpriteRotation structure is fairly simple and identical to the one used by most GBA programmers (see Code 4.8). An interesting part of the structures is the filler data that is apparent in all three of them. The reason that filler data is used is because the OAM is structured so that the attributes and rotational data are overlaid so that for every three attributes of a sprite there is one piece of the rotational data for a sprite, as seen in Table 4.11. Since OAM has 1024 bytes and each register takes two bytes, there is enough room for a maximum of 128 sprites and 32 affine rotations.

```
typedef struct tagOBJ_ATTR
{
   u16 attr0;
   u16 attr1;
   u16 attr2;
   s16 fill;
} ALIGN4 OBJ_ATTR;
```

Code 4.6: Basic GBA structure used to store sprite attributes.

```
typedef union {
    struct {
        struct {
            u16 posY        :8;       /**< Sprite Y position. */
            union {
                struct {
                    u8                :1;
                    bool isHidden     :1;/* Sprite is hidden (isRotoscale cleared). */
                    u8                :6;
                };
                struct {
                    bool isRotoscale  :1;/* Sprite uses affine parameters if set.*/
                    bool rsDouble     :1;// Sprite bounds is doubled (isRotoscale set).
                    tObjMode objMode  :2;       /**< Sprite object mode. */
                    bool isMosaic     :1;       /**< Enables mosaic effect if set. */
                    tObjColMode colMode       :1;       /**< Sprite color mode. */
                    tObjShape objShape        :2;       /**< Sprite shape. */
                };
            };
        };
```

```
        union {
            struct {
                u16 posX                        :9;       /**< Sprite X position. */
                u8                              :7;
            };
            struct {
                u8                              :8;
                union {
                    struct {
                        u8              :4;
                        bool hFlip:1; /* Flip sprite horizontally (isRotoscale cleared). */
                        bool vFlip:1; /* Flip sprite vertically (isRotoscale cleared).*/
                        u8              :2;
                    };
                    struct {
                        u8              :1;
                        u8 rsMatrixIdx :5; /**< Affine parameter number to use
(isRotoscale set). */
                        tObjSize objSize:2; /**< Sprite size. */
                    };
                };
            };
        };

        struct {
            u16 tileIdx                         :10;/**< Upper-left tile index. */
            tObjPriority objPriority            :2;       /**< Sprite priority. */
            u8 objPal:4;      /**< Sprite palette to use in paletted color modes. */
        };

        u16 attribute3;/**< Four of those are used as a sprite rotation matrice */
    };

    struct {
        uint16 attribute[3];
        uint16 filler;
    };

} SpriteEntry, * pSpriteEntry;
```

Code 4.7: The definition of the SpriteEntry struct in libnds.

```
typedef struct sSpriteRotation {
  uint16 filler1[3];
  int16 hdx;

  uint16 filler2[3];
  int16 hdy;

  uint16 filler3[3];
  int16 vdx;

  uint16 filler4[3];
  int16 vdy;
} SpriteRotation, * pSpriteRotation;
```

Code 4.8: The definition of the SpriteRotation struct in libnds.

SPRITE ATTRIBUTES Section 4.13

As shown above, each sprite has three attributes that control how the sprite is rendered to the screen. All three of these attributes need to be set in some capacity because they all contain various parts of the sprite's render instructions. For instance, attribute 0 contains the Y coordinate while attribute 1 contains the X coordinate that the sprite is placed at on the screen. Remember, the NDS renders from left to right and from top to bottom so (0, 0) is at the top left corner. Attribute 0 also contains flags for sprite rendering mode, the type of graphics, the mosaic effect, the color mode, and sprite shape. In addition to the X coordinate, attribute 1 contains the index to the affine rotation, horizontal and vertical flipping flags, and the sprite size. Finally, attribute 2 contains the base tile-index of the sprite's tile graphics, sprite render priority, and the palette that this sprite uses.

For more information about sprite attributes, refer to Registers 4.5-4.7 and Tables 4.7-4.9.

Sprite Attribute 0

F E	D	C	B A	9 8	7 6 5 4 3 2 1 0
Sh	CM	Mos	GM	OM	Y

Register 4.5: Attribute 0 for all sprites.

Bit	Name	SpriteEntry Variable	Value Range	Description
0-7	Y	posY	0-255	Y coordinate of the top of the sprite.
8-9	OM	isRotoscale, isHidden, rsDouble	TRUE or FALSE	The object mode of the sprite: normal, affine/rotation, disable/hidden, double size
A-B	GM	objMode	OBJMODE_NORMAL OBJMODE_BLENDED OBJMODE_WINDOWED OBJMODE_BITMAP	Graphics special effects: normal, alpha blending (see Section 4.19), object is part of object window (see Section 4.20), forbidden
C	Mos	isMosaic	TRUE or FALSE	Enables the mosaic effect (see Section 4.18)
D	CM	colMode	OBJCOLOR_16/256	The color mode the sprite uses: 16-colors (4bpp) or 256-colors (8bpp)
E-F	Sh	objShape	OBJSHAPE_SQUARE OBJSHAPE_WIDE OBJSHAPE_TALL OBJSHAPE_FORBIDDEN	Sets the shape of the sprite. Used with the sprite's size in Table 4.10.

Table 4.7: Listing of bits in all Attribute 0 registers.

Sprite Attribute 1 (Normal)

F E	D	C	B A 9	8 7 6 5 4 3 2 1 0
Sz	VF	HF	-	X

Register 4.6a: Attribute 1 when the sprite is not affine.

Bit	Name	SpriteEntry Variable	Value Range	Description
0-7	X	posX	0-511	X coordinate of the left of the sprite.
9-B	-			Not used in normal sprite mode.
C	HF	hFlip	TRUE or FALSE	A flag to flip the graphics of the sprite horizontally.
D	VH	vFlip	TRUE or FALSE	A flag to flip the graphics of the sprite vertically.
E-F	Sz	objSize	OBJSIZE_8 OBJSIZE_16 OBJSIZE_32 OBJSIZE_64	Sets the size of the sprite. Used with the sprite's shape in Table 4.10.

Table 4.8a: Listing of bits in Attribute 1 registers if the sprite is not affine.

Sprite Attribute 1 (Affine)

F E	D C B A 9	8 7 6 5 4 3 2 1 0
Sz	AID	X

Register 4.6b: Attribute 1 when the sprite is affine.

Bit	Name	SpriteEntry Variable	Value Range	Description
0-7	X	posX	0-511	X coordinate of the left of the sprite.
9-D	AID	rsMatrixIdx	0-31	Maps the sprite to one of the 32 sprite affine matrices.
E-F	Sz	objSize	OBJSIZE_8 OBJSIZE_16 OBJSIZE_32 OBJSIZE_64	Sets the size of the sprite. Used with the sprite's shape in Table 4.10.

Table 4.8b: Listing of bits in Attribute 1 registers if the sprite is affine.

Sprite Attribute 2

F E D C	B A	9 8 7 6 5 4 3 2 1 0
PB	Pr	TID

Register 4.7: Attribute 2 for all sprites.

Bit	Name	SpriteEntry Variable	Value Range	Description
0-9	TID	tileIdx	0-1023	The offset in the tile base where this sprite's graphic tiles start.
A-B	Pr	objPriority	OBJPRIORITY_0 OBJPRIORITY_1 OBJPRIORITY_2 OBJPRIORITY_3	The priority of the sprite.
C-F	PB	objPal	ATTR2_PALETTE(n)	The palette that is used by this sprite's tiles. Has no effect in 256-color mode.

Table 4.9: Listing of bits in all Attribute 2 registers.

libnds Shape/Size	OBJSIZE_8	OBJSIZE_16	OBJSIZE_32	OBJSIZE_64
OBJSHAPE_SQUARE	8x8	16x16	32x32	64x64
OBJSHAPE_WIDE	16x8	32x8	32x16	64x32
OBJSHAPE_TALL	8x16	8x32	16x32	32x64

Table 4.10: The available sprite sizes in pixels using the libnds shape and size attributes.

