Spring 2023 RTOS Project

Wolfenstein Assault 2049

Change log:

- Version 1: Initial release
- Version 2: Display and Physics Simplifications, and explanation of Platform Force Indication added

March 9, 2023

Authored by: Jon Haines

The naming of the project is in memoriam of:

- Pioneer video game programmer Silas S. Warner (c.1950-2004), who authored the original Wolfenstein game for Muse Software for the Apple II in 1981.
- Science Fiction writer Philip K. Dick (1928-1982), who authored Do Androids Dream of Electric Sheep the year after I was born. He has written many other books that you might recognize (Man in the High Castle, A Scanner Darkly, The Minority Report, Paycheck, Radio Free Albemuth). Do Androids Dream of Electric Sheep was produced into film in 1982 as Blade Runner—and was the first movie that I recall seeing gliding flying cars that looked sci-fi-ish. (Chitty-Chitty Bang Bang and a Ford Pinto with a wing attached to its roof (James Bond) don't count!) After his death, a sequel of the movie was made with the suffix "2049".

Both iconic titles are poster-children for managing one's intellectual property. Muse Software went bankrupt in 1985 and didn't maintain the trademark name—so it was taken over by Id Software for very profitable marketing of later refreshes, and BR2049 was delayed by 18 years due to licensing issues.

Crack open the fortress, Free the prisoners!

> *We have found the legendary Wolfenstein Castle, where prisoners are being held on a fortified cliff top. Our advanced artillery unit has assembled a cutting-edge firing platform for you (that includes an on-demand force-field shield and an anti-gravity sled) with a slightly-underpowered, fixed-elevation rail gun. Unfortunately, everything was a trade-off! We are supplying you with a virtually unlimited ammo supply and fuel for the generator, but could only acquire an underpowered generator and fewer capacitors than we'd have liked. Either hit the castle just once (we don't want collateral damage once it's been breached!) or the space just below the castle enough to crack the foundation to provide escape, before the defenders destroy your platform with hand-thrown satchel charges.*
>
> *Good luck, noble defenders! I'm just going to back up a bit here to give you room to work… Don't mind me… I've just got some really important work to do back at our heavily-armored base…*

Basic idea:

You will model the physics for a couple simple mechanics problems (parabolic projectiles, and a linear-motion platform), with some "environmental/configuration" inputs that will affect the physics, general purpose inputs from our SDK affecting the simulation, and with the current physical state periodically updated to the LCD and LEDs.

One of the great advantages to our simulating physics is that we can have ideal conditions (and thereby avoid "real physics" when it would be very difficult (e.g. friction that complicates things); instant response to inputs; configurable mass, forcing capability, etc).

The basic challenge:

> Using the slider, apply a force to the platform to move it left or right in an attempt to position your firing platform where you want to shoot from, or to avoid thrown satchel charges.

Use the left button to instantaneously discharge the force field shield, using energy from the capacitors. As long as the satchel charge is close enough to any part of the sled AND the required energy is available at that instant, the satchel will be harmlessly destroyed.

Use the right button to indicate how powerful a rail gun discharge you'd like, as long as the capacitors can provide it (start at zero when the button is pressed, and continue increasing the energy until the button is either released, or until the maximum energy is indicated). Fire the rail gun with the maximum of the available capacitive energy or the indicated shot energy when the button is released. Rail gun ammunition will not bounce at all. Upon first impact, it takes a chunk out of the cliff face, or sinks into the mud that the rail gun is hovering above. Beware though of weak shots that might hit the platform!

Track how much capacitive energy is available.

The construction of the platform allows it to bounce off of the canyon walls (100% elastic, as long as it is moving slowly enough; else it is destroyed).

The satchel charges will bounce (100% elastic) off the canyon walls, but will detonate automatically when hitting any horizontal surface (the sled or ground).

