SeeGOL

Shoyler's Extremely Experimental Graphical Operating Library

Available at: https://github.com/schuylermartin45/seegol/

Schuyler A. Martin
Computer Science, Rochester Institute of Technology
Rochester, NY
sam8050@rit.edu

***Abstract*—We introduce a 16-bit operating system, SeeGOL, to test the old systems programmer's adage that Intel processors "support everything written since the dawn of time". As the saying suggests, the Intel family of x86 processors are supposed to be compatible with any piece of software written since the advent IBM's first personal computer in the early 1980s. SeeGOL aims to test this claim by being a graphics-capable 16-bit OS that can be booted on any modern PC.**

Keywords— *Operating System, Systems Programming, VGA, 2D Computer Graphics, Real Mode, 16-Bit Intel Architecture, Open Source*

I. Introduction

SeeGOL, or Shoyler's Extremely Experimental Graphical Operating Library, is a 16-bit operating system with graphical capabilities, created as an educational endeavor. Although it is still theoretically possible to develop an operating system to target the original Intel 8086 architecture using modern tools, very few have pulled it off outside of emulating the architecture. Even fewer have attempted to boot such an OS on original IBM PC hardware. In fact, historically there have been only two commercial operating systems to ever target the original 8086: MS-DOS and IBM's OS/2. SeeGOL can be thought as a "toy" operating system that is built to demonstrate a point.

SeeGOL is primarily written in C and built using the modern GNU C Compiler (gcc) and related tools. Although it was originally targeted for IBM's PCjr from 1984, SeeGOL can only run on Intel chips made on or after the i386. This is due to the lack of support for targeting the 8086 in freely available C compilers.

The work presented here is provided for educational purposes. SeeGOL is open source, well documented, and includes a journal logging the problems encountered through the development process. We hope that SeeGOL might act as an inspiration for others interested in operating system and low-level graphics development.

II. Implementation

SeeGOL is written in C and x86 assembly. For easier development, SeeGOL is primarily written in C. x86 assembly is used only in situations where C cannot, such as the bootloader code and communicating with the hardware. To save development time and code size, SeeGOL is heavily reliant on BIOS interrupts to monitor and trigger hardware events.

A. Build Pipeline and Testing

SeeGOL is built using freely available open source tools available on Linux. This includes standard tools such as gcc, ld, as, dd, make, and makedepend. There are also some custom Bash, Python, and ImageMagick scripts used to compress and compile images for SeeGOL. SeeGOL is built as a 16-bit Intel x86 operating system using special gcc build directives. Unfortunately, given the difficulty of building for such an old platform, SeeGOL's build process is dependent on several non-standard features of gcc. These problems are described in further detail in Section III.A but for now it is important to note that SeeGOL will not boot on any true 16-bit x86 processor. SeeGOL can only boot into Real Mode on 32-bit and 64-bit x86 processors. Despite all that, SeeGOL is relatively easy to build.

A single Makefile in the top-level directory automates the entire build process. A linker script is also included in this directory to ensure that the assembled bootloader is placed at the correct memory address. C and x86 files are stored in under the src/ directory in subfolders for easier organization. The Makefile is aware of these subdirectories and compiles all C and x86 files in these directories into ELF object files, stored in the bin/ directory. Additional debugging output from compilation process is placed in the dbg_bin/ directory. The linker script then combines all of the object files into a single flat binary image. Using some additional make directives, the flat binary can be converted into a bootable image file that can then be dd'd again to a boot device or used in an emulator. At the time of writing SeeGOL has been successfully booted off of a USB flash drive and a physical hard drive. In theory it should be able to boot off of a floppy disk, but attempts at doing so have been unsuccessful.

SeeGOL was tested on three separate development environments: QEMU, the lab machines in the Distributed Systems Labs (DSL) at RIT, and another machine I personally

own. QEMU is a popular emulator used by Linux and OS developers and it gives a good representation of the actual hardware. However QEMU is far from perfect. The default BIOS implementation QEMU provides, for example, is often too forgiving. That is to say that there were many times that SeeGOL would boot in QEMU and not on the real hardware. The DSL machines all have Intel Core-2 Quad processors and my personal machine has an early i5 processor. These two machines are ideal testing candidates. These machines have different motherboards, processors, and BIOS implementations. There have been bugs encountered while developing SeeGOL where SeeGOL will run on one of the test environments and not the other(s). Ideally, SeeGOL should be able to boot off of any 32-bit x86 machine. Practically speaking this is dependent on the BIOS software put out by the motherboard manufacturer. In general, SeeGOL should be able to boot off of any x86 computer with a processor made after and including the i386.