OAM (u16)	0	3	4	7	8	b	c	f
SpriteEntry	0 1 2		0 1 2		0 1 2		0 1 2	
SpriteRotation		hdx		hdy		vdx		vdy

Table 4.11: How SpriteEntries and SpriteRotations are overlaid in OAM.

```
//Define a constant for the NPC1 sprite
#define NPC1        0
void InitSprites(tOAM * oam, SpriteInfo *spriteInfo)
{
   //loads the Sprite's graphic's palette into the Sprite Palette in RAM.
   // In 256 color mode so use the whole palette
   //    using a libnds function
   dmaCopyHalfWords(SPRITE_DMA_CHANNEL, SpriteChronoPalette,
        SPRITE_PALETTE, SpriteChronoPaletteSize);

   //sprite 0 (NPC 1)
   //create a SpriteEntry and a SpriteInfo for the NPC1
   SpriteInfo * NPC1Info = &spriteInfo[NPC1];
   SpriteEntry * NPC1Entry = &oam->spriteBuffer[NPC1];

   //Initialize the SpriteInfo for NPC1
   NPC1Info->oamID = NPC1;
   NPC1Info->width = 16;
   NPC1Info->height = 32;
   NPC1Info->angle = 0;
   NPC1Info->entry = NPC1Entry;

   //Configure attribute 0 for NPC1 using libnds defined SpriteEntry struct
   //set sprite to 256 color, tall, and y placement = NPC1YCoordScreen
   NPC1Entry->posY = NPC1YCoordScreen;
   NPC1Entry->isRotoscale = false;
   NPC1Entry->rsDouble = false;
   NPC1Entry->objMode = OBJMODE_NORMAL;
   NPC1Entry->isMosaic = false;
   NPC1Entry->colMode = OBJCOLOR_256;
   NPC1Entry->objShape = OBJSHAPE_TALL;

   //Configure attribute 1 for NPC1 using libnds defined SpriteEntry struct
   //set size=32 and x placement = NPC1XCoordScreen
   NPC1Entry->posX = NPC1XCoordScreen;
   NPC1Entry->objSize = OBJSIZE_32;

   //Configure attribute 2 for NPC1 using libnds defined SpriteEntry struct
   //sets NPC1 to down stand (the tile offset we want) and priority 1
   NPC1Entry->tileIdx = (NPC1 * OMA_OFFSET) + SPRITE_DOWN_NORM;
   NPC1Entry->objPriority = NPCPriority;
   NPC1Entry->objPal = NPC1Info->oamID;

   //loads sprite picture for NPC 1 for this screen using libnds function
   dmaCopyHalfWords(SPRITE_DMA_CHANNEL, SpriteMagnusData,
        &SPRITE_GFX[(NPC1Info->oamID) * DMA_OFFSET], SPRITE_DATA_SIZE);
   return;}
```

Code 4.9: InitSprites() function that declares and initializes any sprites.

The function InitSprites() (see Code 4.9) is called in main() to declare and initialize sprites. The function works as a simple way to contain all sprite initialization calls much like InitBackgrounds() (see Code 4.5). This function declares a SpriteInfo and a SpriteEntry for the only sprite in this program (NPC1). In other programs this function could declare and initialize many sprites. All variables are set in the sprite, leaving no default value to chance.

Direct Memory Access Section 4.14

Many of the functions discussed so far include a reference to the dmaCopyHalfWords() function. This is a built-in function of libnds that allows use of the NDS's Direct Memory Access (DMA). The DMA is a fast way of transferring data internally in the NDS. Not only can it be used to copy data, but it can also be used to fill data. An example of filling data would be when a sprite is removed and the current palette alpha color is used to clear out all of its tiles. When DMA is activated, the DMA controller actually takes over the hardware (which temporarily halts the CPU), does the transfer, and then returns control back to the CPU. All of this is done through much quicker channels than a direct array copy would in a for loop. There are four DMA channels that range from 0 to 3 and work on the same priority system as backgrounds and sprites. Channel 0 is the highest priority and is usually reserved for time-critical operations and can only be used with

internal RAM. Channels 1 and 2 are usually used to transfer sound data to the right sound buffers for playback. Finally, the lowest priority channel, channel 3, is used for any general purpose transfer. Channel 3 is usually labeled the "Sprite DMA Channel" and is primarily used for loading in new bitmap or tile data.

All transfer routines, including the use of DMA, need 3 things: a source, a destination, and the amount of data to copy. The NDS has three separate registers for each channel of DMA. The source addresses are referred to in libnds as DMAx_SRC and reside in physical memory at "0x040000B0 + (x * 12)" where x is the DMA channel. The destination address registers work in much the same way where they are defined in libnds as DMAx_DEST and are located at "0x040000B4 + (x * 12)" where x is the DMA channel. Finally, control registers for DMA are defined in libnds as DMAx_CR and are found at "0x040000B8 + (x * 12)" where x is the DMA channel (see Register and Table 4.12 for more). All of the DMA registers (source, destination, and control) are 32 bit registers.

Libnds name: DMAx_CR, **Address:** 0x040000B8 + (x * 12)

F E D C B A 9 8 7 6 5 4 3 2 1 0
N

1F	**1E**	**1D 1C**	**1B**	**1A**	**19**	**18 17**	**16 15**	**14 13 12 11 10**
En	I	TM	-	CS	R	SA	DA	-

Register 4.8: Register map for all of the DMA control registers.

Bit	Name	Definition	Description
0-F	N		The number of transfers in this DMA call.
15-16	DA	DMA_DST_INC DMA_DST_DEC DMA_DST_FIX DMA_DST_RESET	Destination Adjustment: increment the destination, decrement the destination, none because the address is fixed, increment and then reset
17-18	SA	DMA_SRC_INC DMA_SRC_DEC DMA_SRC_FIX	Source Adjustment, works exactly like Destination Adjustment but there is no RESET mode.
19	R	DMA_REPEAT	Repeats the copy at each VBlank or HBlank if those transfers are enabled.
1A	CS	DMA_16_BIT DMA_32_BIT	The chunk size of the transfer: 16-bit or 32-bit
1C-1D	TM	DMA_START_NOW DMA_START_HBL DMA_START_VBL DMA_START_FIFO	The timing modes of the DMA channel which include: start immediately, at HBlank, at VBlank, or run on a first in, first out model
1E	I	DMA_IRQ_REQ	Allows the DMA to raise an interrupt when finished.
1F	En	DMA_ENABLE	Enables the DMA transfer for this channel.

Table 4.12: Listing of bits in the DMAx_CR registers.

Fortunately, libnds has built-in functions to help with the use of DMA. The libnds functions allow synchronous, asynchronous, and fill type of operations. All three methods have the option of being run either as a half word (16-bit) or a full word (32-bit). Both synchronous and asynchronous have a standard copy function that runs on DMA channel 3. Also, both of the fill functions run on DMA

channel 3. Otherwise, the functions leave the channel choice up to the developer.

INPUT Section 4.15

As with any computer or video game system, the NDS has various ways to provide input to the system. The NDS has a 4-way directional pad (otherwise known as a D-pad), four face buttons (A, B, X, Y), two control buttons (Start and Select), two shoulder buttons (R and L), a touch screen, a microphone, and recognition when the lid of the unit is shut. Between the buttons and the touch screen there are many opportunities to make a fully functioning interactive program. While the microphone is a rather powerful input device, it is outside the scope of this project.

BUTTON INPUT Section 4.16

To interact and read input from these buttons, two basic registers are used. The first register is defined by libnds as REG_KEYINPUT (see Register 4.9). The process of checking to see if a key is currently being pressed, or down, would be rather straightforward. However, this register works opposite to all of the other registers that have been discussed up to this point. In a state of rest, the register is full of 1s and has a value of 0x3FFF and not 0x0 as would be expected. Still this is easily avoided if kept in consideration

and all key checks are done in this way: #define KEY_DOWN_NOW(key) (~(REG_KEYINPUT) & key). In this macro the register is first inverted to a more intuitive form of a bit being set means a key is down. Then the register is masked by checking it against the key that we want to check. Note that this register and method can be used to check if combinations of keys are all down at once.

Libnds name: REG_KEYINPUT, **Address:** 0x04000130

F E	**D**	**C**	**B**	**A**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-	Lid	Touch	Y	X	L	R	Down	Up	Left	Right	Start	Select	B	A

Register 4.9: The register in memory that holds the current state of all keys.