Choices, and bells-and-whistles left to your discretion:

1. Choices about game priorities when energy is not sufficient for shields and cannon, e.g.
   - Will you force an under-powered shot to retain a shield charge, once the shield requirement is buffered?
   - Will you allow the shot even if the capacitors will then be insufficient for the shield?
2. Graphical enhancements that assist in play, e.g.
   - A status bar showing the capacitive charge, with other bars shown nearby to illustrate how much charge would be needed for your indicated rail gun power level, and for a force field pulse
   - A status bar showing the magnitude of the sled's velocity, with an indicator of the speed at which a canyon wall collision would be catastrophic
3. Allow modification of configurable data after defaults are loaded via some sort of menu
4. Choose how you will represent damage to the castle, escaping prisoners, exploding satchels, shield operation, etc.
5. Possible end-game reporting, such as:
   - Victory(castle hit)/Victory(escape complete)/Defeated(by satchel)/Defeated(by collateral damage)
   - Number of satchels thrown
   - Number of shield activations
   - Number of useful shield activations
   - Number of rail gun shots

Baseline project expectations:

- Physics gets serviced every $\tau_{Physics}$ in its own task. This task will receive information about button changes (both presses AND releases will need to generate unique messages for one of the buttons, while the other will only need presses to be noted) and CapSense position (or "no position" when finger is not on it), and will communicate information that will affect the LCD and LEDs to a display task. (Part of your work in a later week on your project may be to push up the physics update frequency to find out when you no longer have uniformly-periodic updates to physics)
  - In each time increment, you will:
    - Apply force to the platform (possibly zero) and update its horizontal velocity based on the resulting acceleration, possibly reverse its horizontal velocity direction if it bounces off a canyon

wall, possibly adjust its horizontal velocity for the projected impulse imparted by a rail gun shot, and update its horizontal position
        - Update the satchel charge's path based on bounces (which reverse the direction of its X-velocity) and Earth-normal gravity.
- CapSense Slider will be sampled every τ_{Slider}
    - Commit to a number of positions (4 may have some benefit due to better discrimination), and decide how the positions map to force values within your configured limits (finger off the CapSense means no force applied to the platform)
    - Choose and justify your value of τ_{Slider}.
- Pushbutton controls
    - Configure the General Purpose Inputs to appropriately generate interrupts only when your design needs them. Send information to the Physics task as soon as these transitions are noticed.
- Left LED
    - When sufficient damage has been caused to the castle or its foundation, blink on/off with a 50% duty cycle at 1Hz to indicate that evacuation is under way. Once evacuation is complete, continuously light the LED.
- Right LED
    - Pulse width modulated to show (via human-perceived brightness) the current force magnitude, as a % of MAX_FORCE.
- Display Task will be serviced every $\tau_{Display}$, at which point it should process any information necessary to update of the LCD. LED controls should be updated as well, but you may have other OS mechanisms performing the low-level control of the LEDs—they are not required to be updated by the Display Task.
- Graphics shall show satchel positions within the canyon (bounded by the cliff on the left, and the screen edge on the right), the position of the platform on the bottom of the canyon, and any rail gun shots in flight. Only draw double lines on the left and a single line on the right edges to show the cliff and canyon, and draw only the platform at the bottom of the canyon. The picture should update every $\tau_{Display}$ (As with the physics update, you may be finding out how fast you can push your solution in later weeks.) Figure out the pixel geometry (NxM pixels) of your SDK's LCD screen from documentation and make the canyon width (including the cliff width) and depth scale to it (see below) assuming that the pixels are physically square. Place the origin in the center of the floor.
- Satchel throwing: Use a uniform random number generator to determine where on the canyon floor the satchel will impact. The satchel will be thrown with the cm/s horizontal velocity (and zero vertical velocity) to hit this point. You must, at a minimum support the "AlwaysOne" satchel method. If you do not support others and they are indicated by the configuration data, your code must detect this mal-configuration at runtime and indicate it. Each of the methods is described as follows:
    - **AlwaysOne**: There will always be one in flight. As soon as it is resolved, another is immediately thrown.
    - **MaxInFlight**: Once the first set get thrown (spaced out in time by τ_{Throw}), every time one gets destroyed or hits the ground, another is immediately thrown.
    - **PeriodicThrowTime**: Regardless of how many are in-flight, new satchels will be thrown on a periodic interval, τ_{Throw}. Therefore, depending on CanyonSize, the number will grow from zero to approximately a maximum, with random variation.
- Configurable data to drive the game, with clear and appropriate units noted in comments for each 32b quantity, as shown below. (Your data structure must have space set aside for each of these items in specified groupings (i.e. data structures of data structures), even if you don't implement anything for a particular datum (e.g. StachelCharges has some elements that you may not use). Your code should check for settings with respect to your implementation and fail if values are not supported). The suggested default values for these (shown