B. Booting

SeeGOL's bootloader is simple and completely written in x86 assembly. It does the minimal amount of work required to make the jump to the kernel's main method. Once the jump has been made, the rest of SeeGOL has been implemented in C. The bootloader is loaded into main memory by the BIOS chip at memory address 0x7C00 and must be contained within 512 bytes, including a 2-byte bootloader signature at the end.

The primary goal of SeeGOL's bootloader is to load the rest of the operating system into main memory from the boot device. This is done using BIOS interrupt 13h which for legacy support treats the boot device as a floppy disk. This is done all at once with a single call to interrupt 13h that loads all sectors containing SeeGOL after the bootloader segment, starting at address 0x7E00.

The secondary goal of SeeGOL's bootloader is to prepare the system to make the jump to kernel code. The segmentation registers that are used to access memory in Real Mode are all set to 0x0000, with the exception of the stack segment register. The stack starts at the beginning of the bootloader (0x7C00) and grows towards zero. Section III.B has a more in-depth discussion about the memory limitations in SeeGOL. After the segment registers have been initialized and the OS has been loaded into main memory, the bootloader makes a jump to the kernel's main function. For debugging purposes, the bootloader also draws two characters to the screen: "B" for "booted" at the start of the bootloader and "L" for "loaded" before the jump to the kernel's main has been made.

C. Kernel Faculties

There is a separation between between kernel and user space in SeeGOL but the line is blurry compared to more practical operating systems. SeeGOL's kernel package (src/kern/) contains both hardware drivers for systems devices and some software utilities. The kernel package is also where SeeGOL defines custom types, debugging macros, and other constant values, common throughout the other packages.

In terms of hardware, SeeGOL's kernel package includes drivers for the Real Time Clock (RTC) and the Video Graphics Array (VGA) devices. Currently only VGA Mode 13h is supported in SeeGOL. The RTC is described in further detail in Section III.D and the VGA driver is described in Section II.D.

In terms of software, SeeGOL's kernel package comes equipped with a Random Number Generator (RNG) and a basic text-based input/output library, known as the Kernel I/O (KIO) library. The pseudo random number generator is implemented as a software-defined linear feedback shift register (LFSR). By default, the RNG is seeded with the current system time. The RNG can also be seeded with any integer, guaranteeing the same number sequence between runs.

The KIO library provides basic string manipulation functions, the ability to print text to the screen, and the ability to read basic keyboard input. KIO draws text using VGA's default text-only mode which operates at a character resolution of 80x25. If too much text is put on the screen, all KIO calls will result in automatically scrolling-up text to prevent the loss of new output. When a graphics mode is enabled, any text printed using the KIO's output functions will be stored in a temporary frame buffer. One can think of this feature as being similar to using TTY sessions in Linux. The KIO library also features an input prompt function that allows a user edit the text they have entered in. All of these advanced features came out of a need to build debugging tools for SeeGOL and eventually lead to the creation of SeeGOL's first user program, SeeSH, the shell used to access all other available programs in SeeGOL. SeeSH is further described in Section II.F.

D. Graphics Driver

The graphics driver code can be found in the kernel directory in src/kern/vga. Currently there is only one graphics mode supported in SeeGOL and that is VGA Mode 13h (often shortened to VGA13). Adding support for additional graphics modes is trivial. The vga.h header file includes a C-struct filled with function pointers. This enforces standard set of functions that all graphics drivers must implement in order to guarantee the driver will work with the graphics libraries. In other words, the graphics library provides methods to initialize a specific graphics mode and then calls the appropriate driver functions using a single VGA structure. The graphics libraries are explained in further detail in Section II.E.

SeeGOL's VGA13 driver was ported and improved from my previous work on the Bobby Senior Operating System (BSOS) [1]. The BSOS was a 32-bit OS so most of the changes in the driver came in the form of switching to use 16-bit integers. The VGA13 driver provides methods for initializing the VGA device mode, putting pixels to the screen,

and reading pixel values. Pixel values are written and read to the addresses reserved for video memory.