All of this might sound like a lot of work, and in the days of GBA development it was. Fortunately the people who developed libnds took all of this into consideration and made a few useful functions to help make key input controls much friendlier to the developer. Libnds includes a simple function called scanKeys(). This function is called immediately before any key press processing is done. It scans the REG_KEYINPUT register and prepares data to be used for any of the three key state comparison functions: keysHeld(), keysDown(), and keysUp(). keysHeld() is used to scan for any keys that are down currently and have been down for any time. While keysDown() and keysUp() check for keys that have recently either have been pressed

down or let go of (the key has been allowed up) respectively. These three functions are rather powerful and can make what used to be very complex data input on the GBA, as demonstrated in Code 4.10, very easy on the NDS.

```
scanKeys();

if(keysDown() & KEY_A)
{
   //Do something only when the key is initially pressed such as make a sound.
}

if(keysHeld() & KEY_A)
{
   //Do something such as move a character, shoot at a target, increment a timer.
}

if(keysUp() & KEY_A)
{
   //Do something only when the key is let go, such as stop a timer.
}
```

Code 4.10: An example of how keysDown(), keysHeld(), and keysUp() can make the same button have three different functionalities.

The other register, REG_KEYCNT, is used only for a key-based interrupt and can be seen in Register 4.10 and Table 4.13. REG_KEYCNT has the same 14 bit positions for all the keys that exist on the NDS just like REG_KEYINPUT. REG_KEYCNT, however, works just like all of the other NDS registers in which bits need to be set in order to be used. First a developer would want to set the bits for the keys that would trigger the interrupt. Then, Bit 14 would need to be

set to enable the interrupt. Finally, Bit 15 would either need to be cleared or set depending on how the developer wants the keys to be evaluated. If the developer would wish that any of the keys that are set in Bits 0-13 triggers the interrupt, then Bit 15 needs to be cleared. If the developer would like multiple keys being pressed simultaneously to trigger the interrupt, then Bit 15 would need to be set. While REG_KEYCNT is not necessary for use to read in input, it can be used to do some impressive things, for example, the REG_KEYCNT can cause a software reset as demonstrated in Code 4.11.

Libnds name: REG_KEYCNT, **Address:** 0x04000132

F	**E**	**D**	**C**	**B**	**A**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Op	I	Lid	Touch	Y	X	L	R	Down	Up	Left	Right	Start	Select	B	A

Register 4.10: The register in memory that holds the settings for a key-based interrupt.

Bit	**Name**	**Definition**	**Description**
0-D	Keys	KEY_x	The same keys that are in REG_KEYINPUT. These are the keys that will set off the interrupt
E	I	BIT(14)	Enables the keypad interrupt.
F	Op	BIT(15)	Boolean operator to decide which way the keys will be evaluated to cause the interrupt. If clear, it uses an OR (raise the interrupt if any of the keys selected are down). If set, it uses an AND (raise the interrupt if all of the keys selected are down).

Table 4.13: Listing of bits in the REG_KEYCNT register.

```
irqSet(IRQ_KEYS, swiSoftReset); // Enable Keypad interrupt and set it to run
        //swiSoftReset upon interrupt.  irqSet is a libnds function.
// sets the key interrupt and triggers it when Select, Start, A, and B are pressed
//BIT(14) makes sure the key interrupt is set
//BIT(15) makes it so the interrrupt is triggered when all the
// keys are pressed and not just any of the keys
//The rest sets the interrupt to be triggered by pressing select + start + A + B
REG_KEYCNT |= KEY_A | KEY_B | KEY_SELECT | KEY_START | BIT(14) | BIT(15);
```

Code 4.11: This is an example of the use of the REG_KEYCNT register to set up a key press interrupt.

TOUCH SCREEN INPUT Section 4.17

In both REG_KEYINPUT and REG_KEYCNT, Bit 12 just represents if the screen is touched. Unlike the other inputs on the NDS, the touch screen is not just a Boolean that is either on or off. The touch screen records the coordinates of the touch and in this way the developer can make the objects on the screen interactive to the user. However, to actually use the data of what was touched (or more appropriate, the coordinate on the screen that is being touched), more information is needed. Fortunately, libnds also has built-in structs to handle this data and make it easy for a developer to interact with it. This struct is called touchPosition and holds the px and py variables. The variables px and py hold coordinates on the touch screen. To update the struct to the currently touched position on the touch screen, the developer simply has to call the other libnds function touchReadXY() (for a simple code demonstration of this, refer to Code 4.12). Then the

developer can use the touchPosition struct to interact with the REG_KEYINPUT state of the KEY_TOUCH variable (see Code 4.13).

```
//To read from the touch screen, declare a variable and set it to touchReadXY()
touchPosition touch = touchReadXY();
touch->px; // This contains the x pixel location.
touch->py; // This contains the y pixel location.
```

Code 4.12: This is a simple example of how the coordinates of the touch screen are loaded into a variable.

```
//Variable to hold the current X and Y values on the touch screen.
int XTouch = 0;
int YTouch = 0;

//Set the current coordinates on the touch screen to a touchPosition struct
touchPosition touchXY = touchReadXY();

//If the touch is being held and not recently down or up
if(keysHeld() & KEY_TOUCH)
{
   XTouch = touchXY.px; // This contains the x pixel location.
   YTouch = touchXY.py; // This contains the y pixel location.
}
else
{
   XTouch = 0;
   YTouch = 0;
}

//Top answer box
if( 10 <= XTouch && XTouch <= 100 &&
   10 <= YTouch && YTouch <= 100 )
{
   //Do something depending on what the user touched on the screen
}
```

Code 4.13: This is an example of how to use the coordinates received from a call to touchReadXY().

After learning how to interact with the touch screen, some interesting input possibilities open up. For example, in the open source, user created PALib there is a function to do character recognition of the Graffiti PDA language (see Figure 4.8). Using this function a developer can create a new text entry process for users that allows them to enter letters and words with the stylus instead of navigating through the letter menu systems found on most video game consoles. Through the use of the PA_CheckLetter() function, the input from the stylus can be recorded and matched to an alphabet letter. To see this function in action, refer to Code 4.14. Note that in this case the return symbol is being used as an exit from the text entering.

Figure 4.8: The PA Graffiti alphabet (22). Note the last three characters are backspace, forward space, and the return characters.

```
bool EndGraffiti = false;

// Until the user is done, stay in loop
while (EndGraffiti == false)
{
	scanKeys();

	// Reset the screen when we start a new character
	if(keysDown() & KEY_TOUCH)
		PA_Clear8bitBg(0);

	PA_8bitDraw(0, 1);

	char letter = PA_CheckLetter(); // Returns the letter

	// there is a new letter
	if (letter > 31)
	{
		Name[NameLength] = letter;
		NameLength++;
	}
	// Backspace is entered
	else if ((letter == PA_BACKSPACE) && NameLength)
	{
		NameLength--;
		Name[NameLength] = ' '; // Erase the last letter
	}
	//return symbol is entered
	else if (letter == '\n')
	{
		EndGraffiti = true;
	}

	PA_OutputText(1, 2, 5, Name); // Write the text

	PA_OutputSimpleText(1, 2, 0,
		"Welcome to the NDS IBL system.  Please enter your name.");

	swiWaitForVBlank();
}
```

Code 4.14: A simple while loop that shows how PA_Graffiti can be used through the PA_CheckLetter() function.