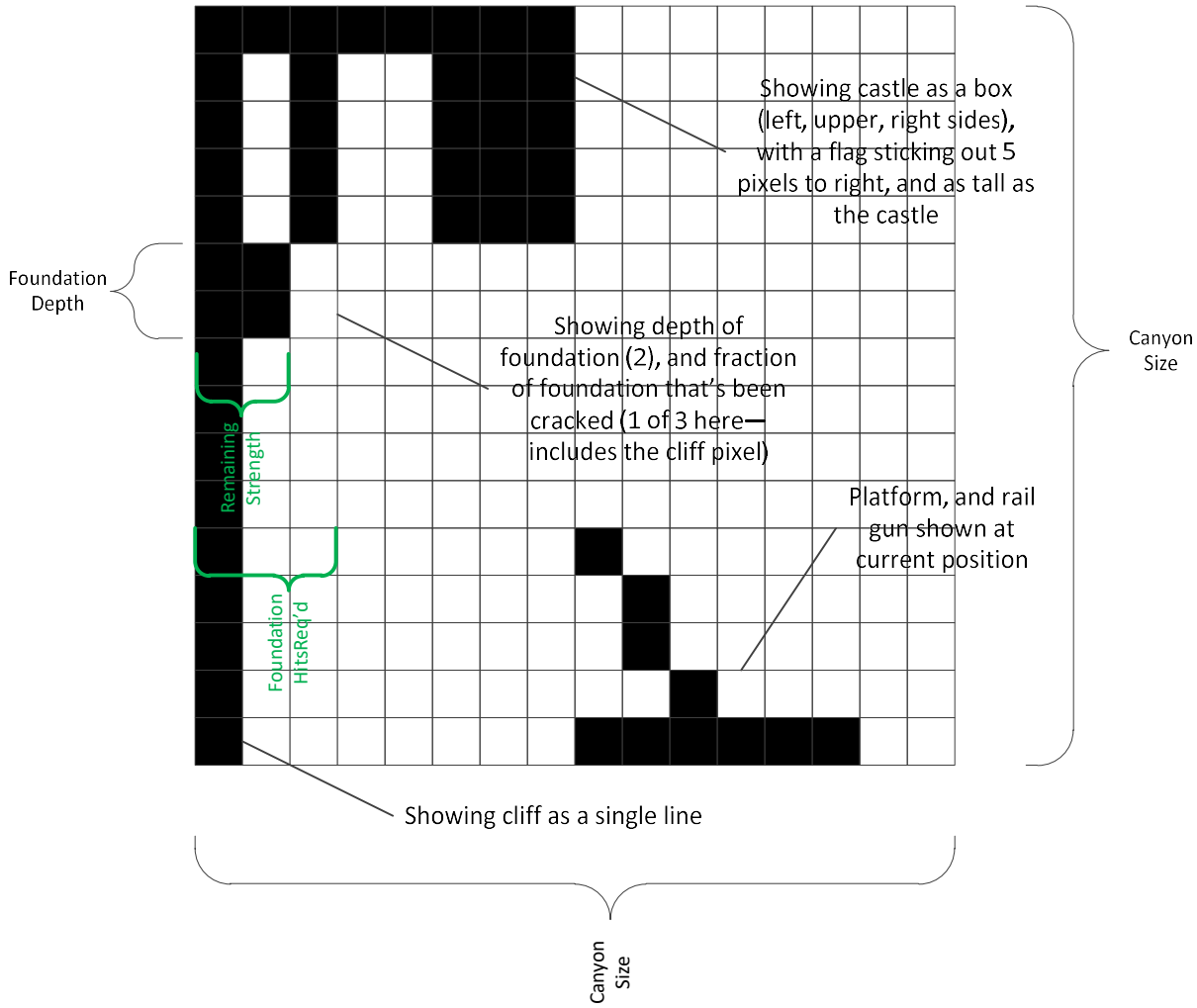
below, after "=") are merely SWAGs. Be prepared to explain your chosen values at your demo. At Demo time, expect to be able to substitute in numbers supplied by instructional staff.

