Porting DOOM (1993) to the Apple Watch

Joey Tarnowski

University of North Georgia  
Department of Computer Science and Information Systems  
82 College Circle, Dahlonega, GA 30597

jmtarn3727@ung.edu

A screen shot of a video game

Description automatically generated

Figure 1. The original DOOM source code running on an Apple Watch Series 10 simulator

**ABSTRACT**

In this paper, I describe a method of porting the 1993 first-person shooting video game DOOM to WatchOS. The port uses the doomgeneric codebase to simplify the process of porting, so the port only required 5 functions: an initialization function, a rendering function, a controller function, and a sleep and a “get time since start” function. Since the frontend for WatchOS applications must be written in Swift and Swift does not support directly integrating C code, Objective-C is used as a “bridge” for the C-based DOOM source code to the frontend and for the implementation of the necessary doomgeneric functions.

**General Terms**

Reverse engineering, embedded systems

**Keywords**

Software reverse engineering, porting, embedded software, DOOM Engine, WatchOS, input handling, framebuffer rendering

**1. INTRODUCTION**

This paper details the process of creating a port of DOOM for WatchOS on the Apple Watch, in which Swift is used to create the frontend control scheme management and load DOOM’s exposed framebuffer into a SpriteKit and Objective-C is used to make the DOOM Engine and Swift interoperable.

The process began with choosing the version of DOOM to modify and port. The doomgeneric repository was the most appealing version for this project as it only requires creating 5 new functions for the DOOM source code to properly function: DG\_Init, DG\_DrawFrame, DG\_SleepMs, \_GetTicksMs, and DG\_GetKey [1]. These functions were created primarily in Objective-C, although DG\_DrawFrame was implemented directly into the doomgeneric.c file since it is never called from the WatchOS Swift code.

The Swift frontend could then access DOOM’s exposed framebuffer to load it into a SpriteKit, and then use a combination of position-based touch screen and digital crown input to control the game.

**2. RELATED WORK**

The DOOM source code has been unofficially adapted to run on other hardware, known colloquially as “source ports”, dozens of times across many different platforms [2]. However, there are limited instances of DOOM being ported to an Apple watch, with the only true “source ports” I found being a 2015 port by Facebook developers with no source code available [3] and a 2023 port by Tanner Stokes [4]. Stokes’s brief documentation on the challenges was helpful in preparing for some potential issues that may arise, although his implementation used SDL libraries not used by my approach and we differed significantly in the ways the controller scheme was implemented and the ways in which the framebuffer was fed into the SKTexture. Other approaches have been taken to run DOOM on WatchOS, but these approaches typically rely on DOS emulation rather than a “true” source port [5].

**3. APPROACH**

**3.1. Project Structure**

The starting point of developing the project structure was the two main “mandatory” components, the DOOM source code (or DOOM engine) and the Swift frontend, which had to be used since nearly all WatchOS app development tools use Swift as the primary language. I had to then develop a “bridge” in Objective-C to make the Swift frontend and the DOOM Engine interoperable, since Swift is unable to natively run or call C code directly. To accomplish this as shown in Figure 1, the interoperability code was broken up into two different parts: doomgeneric\_bridge and doomgeneric\_watchos. doomgeneric\_bridge was designed to allow Swift access to small parts of the DOOM engine directly, including the framebuffer, the variable determining if a menu is open, the path to the DOOM .wad file, and the initial call to the DOOM Engine’s start function (doomgeneric\_Create). doomgeneric\_watchos was responsible for implementing the core system-specific calls required by doomgeneric, which will be discussed in the following section, essentially allowing the DOOM Engine to access system-specific information from the frontend.

A diagram of a bridge

Description automatically generated

**Fig. 1: Diagram of Swift to DOOM Engine Data Flow**

**3.2. Doomgeneric Function Implementation**

**3.2.1. Initialization**

The first function to implement from doomgeneric is the DG\_Init function. The purpose of this function in the DOOM Engine is to do any additional work needed for a system during startup. For WatchOS, this function is unnecessary since doomgeneric\_bridge’s startup function accomplishes the only system-specific task (giving the DOOM engine the path to the .wad file).