Another common use of the touch screen as input is a keyboard application. Many keyboard applications are available for the NDS in the homebrew community, some through the open source PALib and

others through user-created experiments shared with the community. The keyboard application used in this project was borrowed from HeadKaze's webpage (http://headkaze.drunkencoders.com/). The keyboard used in particular is Extended Keyboard Example 5. HeadKaze allows his programs to be used by the community at large as long as credit is given. This is how much of the homebrew community works. The community, generally, wishes that people would make progress as a collective and not as individuals, so sharing code and applications is very common. All that was necessary to get HeadKaze's keyboard functions to work in this project was the inclusion of a few files and a few simple function calls (see Code 4.15 for an example).

```
//Keyboard functions and graphic provided by HeadKaze
#include “keyboard.c”

//A bool to decide when typing is done
bool EndTyping = false;

//While not done typing
while(EndTyping == false)
{
   //Scankeys for input
   scanKeys();

   //Set test to a character returned by the keyboard function
   test = processKeyboard(&str[0], MAX_TEXT, ECHO_ON);

   //If test is 0 do nothing.
   if(test == 0)
   {}
   //else if test is the return character
   else if(test=='\n')
   {
      //Clear the line being outputed to the current line on string
      strcpy(str, “”);

      //Add a return character to the string
      Question1[Question1Length] = test;
      Question1Length++;
   }
   //else if the close application button was touched
   else if(test == '\x1')
   {
      //end typing for the user
      EndTyping = true;
   }
   //else some other key was pressed
   else
   {
      //Add the character to the string
      Question1[Question1Length] = test;
      Question1Length++;
   }

   swiWaitForVBlank();
}
```

Code 4.15: A simple example of how the keyboard functions can be used to collect data.

As shown throughout this section there are many ways to collect data on the NDS. Like many other video game console systems, buttons provide much of the input to the system. However, the NDS is unique in that it is equipped with a touch screen. This touch screen allows many unique and interesting opportunities for interactivity by both the developer and the use. Not only can users touch objects, but they can draw, enter text like on a PDA system, and even type on a virtual keyboard.

GRAPHIC EFFECTS - MOSAIC Section 4.18

The NDS has hardware support for graphic effects such as mosaic, alpha blending, and windowing. Mosaic is a graphics effect that became popular in the 1990s both in video games and other media like television and film. Mosaic effects were introduced on the Super Nintendo hardware first with the release of the launch title Super Mario World. Mosaic effects were used as a transition effect and have since become a staple of what is considered the "look" of the 1990s.

The effect works by taking in two dimensions with parameters **w** and **h**. These numbers divide the sprite or background into blocks of size **w** x **h** pixels. These blocks act like pixels on a much larger scale by using the top-left pixel of each block and using it to fill the rest of the block, which is what causes the "blocky" effect. To see the effect

in action, refer to Figure 4.9. Figure 4.9 shows a normal sprite on the left and a sprite that has been put through a 2x2 mosaic (stretched 2 times both horizontally and vertically). The black lines in the pictures indicate the vertical block-boundaries of the mosaic effect in the sprite. In the picture that is mosaiced, the top-left pixel is copied down into the 2x2 matrix in each block.

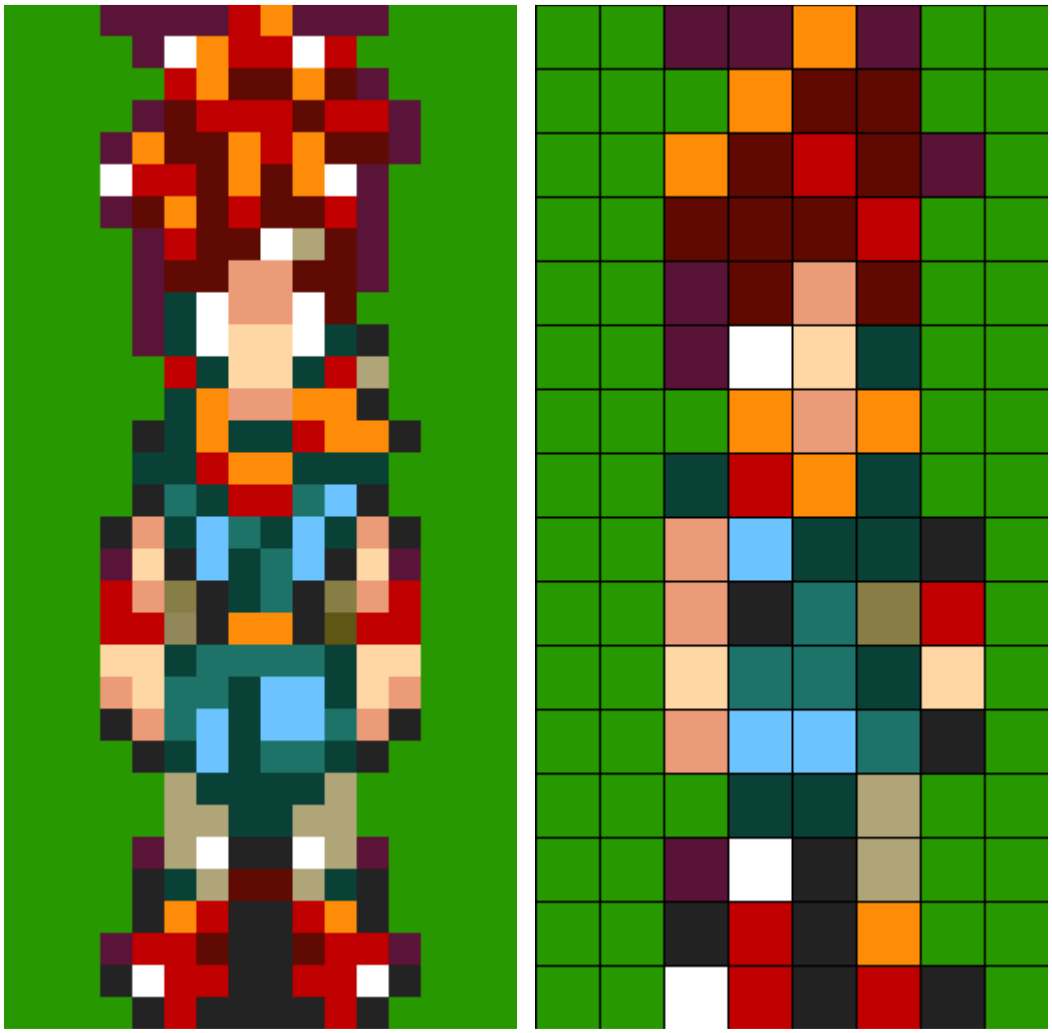

Figure 4.9: A normal sprite on the left and a 2x2 mosaiced version on the right.

This effect is not trendy anymore; however it still can be used on the NDS very easily on either screen. First, on any background or sprite, mosaic needs to be enabled. To do this in background x, Bit 6 of BGx_CR (see Register 4.2 and Table 4.5) needs to be enabled. To enable mosaic on a sprite, Bit 12 of attribute 0 needs to be set for each sprite to which the mosaic effect applies (see Register 4.5 and Table 4.7). The values that the sprites or backgrounds need to be mosaiced by are entered in MOSAIC_CR (see Register 4.9 and Table 4.13). Each portion of MOSAIC_CR is divided into a four bit section. This means that both backgrounds and sprites have up to 16 different mosaic states in each direction. Generally a cycle takes place to go from state zero to state 16 or vice versa (see Code 4.16 for an example).

Libnds name: MOSAIC_CR, **Address:** 0x0400004C,
Sub name: SUB_MOSAIC_CR, **Sub Address:** 0x0400104C

F E D C	**B A 9 8**	**7 6 5 4**	**3 2 1 0**
Ov	Oh	Bv	Bh

Register 4.11: The contents of the Mosaic control register.

Bit	**Name**	**Description**
0-3	Bh	The horizontal stretch of the backgrounds.
4-7	Bv	The vertical stretch of the backgrounds.
8-B	Oh	The horizontal stretch of sprites or objects.
C-F	Ov	The vertical stretch of sprites or objects.

Table 4.14: Listing of bits in MOSAIC_CR.

```
void MosaicFunc()
{
   int MosaicState = 0;  //The current mosaic state

   for(MosaicState = 0; MosaicState < 16; MosaicState++)
   {
      MOSAIC_CR = 0;    //Clears the current Mosaic state

      MOSAIC_CR = ( ((MosaicState)) |          //Sets the BG horizontal stretch
                  (((MosaicState))<<4) |       //Sets the BG vertical stretch
                  (((MosaicState))<<8) |       //Sets the sprite horizontal stretch
                  (((MosaicState))<<12) );     //Sets the sprite vertical stretch

      swiWaitForVBlank();    //Wait for the screen to be drawn
   }

   return;
}
```

Code 4.16: Example of a function that effects both sprite and background mosaic levels.