- Data Structure Version = 1 (for the data structure shown below)
- TauPhysics [ms] = 50
- TauDisplay [ms] = 150
- TauSlider [ms] = 100
- CanyonSize [cm] = 100000
- Wolfenstein
  - CastleHeight[cm] = 5000
  - FoundationHitsRequired [-] = 2
  - FoundationDepth[cm] = 5000
- SatchelCharges
  - LimitingMethod [enum: AlwaysOne=0, MaxInFlight=1, PeriodicThrowTime=2] = 0
  - DisplayDiameter [pixels] = 10
  - Union (usage based on LimitingMethod):
    - N/A, TauThrow, TauThrow
  - Union (usage based on LimitingMethod):
    - N/A, MaxNumInFlight, N/A
- Platform
  - MaxForce [N] = 20000000
  - Mass [kg] = 100
  - Length [cm] = 10000
  - MaxPlatformBounceSpeed [cm/s] = 50000
- Shield
  - EffectiveRange[cm] = 15000
  - ActivationEnergy [kJ] = 30000
- RailGun
  - ElevationAngle [mrad] = 800
  - ShotMass[kg] = 50
  - ShotDisplayDiameter[pixels] = 5
- Generator
  - EnergyStorage [kJ] = 50000
  - Power [kW] = 20000

Display Simplifications:

- The instructions about how to draw the project on the screen were actually intended to make it easier for you than it might be otherwise. You can pretend that the cliff and castle are x-width of zero, and truly exist at the left edge of the left-most pixels. Therefore, you don't have to figure out where your rail gun shots intersect with the cliff at some distance from the screen edge, but rather you can simply find out where the intersect with the edge of the screen to find out if they hit the castle, foundation, or neither.
- You'll note that the configuration data describes how big to draw the satchel charges and rail gun shots on the screen, but not their physical sizes. This is because it's easier (somewhat for display, but especially for physics) if you assume they are zero-sized, but get drawn so that they can be seen.
- Don't worry about making the canyon go from middle-of-leftmost-pixel to middle-of-rightmost-pixel. You can simply divide the CanyonSize by the number of pixels in a direction to determine the dimensional size of a pixel.
- There is no requirement about how you draw the castle, foundation, or rail gun. My example below is intended only to help the player see what counts as castle or foundation, and how much foundation damage has been

caused. (OMITTED from picture below, but required for your project: the right edge pixels must also be "drawn" to help the player see where their platform will bounce.)



The number of vertical pixels that are used to draw the castle would be:

- (NumPixels · Wolfenstein.CastleHeight/CanyonSize)

The number of vertical pixels consumed by the castle plus foundation would be:

- (NumPixels· (Wolfenstein.CastleHeight+Wolfenstein.FoundationDepth))

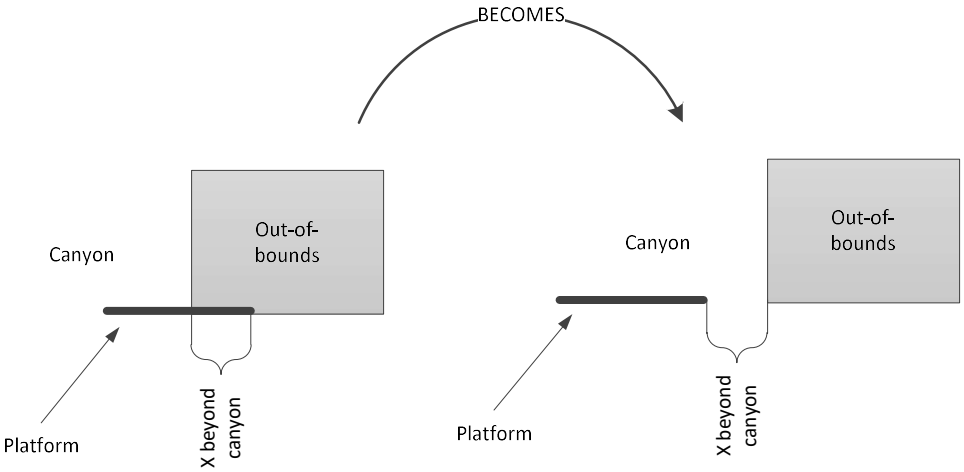
I've shown the rail gun as a line from the middle of the platform with an arbitrary length (4 pixels, or about ¼ of NumPixels in this example) at the elevation angle (about 65% here) . You could anchor it there or at either end of the platform—your choice.