In Mode 13h, pixels are represented by a single byte. That byte acts as a key into a color palette table stored on the VGA device. This is similar to how the Color Graphics Array (CGA) graphics card operates on the PCjr. CGA is an early predecessor to VGA and both created by IBM. There is one key difference between the two that SeeGOL abuses which likely breaks backwards compatibility with the PCjr (outside of the build-system limitations discussed in Section III.A). The color palette with the CGA card is fixed in hardware and can be set to one of a handful of pre-defined color palettes. The color palette in VGA Mode 13h can be programmed by telling the VGA hardware to set an RGB value (6-bits per channel) for a specific entry in the table. This feature has been exploited by SeeGOL's improved VGA13 driver to allow the user to specify colors on the fly.

The VGA13 driver keeps its own look-up table of colors that have been requested to be drawn. If a color was found in the table, the look-up values is written to video memory. If it was not found, the driver sets the next available color in the table and replicates the change with the VGA controller. White and black are permanently reserved in the table, leaving the the table with 254 other colors that can programmed. If more than 254 colors are requested, the table wraps-around and begins to overwrite programmed colors. It is unlikely that there are enough unique colors used in SeeGOL for this to ever actually become a problem. The "correctness" of the programmed colors was tested against the Macbeth ColorChecker, a standard color chart used by photographers.

E. Graphics Libraries

SeeGOL comes equipped with two major graphics libraries, found under the src/gl/ directory. The first library is simply known as the gl_lib (the graphics library library...a name that happened by accident). This library provides the basic graphics capabilities for user programs. A user program can use this library to enter and exit a specific graphics mode, draw rectangles, draw lines, write text, and draw images to the screen. The latter three functions support simple scaled rendering, expanding a single pixel with a call to draw a multi-pixel rectangle. SeeGOL also includes its own original bit-mapped font, SeeFont (Shoyler's extremely experimental Font). SeeFont is exclusively used for rendering text in graphical environments.

As discussed in Section II.D, all of these functions make direct calls to the graphics driver that was configured when the graphics mode was initialized. The graphics library functions are independent of the hardware and offer more generic solutions to rendering requests. For example, the line draw functions implements Bresenham's midpoint line algorithm and is hardware agnostic. This will make it easier to add new graphics drivers to SeeGOL in the future.

The second graphics library, the Panes library, builds on top of the first one and provides SeeGOL with the ability to produce simple GUIs. Panes come are GUI windows that come in a variety of forms. Most panes look like the slides one would find in a modern-day slideshow program, being able to hold a title with text or an image or sometimes both. All panes are highly customizeable. For example, panes do not require a title text and can conditionally have a drop shadow. A user program can also set the look of a pane by setting the current pane theme. Most notably, the library also provides a "prompt" pane. This allows the user to select an item from a list of options with the arrow keys. The prompt pane is the primary GUI input method used in graphical SeeGOL programs. It also forms the basis of the visual program menu, discussed in Section II.F.

F. User Space & SeeSH

The user programs can all be found under the src/usr/ directory. SeeSH (Shoyler's extremely experimental SHell) was the first user program developed for SeeGOL. SeeSH is also the only program spawned by the kernel's run-time loop. In fact, all other SeeGOL programs must be run from SeeSH. SeeGOL lacks a file system so all user programs are baked into the OS. SeeGOL also lacks a scheduler so programs execute sequentially. Put another way, there are no system services or background tasks that are ever run on SeeGOL. There is only one program executing at a time, and each program is launched by calling that program's "main method". Effectively, SeeSH is the only system on SeeGOL that launches tasks.

SeeSH made debugging SeeGOL much easier as multiple test programs could run at the user's request. After SeeGOL gained a graphical input pane, SeeSH was further expanded to include a graphical mode to launch other graphical programs. This is the mode that SeeSH starts up when the machine is booted. The original SeeSH terminal mode can be launched from the visual menu. Every program built into SeeGOL can be launched from SeeSH.

All other programs follow a strict specification, enforced by a C-struct. This structure includes space for strings that provide a calling name for the program, a description of the program, and a usage message. The latter two strings are used in SeeSH's built-in help menu. The C-struct also includes a definition for a function pointer that points to the main method of the program. This main method takes the standard C parameters, an argument count and a list of string arguments. SeeSH provides these to the program after parsing the user input. Although not enforced by the program specification, every SeeSH program also includes an init function that sets initializes these parameters. SeeSH uses the init methods to "install" user programs to the system. Once this is done, the program can be executed via SeeSH.