GRAPHIC EFFECTS – ALPHA BLENDING Section 4.19

Another graphic effect that is used frequently on the NDS is alpha blending. Alpha blending allows the combination of color values of two overlapping layers, which creates a semi-transparency of one of the layers. For example, if there are two layers, A and B, that overlap each other, and consider A to be on top of B, then the color value of a pixel in this region is defined as

C = WA * A + WB * B

where WA and WB are the weights of the layers. Usually these weights are normalized between the values of 0 and 1, where 0 would represent a layer being fully transparent and 1 would represent a layer being fully visible. These values allow either of the layers being fully

transparent or a mixture of the two layers being combined as shown in Table 4.18. Note that in these values the sum of the weights is 1 so that the final color of C is between 0 (black) and 1 (white) as well.

On the NDS, backgrounds are always enabled for blending. However, to allow blending in sprites, the sprite's Attribute 0 Bit A has to be set (as seen in Register 4.5 and Table 4.7). Three registers control blending on the NDS; these registers are defined in libnds as BLEND_CR, BLEND_AB, and BLEND_Y (Registers 4.12-4.14 and Tables 4.15-4.17). BLEND_CR defines which parts of the graphics system (Backgrounds, sprites, and backdrop) are in which layers of the blend. This register also defines which type of blend mode is in use. BLEND_AB defines the weights for the A and B layers to use when blending. Finally BLEND_Y defines the weight that is being used while fading to black or white, which are two of the modes defined in BLEND_CR. Note that since the NDS does not compute floating point numbers, the weights in BLEND_AB and BLEND_Y are fixed-point numbers in 1.4 format. Because of this, and the fact that they are binary values, there are 17 blend levels in each of these registers.

Libnds name: BLEND_CR, **Address:** 0x04000050,
Sub name: SUB_BLEND_CR, **Sub Address:** 0x04001050

F E	D	C	B	A	9	8	7 6	5	4	3	2	1	0
-	bBD	bSpr	bBG3	bBG2	bBG1	bBG0	BM	aBD	aSpr	aBG3	aBG2	aBG1	aBG0

Register 4.12: The blend control register.

Bit	Name	Definition	Description
0	aBG0	BLEND_SRC_BG0	Use BG0 as part of the A layer of the blend.
1	aBG1	BLEND_SRC_BG1	Use BG1 as part of the A layer of the blend.
2	aBG2	BLEND_SRC_BG2	Use BG2 as part of the A layer of the blend.
3	aBG3	BLEND_SRC_BG3	Use BG3 as part of the A layer of the blend.
4	aSpr	BLEND_SRC_SPRITE	Use all blend enabled sprites as part of the A layer of the blend.
5	aBD	BLEND_SRC_BACKDROP	Use the backdrop as part of the A layer of the blend. The backdrop is a solid plane of color 0 (black).
6-7	BM	BLEND_NONE BLEND_ALPHA BLEND_FADE_WHITE BLEND_FADE_BLACK	The type of blending being used ie off, standard alpha blending, fade to white, or fade to black.
8	bBG0	BLEND_DST_BG0	Use BG0 as part of the B layer of the blend.
9	bBG1	BLEND_DST_BG1	Use BG1 as part of the B layer of the blend.
A	bBG2	BLEND_DST_BG2	Use BG2 as part of the B layer of the blend.
B	bBG3	BLEND_DST_BG3	Use BG3 as part of the B layer of the blend.
C	bSpr	BLEND_DST_SPRITE	Use all blend enabled sprites as part of the B layer of the blend.
D	bBD	BLEND_DST_BACKDROP	Use the backdrop as part of the B layer of the blend. The backdrop is a solid plane of color 0 (black).

Table 4.15: Listing of all the bits in the BLEND_CR register.

Libnds name: BLEND_AB, **Address:** 0x04000052,
Sub name: SUB_BLEND_AB, **Sub Address:** 0x04001052

F E D	C B A 9 8	7 6 5	4 3 2 1 0
-	evb	-	eva

Register 4.13: The A and B weight blend control register.

Bit	Name	Definition	Description
0-4	eva	(no libnds definition)	The top blend weight used for layer A. Only used in alpha blend mode.
8-C	evb	(no libnds definition)	The bottom blend weight used for layer B. Only used in alpha blend mode.

Table 4.16: Listing of all the bits in the BLEND_AB register.

Libnds name: BLEND_Y, **Address:** 0x04000054,
Sub name: SUB_BLEND_Y, **Sub Address:** 0x04001054

F E D C B A 9 8 7 6 5	4 3 2 1 0
-	evy

Register 4.14: The Y weight blend control register.

Bit	Name	Definition	Description
0-4	evy	(no libnds definition)	The weight used in fades to black and white in layers.

Table 4.17: Listing of all the bits in the BLEND_Y register.

WA	WB	Effect
1	0	Layer A is fully visible and Layer B is hidden.
0	1	Layer B is fully visible and Layer A is hidden.
a	1-a	Alpha blending, a is the opacity in this case.

Table 4.18: Listing of all the different uses of weighted layers in alpha blending.

A few minor rules must be followed for alpha blending to work correctly on the NDS. All elements in Layer A must be in front of the elements in Layer B. This means that all elements in Layer A need to

have a higher priority than the elements in Layer B to blend properly. Also, when in alpha-blend mode, the blend will only occur on the pixels that are overlapping and non-transparent in Layers A and B. Any pixels that are non-overlapping will not be affected by the blend. Finally, if windows are in use the correct bits need to be enabled in the WIN_IN or WIN_OUT registers or else no blend effects will work correctly (please refer to Bits 5 and D in Registers 4.15-4.16 and Tables 4.20-4.21). For a brief example of how blending works please refer to Code 4.17 for an example of fading out to black and then fading in from black.

```
// void FadeOut(void)
// This function takes the current screen and fades to black.
//
// Input: None
// Output: None (Fades screen to black.)
void FadeOut(void)
{
   int y;    //loop variables

   //Set fade control register to fade to black BG0, BG1, and sprites
   BLEND_CR = BLEND_SRC_BG0 | BLEND_SRC_BG1 | BLEND_SRC_SPRITE |
        BLEND_FADE_BLACK;

   for(y = 0; y < 17; ++y)    //cycle through all 16 modes of fading
   {
      BLEND_Y = y;

      swiWaitForVBlank();    //Wait for the screen to be drawn
   }

   return;
}


// void FadeIn(void)
// This function loads the new screen and then fades from black.
//
// Input: None
// Output: None (Loads new screen and then fades screen from black.)
void FadeIn(void)
{
   int y;    //loop variables

   //Set fade control register to fade to black BG0, BG1, and sprites
   BLEND_CR = BLEND_SRC_BG0 | BLEND_SRC_BG1 | BLEND_SRC_SPRITE |
BLEND_FADE_BLACK;

   for(y = 16; y >= 0; --y) //cycle through all 16 modes of fading
   {
      BLEND_Y = y;

      swiWaitForVBlank();    //Wait for the screen to be drawn
   }
   return;}
```

Code 4.17: Example functions of how to fade out to black and fade in from black.

One more graphic effect that is used by the NDS is windowing. Windowing allows the developer to divide the screen into separate regions. Two basic windows are known as Win0 and Win1 and a lesser used window is called the Sprite Window. While Win0 and Win1 can be used to show a portion of the display screen, the Sprite Window is a window created out of the visible pixels of all the sprites on screen. These windows are enabled in the DISPLAY_CR register as seen in Register 4.1 and Table 4.4. Win0 and Win1 are defined by their left, right, top, and bottom boundaries. These boundaries are represented by the X and Y coordinates of the line that is the limit of the side of the window.

As shown in Table 4.19 each window has 8 bits for each coordinate with a value range of 0 to 255, which is enough for the screen width and a little extra for the screen height. The NDS hardware does not support any types of overlap or out-of-range values for the windows. If either of these two scenarios occurs, the NDS will not render the window. For example, if a bottom coordinate of a window is placed above a top coordinate, then the NDS will not render. The same thing would happen if either the top or bottom coordinate went over 192, which is the screen height of the NDS. In addition to the size of the window, a developer needs to determine

which elements of the graphical system will be rendered inside and outside of the window. In this way, windows can be used as masks to show parts of a lower priority background or sprite. Finally, the windows have their own priority structure. Win0 takes priority over Win1, which takes priority over WinOut. Refer to Code 4.18 for a simple demonstration of opening, expanding, collapsing, and then closing a window.