**3.2.2. Timekeeping**

The “timekeeping” functions required by doomgeneric are the DG\_GetTicksMs and DG\_SleepMs functions. DG\_GetTicksMs stored the system the time at program start, and when called would take the current time and subtract it from the stored initial time, and DG\_SleepMs, which makes a system call to sleep for 1000ms.

**3.2.3. Rendering**

While doomgeneric only requires implementation of the DG\_DrawFrame function, getting each frame to render on the Apple Watch took substantially more frontend work than the previous functions. DOOM uses a framebuffer for rendering, where the RGBA values of each pixel is calculated by files with the prefix r\_ (rendering), and then stored in DG\_ScreenBuffer object. The purpose of DG\_DrawFrame is to take the initial RGBA values, apply the proper palette, and put the RGBA values into the correct color channels. Since DG\_DrawFrame is only called by the DOOM Engine and the Swift frontend only ever uses the finished framebuffer for rendering, I chose to put DG\_DrawFrame directly into the main doomgeneric.c file instead of either of the the Objective-C interop files. The primary modification that had to be made to the rendering pipeline during the DrawFrame process was changing the default color channels, as DOOM calculates the color values using a BGRA format instead of Swift’s expected RGBA format.

Since DG\_DrawFrame was inserted directly into the DOOM Engine, the only interop code needed for rendering was a function to get the DG\_ScreenBuffer object.

To render the framebuffer with Swift, a SpriteKit Scene was created, and the framebuffer was loaded as an SKTexture and added as the only child sprite of the SKScene. As shown in Figure 2, initially loading the fraamebuffer into the Scene showed the game was being rendered in the wrong scale, with an incorrect anchor point, with an inverted y-axis, and with the RGB channels set incorrectly (before modifying DG\_DrawFrame to correctly set the color channels). A screenshot of a video game

Description automatically generated

**Fig. 2: Offset, scale, and color rendering glitches compared to the final render (right)**

The scaling was solved by using using doomgeneric\_bridge to directly retrieve the DOOM Engine’s screen width and height variables, which were modified to fit the dimensions of an Apple Watch Screen. The y-axis inversion and sprite centering issues could be solved using basic manipulation of the sprite’s anchor point and y scale. A vertical padding factor of 0.8 was applied as well, after testing showed that the game taking up the full height of the screen cut off parts of the game’s UI and upper left text due to the Apple Watch’s rounded corners.

**3.2.4. Controlling**

The DOOM Engine, initially made for use on PC (specifically MS-DOS), uses keyboard inputs as the standard input method for controlling the game. Doomgeneric requires the implementation of a DG\_GetKey function, which is called by the DOOM Engine to return if a key is true (pressed) or false (not pressed). This was implemented in my version by creating a key queue that stores the ASCII integer values of any inputs with a boolean value corresponding to their pressed status, along with a SetKey function that allowed the Swift frontend to set these values.

While prior implementations of DOOM in WatchOS used exclusively touch controls, I found 2 problems with using the touchscreen as the sole control method. The first issue is basic comfort/usability, as the small size of the touchscreen means fluid control of movement and character rotation and/or firing is difficult, and the user’s fingers can easily obstruct large parts of the screen. The second issue is character control, as DOOM’s gameplay frequently has the player moving forward while simultaneously rotating left/right or firing their weapon, which is impossible without adding more complex UITouch-based controls and would still be difficult logistically given the small area allocated to each control even with a UITouch-based control scheme.

To fix these issues, I created a control scheme which used the Apple Watch’s digital crown for forward and backward movement and used the touchscreen for left/right rotation and other player actions, as seen in Figure 3. This solved many of the core issues with other control schemes by allowing the user to set the watch’s dial for intuitive-feeling continuous character movement without needing continuous inputs, while also freeing up more of the screen for other important inputs. The ability to move and turn/shoot was also what necessitated the implementation of a key queue instead of a single variable tracking the pressed key, since this control scheme was built with the intention of allowing multiple simultaneous inputs. The digital crown-based movement had the additional advantage of allowing the player to access running in-game by setting a high threshold of digital crown rotation where the movement would be set from walk to run.