III. Issues Encountered & Future Work

Building SeeGOL was no easy task. Outlined in this section are the biggest issues that had to be solved while developing the 16-bit operating system.

A. Targeting for the IBM PCjr: Modern Tools Lack 8086 Support

Few modern C compilers support compiling for the original 8086. Remember that all Intel x86 systems boot into Real Mode. Effectively, this means that the CPU believes it is a 16-bit computer to ensure backwards compatibility for legacy software. However, when Intel made the switch to a 32-bit architecture, they expanded Real Mode to allow for some 32-bit register instructions. This allows Real Mode to be significantly more flexible. Naturally, this makes life easier for C compiler developers. Very few individuals (such as myself) need to run code on a machine that pre-dates the i386 so the compiler developers decide do not support such features. Therefore I had to drop one of the original design goals of SeeGOL. I wanted SeeGOL to prove that the Intel architecture was truly backwards compatible to the dawn of the 8086 by both booting off of IBM's PCjr (8088 processor, 1984) and a modern i-series machine. Such is life.

Now that was the end of the story, up until recently. On April 1st, 2017 an email was sent on gcc's mailing list [2]. Andrew Jenner claims to have begun work on adding a new flag gcc flag (-m16) that ensures full compatibility with 8086 processors. Given the date of the email, this sounded like a joke, but Jenner later ensured that his work was no prank. Having this flag would not only allow SeeGOL to run on the PCjr, as originally intended, but it would also clean up the code base. Currently to tell gcc to build a file into Real Mode assembly each file has to include the .code16gcc at the top of the file. Unfortunately at the time of writing, Jenner's work is still a few months down the road from being finished and released, if it is ever released at all. Still, there is hope for the future.

B. Life before the A20 Gate

It is worth noting here that SeeGOL's flat binary image is limited to roughly 28kb in size by gcc. If the compiled code takes up any more space than that, the linking stage will fail. The linker will report an error message saying that it was unable to relocate a one of the object files due to a pre-defined size limitation for building 16-bit code. This like a relatively arbitrary limitation. The x86 architecture is a 16-bit architecture, meaning there should only be 216 = 64kb of addressable memory space. Yet Intel has provided a handful of segment registers that effectively expands each memory offset calculation by 4 bits, thus 220 = 1mb. Unfortunately, gcc's 16-bit compilation mode ignores these segment registers so we are back to only having 64kb of usable memory. The 28kb limitation is likely a safety mechanism to reserve some stack space for the program but I have been unable to find a definitive explanation for this discrepancy.

To combat the size limitation, some compromises had to be made in terms of SeeGOL's design. First, the Makefile was altered to include a number of size optimization flags to reduce the compiled code size according to suggestions on from pts.blog [3]. This was fairly tricky. Some of the suggested flags caused SeeGOL to fail to boot properly on some of the test environments. Second, as mentioned previously in Section II, SeeGOL is heavily dependent on using BIOS interrupts to shorten some of the hardware communication code. Third, simple functions that wrapped other functions to provide alternative parameters listings were replaced with macros. As ridiculous as it sounds, that ended up saving quite a bit of space. Aliasing such functions with macros removed a few kilobytes of unnecessary stack management instructions. Finally, as will be discussed in further detail in Section III.C, the image data built-in to SeeGOL was compressed to save additional space.

C. Image Storage & Compression

As mentioned previously, memory is at a premium in SeeGOL. In the interest of time, SeeGOL also lacks a file system. So to render images, SeeGOL stores compressed image data as part of the operating system. The compression scheme is heavily based on the X Pixel Map (XPM) file format. XPM files store image data as arrays of C-strings. Each character in the string represents a color whose RGB properties are stored at the top of the file in a look-up table. Each string represents a scanline, and the whole image can be built by the array of strings. In fact, XPM was designed as a file format that could be compiled into C programs for the X Windows System. SeeGOL uses a modified version of XPM, called Compressed XPM (CXPM) to reduce the amount of space that image files take up in the OS.

CXPM significantly reduces the amount of overhead required by the XPM by setting a handful of restrictions. At most 15 colors can be used in an image, including an option to set a pixel to be fully transparent. Limiting the color space this way means that each identifier in lookup table can be used as a fixed-length four-bit value. Instead of using strings, CXPM stores each scanline as an array of bytes. Each byte stores the color key for two pixels. CXPM further compresses the image using a run-length compression scheme.