Libnds name	Address	Sub name	Sub Address	Description
WIN0_X0	0x04000041	SUB_WIN0_X0	0x04001041	The left boundary of Win0.
WIN0_X1	0x04000040	SUB_WIN0_X1	0x04001040	The right boundary of Win0.
WIN0_Y0	0x04000045	SUB_WIN0_Y0	0x04001045	The top boundary of Win0.
WIN0_Y1	0x04000044	SUB_WIN0_Y1	0x04001044	The bottom boundary of Win0.
WIN1_X0	0x04000042	SUB_WIN1_X0	0x04001042	The left boundary of Win1.
WIN1_X1	0x04000043	SUB_WIN1_X1	0x04001043	The right boundary of Win1.
WIN1_Y0	0x04000047	SUB_WIN1_Y0	0x04001047	The top boundary of Win1.
WIN1_Y1	0x04000046	SUB_WIN1_Y1	0x04001046	The bottom boundary of Win1.

Table 4.19: A list of all the window registers.

Libnds name: WIN_IN, **Address:** 0x04000048,
Sub name: SUB_WIN_IN, **Sub Address:** 0x04001048

F E	D	C	B	A	9	8	7 6	5	4	3	2	1	0
-	Bld	Spr	BG3	BG2	BG1	BG0	-	Bld	Spr	BG3	BG2	BG1	BG0
-	Win1						-	Win0					

Register 4.15: The WIN_IN register.

Bit	Name	Description
0-5	Win0	Select which elements render inside Win0. 0-3 are backgrounds, 4 is sprites, and 5 is if blending is allowed.
8-D	Win1	Select which elements render inside Win1. 8-B are backgrounds, C is sprites, and D is if blending is allowed.

Table 4.20: Listing of the bits in the WIN_IN register.

Libnds name: WIN_OUT, **Address:** 0x0400004A,
Sub name: SUB_WIN_OUT, **Sub Address:** 0x0400104A

F E	D	C	B	A	9	8	7 6	5	4	3	2	1	0
-	Bld	Spr	BG3	BG2	BG1	BG0	-	Bld	Spr	BG3	BG2	BG1	BG0
-	WinSpr						-	WinOut					

Register 4.16: The WIN_OUT register.

Bit	Name	Description
0-5	WinOut	Select which elements render outside Win0 and Win 1. 0-3 are backgrounds, 4 is sprites, and 5 is if blending is allowed.
8-D	WinSpr	Select which elements render inside the Sprite Window. 8-B are backgrounds, C is sprites, and D is if blending is allowed.

Table 4.21: Listing of the bits in the WIN_OUT register.

```
//Shows BG0, BG1, Sprites, and Blends in Win0
WIN_IN = WIN0_BG0 | WIN0_BLENDS

//make the text box appear to expand from a central point
for(x = ((Bottom - Top) >> 1); x >= 0; x--)
{
   //Store the Text Box Window's Left boundary in it's register
   WIN0_X0 = LEFT((Left + (2 * x)));
   //Store the Text Box Window's Right boundary in it's register
   WIN0_X1 = RIGHT((Right - (2 * x)));
   //Store the Text Box Window's Top boundary in it's register
   WIN0_Y0 = TOP((Top + x));
   //Store the Text Box Window's Bottom boundary in it's register
   WIN0_Y1 = BOTTOM((Bottom - x));

   //wait for end of draw for syncing, sync a few times for a delay
   swiWaitForVBlank();
   swiWaitForVBlank();
   swiWaitForVBlank();
}

//wait for screen to stop drawing to sync
swiWaitForVBlank();

//make the text box appear to shrink to a central point
for(x = 0; x <= ((Bottom - Top) >> 1); x++)
{
   //Store the Text Box Window's Left boundary in it's register
   WIN0_X0 = LEFT((Left + (2 * x)));
   //Store the Text Box Window's Right boundary in it's register
   WIN0_X1 = RIGHT((Right - (2 * x)));
   //Store the Text Box Window's Top boundary in it's register
   WIN0_Y0 = TOP((Top + x));
   //Store the Text Box Window's Bottom boundary in it's register
   WIN0_Y1 = BOTTOM((Bottom - x));

   //wait for screen to stop drawing to sync
   swiWaitForVBlank();
}

//Shows just BG0, BG1, Sprites, and Blends in Win0 (excludes BG0)
WIN_IN = WIN0_BG1 | WIN0_SPRITES | WIN0_BLENDS;
```

Code 4.18: Example of how to use the Window registers.

As discussed previously, the NDS has built-in Wi-Fi IEEE 802.11b and g support with WEP encryption. The NDS has built-in firmware that allows the user to set up access points through Wi-Fi enabled games. Access points can be completely open or have basic WEP encryption; however the NDS does not support WPA or more complicated types of encryption. The NDS also supports a special wireless format created by Nintendo and secured using RSA security signing which is used by the built-in chatting program, PictoChat. Wi-Fi on the NDS can be used to play games on a wireless LAN with people in the same physical area, or to play games through the Nintendo Wi-Fi Connection service. The Nintendo Wi-Fi Connection service is a free service provided by Nintendo, which allows users to connect to other users to play games.

The first breakthrough for NDS Wi-Fi homebrew was when Steve Stair released his dswifi library (13). Since then, the library has been added to devkitPro and has become open source; many users in the homebrew community provide support for the library. Furthermore, the library has been adapted into other, more comprehensive NDS homebrew libraries, such as PALib. The majority of the dswifi library is low-level in regards to how closely it works with Wi-Fi protocols directly and does not have extensive documentation. PALib has built

upon dswifi and provides a much more high-level use of Wi-Fi functions to enable developers access to some of the more basic Wi-Fi functionality. Because of this, the PALib Wi-Fi functions are used in this project.

Since the NDS does not have a direct way to connect to a server or a database, a website is needed to take input through a string of Request variables in the URL and then store the data into a database. Unfortunately, the NDS does not support a built-in web browser. A third-party web browser made by Opera can be purchased; however, it is run through the NDS game slots, which would mean that no other program (like PBL) could be run at the same time. Because of this, the NDS can only interact with this website by sending a URL built of the data that is to be stored over the Wi-Fi. In order for the NDS to be able to do this, the program must initiate the Wi-Fi, connect to the Wi-Fi Connection service, and send the URL request over the Wi-Fi (see Code 4.19 for an example).

```
//if the Wi-Fi has not been initialized yet
if(WifiInitialized == false)
{
   PA_InitWifi();          //Run the PALib Init Wi-Fi to turn it on
   PA_ConnectWifiWFC();    //Connect to the Nintendo Wi-Fi Connection Service

   WifiInitialized = true; //Flip the flag so that Wi-Fi is on
}

//A variable to build the request URL to send
string request_text = "http://umhc-
stgprodw.umh.edu/IBLDS/Default.aspx?Username=Student&Question1=Test";

//Create a const char * that points to the URL
const char *WebSite = request_text.c_str();
//Create a char * that points to the URL
char * Address = const_cast<char*>(WebSite;)
//Create a buffer for the returned data
char *buffer2 = (char*)malloc(256*256);

//Call the PALib function that sends a URL and reads back data
PA_GetHTTP(buffer2, Address);
```

Code 4.19: This demonstrates a brief example of how to send data to a website over the NDS Wi-Fi.

DEVELOPMENT PROCESS Chapter 5

DEVELOPMENT ENVIRONMENT Section 5.1

The production of the prototype was a long and difficult process, mostly because of the lack of a true development environment. The homebrew programmer does not have access to expensive and useful IDEs and debugging tools for NDS programming. The IDE of choice for the average homebrew programmer is Visual C++ Express Edition 2005 which is freely available for download through Microsoft (15). There are tutorials that exist online which claim it is possible to add IntelliSense compatibility for libnds to any Microsoft Visual Studio product, although the instructions rarely work as well as the authors claim.