A screenshot of a device

Description automatically generated

**Fig. 3: WatchOS DOOM control scheme**

The in-game controls were mostly straightforward to implement, as the pressed “keys” just had to be set when being touched (in the case of touchscreen controls) or when the digital crown was set within a movement threshold. However, menu navigation posed a problem both because the menu expected a single press and release of a key to go up/down instead of the continual input from the digital crown, and because the menu navigation keys were distinct from the normal in-game action keys (Fire/action vs back/enter). To solve this, a variable was identified in the DOOM Engine that shows if there is a menu active or not, and a function was added to doomgeneric\_bridge to get that variable for the Swift frontend. Then, the Swift controller file would change the action keys based on the menu active variable, and any crown movement within a menu would wait until a certain up/down threshold was crossed, send a single up/down input, and then reset the digital crown rotation value, simulating a single up/down key press.

**4. RESULTS AND CONCLUSION**

After implementing all the above features, DOOM was able to successfully run on an Apple Watch. It was initially tested in an Apple Watch SE (2nd Generation) simulator running on XCode and once development was complete, the DOOM source port was able to be successfully installed and run on a physical Apple Watch SE (Figure 4). The game ran smoothly and was able to run at a full framerate during normal gameplay [6].

A person's hand holding a smart watch

Description automatically generated

**Fig. 4: DOOM running on an Apple Watch SE**

This project was an interesting deep-dive for me into the DOOM source code and the process of making an Apple Watch app, neither of which I had any prior experience with. Having to write the frontend for controls/rendering in Swift also presented an interesting challenge, as it required that I not only implement the basic necessary features for doomgeneric but also figure out how to make the C-based DOOM Engine and my Swift code interoperable.

The full repository for this project is now available publicly on Github [7].

**5. FUTURE WORK**

There are several improvements that could still be made on the Apple Watch DOOM source port. The most obvious improvement is implementing sound to the game, which was not done because the audio APIs used by doomgeneric (such as SDL\_mixer) are currently not supported by WatchOS. However, WatchOS does support AVAudioPlayer, which can be used to play audio files. It should be possible to convert all game sound into different .wav or .mp3 files that could then be played by AVAudioPlayer any time the DOOM Engine attempts to play a sound, but the difficulty and scope of such an implementation was beyond the scope of this project.

As briefly mentioned in the controls section, UITouch could also be implemented to create a more sophisticated control scheme that allows for 3+ key presses simultaneously. Other controller methods could also be explored, such as using a connected iPhone as a controller.

Lastly, the doomgeneric code does have support for online multiplayer, and future work may explore the possibility of adding support for multiple watches to connect and play, although the feasibility of this was not researched during this project.

**6. REFERENCES**

1. ozkl, “doomgeneric” *GitHub*, 2019. https://github.com/ozkl/doomgeneric/tree/613f870b6fa83ede448a247de5a2571092fa729d
2. “Doomworld -- Source Ports,” *Doomworld.com*, 2025. https://www.doomworld.com/classicdoom/ports/
3. Lior Tubi, “Doom on the Apple Watch - watchOS2,” *YouTube*, Oct. 01, 2015. https://www.youtube.com/watch?v=KXhf\_SpO6wA
4. Tanner Stokes, “DOOM on the Apple Watch”, 2023. https://www.tannr.com/2023/07/09/doom-on-the-apple-watch/
5. “The answer is finally yes. Yes it can.,” *Reddit.com*, 2024. https://www.reddit.com/r/AppleWatch/comments/1dlyka0/the\_answer\_is\_finally\_yes\_yes\_it\_can/#lightbox
6. Joey Tarnowski, “DOOMWatch E1M1,” *YouTube*, Apr. 23, 2025. https://www.youtube.com/watch?v=nkq0mLkQVxY
7. joeytarnowski, “GitHub - joeytarnowski/doomwatch: Final project for Advanced Reverse Engineering. Porting DOOM to WatchOS,” *GitHub*, 2025. https://github.com/joeytarnowski/doomwatch