If it is worth the memory overhead, CXPM will store repeated bytes in a 3-byte format. The first byte is a null marker. Zero is reserved in CXPM, hence why CXPM is limited to 15 colors despite having a 4-bit key. The next byte indicates the number of repetitions. The final byte stores the bit-packed color key representing two pixels. If there are more than 255 repetitions of a color, then the 3-byte pattern is repeated to prevent an overflows. The run length encoding makes a significant difference. Single color backgrounds can be represented in 3-bytes per scanline, such as the case with

Pink Floyd's *Dark Side of the Moon* album artwork.

As an added bonus, the run-length encoding actually makes image drawing faster. The VGA driver has a hardware-optimized rectangle draw function that significantly reduces the number of memory accesses required to draw solid rectangles on the screen. Knowing that there is a strip of pixels on a scanline that share a single color allows the draw image function to draw all of these pixels at once with the optimized driver call. A scanline is just a 1-pixel tall rectangle, after all.

At one point, there was another iteration of CXPM in the works but has since been abandoned. The newer scheme would improve upon the run-length compression by using two dimensions. This system would specify rectangular regions of the same color. A version of this was nearly working at one point but had significant rendering bugs. On top of that, it was unclear if the new CXPM files were significantly smaller than the run-length scheme. This work was abandoned in the interest of time. However the code was preserved as a separate branch in SeeGOL's git repository. The current CXPM specification is laid out in the top of the cxpm.py script found under the src/res/ directory.

The build process for image conversions is entirely automated by image file rules laid out in the top-level Makefile. Using ImageMagick, a Bash script will first convert an original image file to a color-quantized XPM file with a set maximum resolution threshold. This script ensures that we retain a relatively decent amount of quality in the image that is displayed by SeeGOL. ImageMagick also ensures that we have reduced the color space to be within CXPM's constraints. It is worth noting that as an unintended consequence of the color reduction process is that we get a better compression in CXPM. After the Bash script is finished, the XPM file is converted to a CXPM file by a Python script. Each CXPM file is then conditionally included into the SeeGOL project based on the macros set by the img_fids.h and img_tbl.h header files found under the src/gl/ directory. As the code segment of SeeGOL has grown, I have had to continually reduce the number of images that are compiled with the system. That being said, without the CXPM file format, there would be no way to include any images in SeeGOL.

D. Problems with the Programmable Interrupt Timer

Despite having written a driver for the Intel 8254 Programmable Interrupt Timer (PIT) for a 32-bit operating system, I had significant issues doing the same in SeeGOL. This all came down to registering the clock interrupt handler with the system. In Real Mode, one must register the interrupt service routines in the Interrupt Vector Table (IVT). By default the IVT is located at the start of memory and holds the memory locations of 256 interrupts. When I tried to manually add the the handler for the PIT, I would unintentionally disturb the BIOS interrupt routines. The BIOS routines are loaded into memory at boot time by the BIOS chip. For some reason, modifying the table would always screw something else up. Put the handler address in one spot, the keyboard would stop working. Put it somewhere else, the OS would crash. I could not find any clear example on how to register an interrupt handler in Real Mode.

After a week of working on this, I gave up and decided to poll the Real Time Clock (RTC) and use that as the timing mechanism for SeeGOL. The RTC is the clock powered by a motherboard's onboard battery and provides the current system time and date. The RTC has some clear advantages and disadvantages over using the PIT. The RTC is initialized at boot time by the BIOS and retrieving the time and date just requires communicating with some port registers. Unfortunately the RTC lacks sub-second precision. As a consequence of that, all animations in SeeGOL are currently limited to one frame per second. This is fine for the analog clock program, but is rather limiting otherwise.

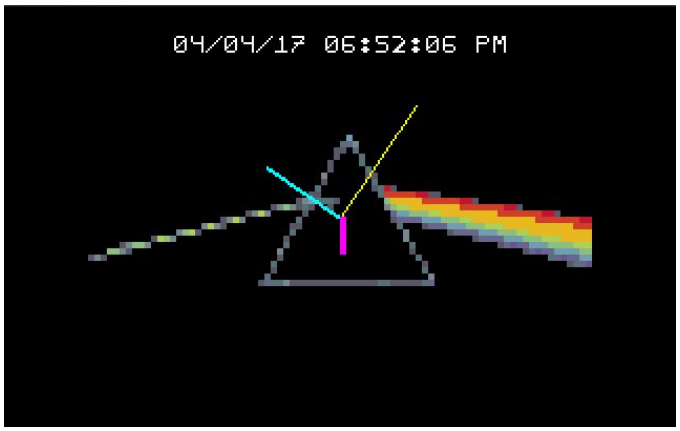
IV. Screenshots

What follows are some screenshots taken from SeeGOL. All of these screenshots are taken from QEMU. More screenshots are available on the GitHub repository.