Standard C or C++ is used to build homebrew NDS games; however, the developer needs to keep efficiency in mind because of the slower speed processors inside the NDS and the amount of graphic computations the NDS processes. Because of this efficiency need, Object Oriented Programming (OOP), functions within functions, and recursion are to be kept to a minimum. In fact, on the GBA, because of the even slower processor, most homebrew developers decided to program only in C with no OOP and optimizing often used sub-routines

in assembly language. Fortunately, the NDS's hardware has advanced enough to allow for richer programming options to be used sparsely.

Visual C++ Express does not have libnds or other NDS homebrew libraries built-in, for that a developer requires devkitPro. DevkitPro (16) is a combination of libraries and toolchains for many homebrew developers including NDS, GBA, PSP, Nintendo GameCube, and more. DevkitPro also includes examples for development on various video game systems, MSYS (a Unix-like shell environment), gcc /g++ (an open source collection of compilers), installs path variables for development, enables the use of the "Make" commands in Windows, and installing many smaller tools that convert files to be used at compile. Further libraries like PALib can be installed with devkitPro for more development options.

The majority of homebrew developers use templates that are provided by devkitPro or PALib to build NDS projects. These templates use Make files to build many different files into a compiled final product. The Make command is not natively supported on Windows and is one of the many reasons to install devkitPro. By running a Make file, a developer can take standard program files (.c, .cpp, .h) from various folders, input various data files (images, audio, flat files), turn all of them into .obj files, and compile them into a final product of a .nds file (the NDS file type) with one simple command. Setting up a

Make file by hand is a rather difficult and extensive task; however the template Make files work very easily out of the box with little configuration, so most developers will never need to edit their Make files. There are also tutorials online to have Microsoft Visual Studio products compile NDS code so that make files and the gcc /g++ compiler are not necessary. However, these solutions usually leave out a lot options that running a full Make file can provide.

After programming and building a NDS game, a developer will need some way to test it. The easiest way to do this immediately is to use an emulation program. An emulator duplicates the functions of one system using a different system, so that a computer can behave in a way similar to an older computer or a piece of hardware, for instance the NDS. The most robust emulator currently available is No$gba (17), which emulates both GBA and NDS games. No$gba provides the most realistic emulation of the NDS hardware and is free to download. The developer of No$gba even offers a developer version of the emulator that contains special tools like a debugger, the ability to see what is currently loaded into OAM and backgrounds, and the contents of registers; however, these extra features need to be purchased directly through the developer of the software (18).

In order to test a homebrew game on the NDS hardware, a developer will need additional software and hardware. There are many

options for how to play homebrew software on the NDS (19). The solution that was used for the development of this project was to purchase a CycloDS Evolution game card (20). A microSD memory card containing the homebrew software is placed inside the CycloDS card and then the CycloDS card is placed inside the SD game slot of the NDS, known as SLOT-1. The CycloDS runs software directly in SLOT-1 without any other types of hardware, software, or cracks like some other SLOT-1 and SLOT-2 (GBA cartridge slot) solutions.

HIGH-LEVEL LIBRARIES Section 5.2

This project has produced the beginnings of what could be a high-level library that limits developer interaction with registers. Some of these examples can be seen throughout this paper in code such as the Background struct (Code 4.1), Background functions (Code 4.2-4.4), and graphical fade functions (Code 4.17). These functions are rather diverse and hard to narrow down to just one library. However, this project has produced a few smaller libraries that have a narrower range of utility. One of these libraries is the Questions library which contains a struct for questions that are asked in multiple choice quizzes and two functions that allow the asking and answering of questions.

```
typedef struct
{
   int QuestionNumber;    //The number of the current question
   char *QuestionText;    //The question text

   char *Answer1Text;    //First choice in the multiple choice window
   char *Answer2Text;    //Second choice in the multiple choice window
   char *Answer3Text;    //Third choice in the multiple choice window
   char *Answer4Text;    //Fourth choice in the multiple choice window

   int CorrectAnswer;    //Correct answer

   char *CorrectAnswerExplanation;        //Text explaining the correct answer
   char *IncorrectAnswerExplanation;    //Text explaining the incorrect answer
}Question;

void AskQuestion(int QuestionBoxLeft, int QuestionBoxRight, int QuestionBoxTop,
        int QuestionBoxBottom, Bg BgQuestion, Bg BgAnswers, bool Transp,
        bool Box, int QuestionNumber)
int AnswerQuestion(int QuestionNumber, Bg QuestionBg, Bg AnswerBg)
```

Code 5.1: This is the contents of Questions.h which contains the Question struct and the function definitions of two functions that ask and answer questions.

The Question struct allows for multiple choice questions that have question text, up to 4 answers, a correct answer, and text for when a user give a correct or an incorrect answer. The AskQuestion() function takes the window coordinate bounds of the box that will hold the question; the backgrounds the question and answers will be displayed on; a Boolean if there will be a bounding box around the question text; another Boolean if that bounding box will be transparent; and the QuestionNumber of the question to be asked. With this information the code will select the Question and load all of

the relevant data in the correct locations on the screen without the use of any registers, all that is required is the developer send the proper variables and define the data in the Questions array in Questions.cpp. The function AnswerQuestion() waits for an answer from the user using the same variables from AskQuestion() and then outputs whether the answer was correct or incorrect from the Questions array.

Another example of a library produced during this project that allows developers to not have to deal with registers is the Text library (contained in the Text.h and Text.cpp files), which outputs windows to the screen with formatted text and can contain optional bounding boxes. The Question library uses the Text library to format the boxes it outputs and the control of flow through the dialog by the user. A developer can implement the Question library without knowledge of the Text library and still have a functioning portion of a project. Of course the libraries that were made for this project are not as robust or extensive as the open source PALib library. If a developer wanted to truly be independent of dealing with registers, it could possibly be achieved through PALib.

However to program without registers completely seems rather problematic because of the depth of knowledge required of the hardware while programming would be necessary in order to know how to program advanced tasks like bit shifting and creating new

functions and features. Because of this level of expertise it is doubtful that someone with very little programming background could develop NDS games after setting up the developer's environment. Developing for the NDS might even require at least a rudimentary knowledge of C or C++ before beginning and a willingness to learn about more advanced features such as bit manipulation. NDS development cannot easily be picked up by a teacher that is just “computer savvy” even with PALib. However, a library similar to mine or PALib could be picked up by an experienced programmer or a small video game shop and be used to make NDS programs rather easily.

Figure 5.1: A demonstration of a multiple choice question being asked in the prototype.

Figure 5.2: The user has selected an answer and it is the correct answer in the prototype.

PROTOTYPE Section 5.3

The prototype that was built for this project did successfully connect to a website to store data in the database. Unfortunately, every time the program is run, it takes a rather long time to connect to Wi-Fi and then the database. However, every successive connection after the initial connection is so quick that the output message to the user that the program is saving is flashed across the

screen so fast that the user may miss it and just see the “Saved” message (see Figure 5.3).

The Graffiti text function works well enough, but it is prone to excessive errors and might not be ideal for major amounts of text entry. On the other hand, the keyboard functionality worked extremely well and would probably make a much better text entry system for the user than the Graffiti system. If the Graffiti system is desired from the developer it could be done better through a tighter tuned recognition function; however the sensitivity of the NDS's touch screen could be the problem.

Unfortunately this prototype did not have the capabilities of testing a chat like system between multiple NDSes. However, a chat system would be overly complex to develop and probably require so many database calls that it would have to be a program in and of itself and could not be built-in to another game or program like the PBL prototype. It is probably possible to make a chatting client for the NDS, but in the context of the PBL program it is probably not feasible.

Figure 5.3: Screen capture of the prototype saving to the database.