Physics Simplifications:

- You COULD find the equation of the parabola followed by each rail gun shot or satchel charge to find out where they will intersect the left edge or bottom respectively, compute the time at which they will reach that point and update between $\tau_{Physics}$ physics updates to see if the platform is hit, but you can also try a MUCH simpler technique.
- Simply assume that the $\tau_{Physics}$ is small enough that nothing important happens between them, aside from a couple simple edge cases, allowing a whole lot of simplification. Don't bother applying higher-order ODE "solver" to the physics to get a closed-form solution—just use high school physics with incremental time updates! If we want to update a satchel's info after a Δt, we can do the following:

- $y'' = -g$; (for satchel and shot. Zero for platform)
- $y' \mathrel{+}= y''\cdot\Delta t$; (for satchel and shot AFTER the shot was initiated)
- $y \mathrel{+}= y' \cdot\Delta t$; (see edge cases, below)
- $x'' = F_{CapSense}$ / Platform.Mass; (for platform. Zero for satchel and shot)
- $x' \mathrel{+}= x''\cdot\Delta t$ + impulse/Platform.Mass ; (impulse is (RailGun.ShotMass · V_{shot}) projected onto x axis when one is shot)
- $x \mathrel{+}= x' \cdot\Delta t$; (see edge cases, below)

- As for the edge cases, check (after the above updates) if:
  - **The platform or the satchel passed a wall.** If it did, (don't forget to check the collision speed for catastrophic platform impact) reverse the x velocity and make its position reflect back into the playing space. (The platform is shown below, but the same rule can work for satchel charges)



In case you're thinking this through and objecting that the acceleration due to the CapSensed force was applied in a direction either in reinforcement of or in opposition to the velocity, and now we're magically flipping the velocity partway through the Δt but not the force's reinforcement-or-opposition—you're correct. And yet, since Δt is assumed to be "small enough that it doesn't matter", ignore it!

  - **The satchel is below the bottom of the canyon.** If it is, check to see if its x value is between the ends of the platform (after it is also updated, so maybe update the platform first!) and therefore ends the game.
  - **The rail-gun shot is at or to the left of the canyon's left edge**. This is why the cliff and castle are assumed to have zero width for impact consideration—you don't have to adjust for how much of the foundation was already taken out, or how you chose to draw them to figure out an x-position that varies from the left canyon edge. At the point your update shows the shot has "left the canyon's left edge", compute the y position, and compare it to the configuration data's CanyonSize-Wolfenstein.CastleHeight and CanyonSize-(Wolfenstein.CastleHeight + Wolfenstein.FoundationDepth) to see if it hit the castle or foundation. This condition may also trigger a new thrown satchel charge, depending on the SatchelCharges.LimitingMethod.
  - **The rail-gun shoots.** There is a new object to track, whose speed is dependent on the energy imparted and RailGun.ShotMass, and the direction is fixed by the RailGun.ElevationAngle. Don't bother finding the end of the rail gun—just treat the shot as instantaneously having its final velocity as it leaves the BASE of the barrel. Since this is an asynchronous event, you COULD set the time of shot, and make the physics task deal with an update for the first partial Δt, but you could also simply note that there is an impulse to apply in the next Physics Task update, which will affect the shot position and velocity, and impart impulse to the platform.

Platform Force Indication:

You're required to use the CapSense input to control how much force is applied to your platform. However, the only two points that I constrain are that the force applied goes from "full force to left" (with finger only on the left edge) to "full force to right" with finger only on the right edge). You can choose to use the CAPSENSE_sense() sensor measurements in many ways to indicate forcing of the platform (Note: FullForce is actually ConfigurationData.Platform.MaxForce):

- Probably in all cases, you'll want "no touch" to result in zero force. Although, perhaps another interesting choice would be to apply max force opposite the velocity direction if there is "no touch" (resulting in auto-braking).
- Any left with no right: left FullForce; Any right with no left: right FullForce
- If only one pad is sensing touch, Pad0: left FullForce. Pad1: left ½FullForce; Pad2: right ½FullForce; Pad3: right FullForce. What would you do if 2 pads were sensing touch at the same time?
- Use the interpolation function CAPSENSE_getSliderPosition() to give you a position from 0 to 256, and adjust it to a signed value as: Force=MaxForce*(_getSliderPosition()-128)/128.