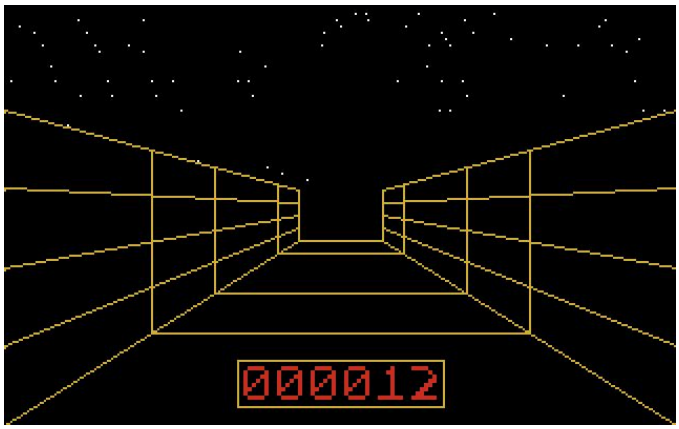
Screenshot 1: SeeGOL's visual SeeSH-backed program menu, written using the pane library. SeeGOL's graphics library comes equipped with a custom bit-mapped font, SeeFont. This is the first interface the user sees after the SeeGOL splash screen.

Screenshot 2: SeeSH, SeeGOL's provided text-only shell.

Screenshot 3: HSC test pattern, shows all the major features of the graphics library at once. The color pattern shown comes from the Macbeth ColorChecker.



Screenshot 4: Analog clock program, which uses Pink Floyd's *Dark Side of the Moon* album artwork. The image is stored in memory as a 50x50 pixel image with a transparent background. SeeGOL's graphics library is scaling-up the image for a better user experience.



Screenshot 5: Animated "trench run" program. These plans are NOT in the main computer. The stars above the trench are procedurally generated with SeeGOL's software-defined random number generator.

Screenshot 6: Pink Floyd's *Wish You Were Here* album artwork drawn in SeeGOL. Unfortunately this render had to be removed from the final OS because it consumed too much of the available code space.

V. Conclusion

SeeGOL is a working 16-bit operating system with significant graphics functionality. Although some sacrifices have been made due to the limitations of gcc, SeeGOL still shows that a Real Mode operating system can be made with freely available modern tools. As a well documented open source project, SeeGOL is bound to be useful example for other hobbyists and beginners looking to write an operating system.

VI. Acknowledgments

[1] Prof. Warren Carithers - Advisor
Warren, taught me almost everything I know about Systems Programming and Computer Graphics. Without him, none of this would be possible.

[2] Prof. Sean Strout - Mentor
Sean is a close friend of mine and initially sparked a lot of my

interest in becoming a C wizard.

[3] Prof. Thomas Kinsman - Mentor

Thomas has taught me how to think creatively with visual problems.

VII. References

[1] Martin, Schuyler. "Project Gallery." *Schuyler "Shoyler" Martin*. Shoyler.com, May 2016. Web. 19 Apr. 2017. <http://shoyler.com/html/projects.html#bsos>.

[2] Jenner, Andrew. "[PATCH 0/9] New Back End Ia16: 16-bit Intel X86." *[PATCH 0/9] New Back End Ia16: 16-bit Intel X86*. Gnu.org, 1 Apr. 2017. Web. 18 Apr. 2017. <https://gcc.gnu.org/ml/gcc-patches/2017-04/msg00009.html>.

[3] "How to Make Smaller C and C++ Binaries." Blog post. *Pts.blog*. N.p., 25 Dec. 2013. Web. <http://ptspts.blogspot.com/2013/12/how-to-make-smaller-c-and-c-binaries.html>.

[4] For a complete listing of web resources that were found to be useful in developing this project, visit the online GitHub repository at https://github.com/schuylermartin45/seegol/blob/master/docs/links.txt