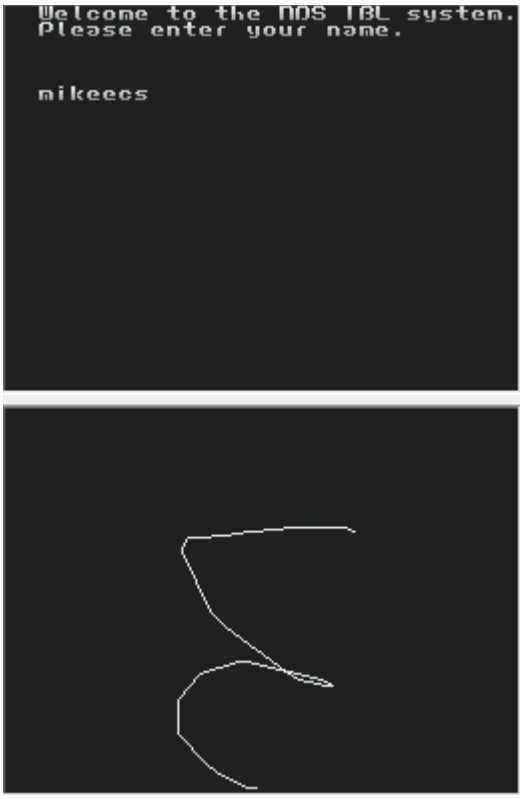

Figure 5.4: Screen capture of the prototype using the Graffiti system. Note how the use of the Graffiti "e" sometimes results in incorrect letters like "c" and "s".

Figure 5.5: A demonstration of the keyboard module working inside the prototype.

CONCLUSION Chapter 6

Problem-Based Learning (PBL) can be a very powerful tool for learning in the classroom. This tool promotes self-learning and motivation that may produce better students who can discover for themselves the love of independent learning and learning strategies that may be a benefit to their future learning. Unfortunately, today's K-12 classrooms are overstressed due to insufficient school budgets which could make PBL teaching a dangerous gamble for most. PBL may become even more problematic if taught in a standard "pen and paper" style because the differences between it and standard teaching are less pronounced and the benefits are less dramatic.

As a result, a more technological approach to PBL may make the learning experience better for the learners through the use of multimedia and game type learning. Most students in K-12 are accustomed to using video games, the Internet, and multimedia like video, audio, and interactives. If these tools can be harnessed to teach students in a more dynamic, interactive, and enjoyable way, not only would students learn more, they would be able to retain it better and have a zest to learn independently. Since a technological PBL model has these added benefits to offer over the "pen and paper" PBL model, more schools may be willing to embrace technology such as a

PBL tool, which could lead to PBL being used more frequently in the K-12 curriculum.

Although a technological approach could add value to the learner's experience, it also could add more cost to the school's budget. The NDS is a relatively less expensive alternative to provide a technological PBL solution. The cost of having a massively functioning website and a computer for every student to log onto the website would be enormous. However, the NDS is priced at about one fifth or even one tenth as much as a computer. The NDS is more affordable for schools to try to teach students in a new way. Not only that, but many children are accustomed to using the NDS or playing other video games. There are many products available today that are specifically built to emulate other video games while educating users, such as the line of products from LeapFrog Enterprises. A full curriculum built around the idea of combining PBL and video game play could be a very powerful educational tool.

Unfortunately, programming for the NDS is not an easy task. Building modules for classroom use will not be something that the average teacher can just easily accomplish. Furthermore, most major video game developers (like Nintendo itself) will probably not see enough profit in this venture to make it economically attractive. If a smaller, more independent third-party developer were to build these

games closely with a small school district though, this program could be successful. The video game developer could work with the school's administration and faculty to develop a curriculum and decide how the students should interact with different problems. This video game developer could then manufacture different modules for each problem for different grades or curricula and still stay in the bounds of what the school district believes is important for students of a particular grade to learn. The video game developer could even work with a small website or Internet company to provide online supplements to the curriculum that the NDS cannot handle, such as High Definition content or talking to live experts for case wrap-up.

Many factors would have to be considered, including the relative cost-benefit ratio. But, if the PBL system proves to be successful in the first school district, perhaps the video game company could expand to more schools nationwide. If a school district could work closely with a small video game developer, PBL could find a successful back door into our K-12 schools.

REFERENCES

1) Norman, G. & Schmidt, H. (1992). The psychological basis of problem-based learning: A review of the evidence. Academic Medicine 67(9), 557-565.

2) Savery, John R. "Overview of Problem-Based Learning: Dentitions and Distinctions." The Interdisciplinary Journal of Problem-Based Learning 1.1 (2006): 9-20.

3) Boud, D., & Feletti, G. (1997). The challenge of problem-based learning (2nd Ed.). London: Kogan Page.

4) Barrows, H. S., & Tamblyn, R. M. (1980). Problem-based learning: An approach to medical education. New York: Springer.

5) Duch, B. J., Groh, S. E., & Allen, D. E. (2001). Why problem-based learning? A case study of institutional change in undergraduate education. In B. Duch, S. Groh, & D. Allen (Eds.), the power of problem-based learning (pp. 3-11). Sterling, VA: Stylus.

6) Ertmer, P. A., & Simons, K. D. (2006). Jumping the PBL implementation hurdle: Supporting the efforts of K–12 teachers. Interdisciplinary Journal of Problem-based Learning, 1 (1), 40-54.

7) Stepien, W. J., & Gallagher, S. (1993). Problem-based learning: As authentic as it gets. Educational Leadership, 50(7), 25-28.

8) Kolodner, J. L., Camp, P. J., Crismond, D., Fasse, J. G., Holbrook, J., Puntambekar, S., & Ryan, M. (2003). Problem-based learning meets case-based reasoning in the middle school science classroom: Putting learning by design into practice. Journal of the Learning Sciences, 12, 495-547.

9) Wood, D., Bruner, J. S., & Ross, G. (1976). The role of tutoring in problem solving. Journal of Child Psychology and Psychiatry and Allied Disciplines, 17, 89-100.

10) Caplow, Julie, and Mark Ryan. "Teaching At Mizzou: a Guide for New Faculty, Graduate Instructors and Teaching Assistants." Teachandlearn.Missouri.Edu. 13 Mar. 2008. University of Missouri. 13 June 2008 <http://teachandlearn.missouri.edu/guide/chapters/pbl.htm>.

11) Korth, Martin. “Specifications.” GBATEK. 2007. 13 June 2008 <http://nocash.emubase.de/gbatek.htm#dstechnicaldata>.

12) “Libnds.” Wikipedia. 17 Apr. 2008. 13 June 2008 <http://en.wikipedia.org/wiki/Libnds>.

13) “DS Wi-Fi Bounty.” 9 July 2006. 13 June 2008 <http://sc.tri-bit.com/dswfb>.

14) “Nintendo DS Lite - ARM Powered Product.” www.arm.com. 22 June 2008 <http://www.arm.com/markets/home_solutions/armpp/11961

15) "Visual Studio 2005 Express Editions." Microsoft. 2008. 25 June 2008 <http://www.microsoft.com/express/2005/>.

16) "DevkitPro." DevkitPro. 24 June 2008. 25 June 2008 <http://www.devkitpro.org/>.

17) "No$Gba - Nocash Gameboy Advance / Nintendo Ds Emulator." 2008. 25 June 2008 <http://nocash.emubase.de/gba.htm>.

18) "No$Gba - Nocash Gameboy Advance / Nintendo Ds Debugger." 2008. 25 June 2008 <http://nocash.emubase.de/gba-dev.htm>.

19) "Nintendo DS Homebrew." Wikipedia. 24 June 2008. 25 June 2008 <http://en.wikipedia.org/wiki/Nintendo_DS_homebrew>.

20) "CycloDS...a World Far Beyond the Realm of Just Gameplay." CyclopsDS. 2008. 25 June 2008 <http://www.cyclopsds.com/cgi-bin/cyclods/engine.pl?page=products-cyclodsevolution>.

20) "Image:Dov DS MemoryMap.Png." Dev-Scene.Com. 17 Oct. 2006. 25 June 2008 <http://www.dev-scene.com/Image:Dov_DS_MemoryMap.png>.

21) "Image: Nds 2D Background Memory.Png." Dev-Scene.Com. 4 Jan. 2007. 1 July 2008 <http://www.dev-scene.com/Image:Nds_2D_background_memory.png>.

22) "Fetch.Php (GIF Image, 160x160 Pixels)." www.palib.info. 20 July 2008 <http://www.palib.info/wiki/lib/exe/fetch.php?cache=cache&media=http%3A%2F%2Fwww.palib.info%2FScreens%2FPAGraffiti.gif>. for PAGraffiti image (Figure 4.8)