Week 1:

Describe at least 2 desirable "cutting points" for testing (see lecture topic 2.5) covering main paths and/or boundary condition handling.

1: I want to get the display able to display a little 2px x 2px dot which I want to move around by simply pressing the buttons. I can use some of Lab 7 as a framework to get started. Once I am happy with the display, I want to get the basics of the physics engine working ASAP so I can continue to make the game with that out of the way.

2: Testing and tuning the physics engine will be necessary, and I would like to get it solid as soon as possible. With the display functional, I will test a single slug's trajectory and check that it matches what I'd expect by doing the math myself and comparing. Once the basic physics calculations are there, I can move on to the position data manipulation that most of the other tasks depend on.

Week 2:

Create your unit testing plan, utilizing at least 3 "cutting points". List and summarize 2 conceptual unit tests for each "cutting point" in the summary after the Unit and Functional test sections.

1: Modular drawing functions

a. Being able to draw all sprites needed for the project (slug, platform, satchel, castle, walls) at any position. It should be able to feed in any coordinate and have sprites drawn there.
b. All sprites should be able to be drawn in a sufficiently short time. Once I find how long the set of sprites takes to draw, that time should be more or less constant, and finding if this time is too long will require shortcuts to be found in drawing.

2: Physics task

a. Pause at any point and verify velocity and positional numbers using the time elapsed and constant acceleration. This could be coded into a set of simple kinematics functions that take the elapsed time, returns either velocity or position and use this result to compare against real measured values in an assert.

b. Add asserts for arbitrarily large values that the data should never exceed. Positional data can be checked for values over `~260 (because the real position may not exceed 256). Velocity maximum can be calculated based on unit-to-pixel scaling factor and kinematics equations, adding ~10% above that theoretical max. I want to make sure if my program goes off with crazy data at any point, I can catch that quickly in the debugging process.

3. State machine tests

a. I will be implementing a state machine to store the game's logical states and I will add tests to make sure the game state is what we would expect given certain parameters. For example, we cannot win if the hostages are dead and we cannot win if the platform has blown up. This will make debugging the gameplay more fluid because incorrect cases will be caught quickly.

b. As a part of the important state machine transitions, I will modify global variables to keep the potential of garbage values to a minimum. Testing that this happened properly would require asserting at the beginning of the task loops depending on the current state and data being examined.

Week 3:

Review your described unit tests, and speculate whether each would Pass or Fail based on your current implementation. Add a section to your test plan for "functional tests", with 10 tests briefly described, and speculate whether they would pass or not as well. (Pass/Fail: likely 80+% will be "Fail" at this point. The P/F-ness does not affect your grade at this point--only describing the (hypothetical) unit tests, describing your functional tests, and indicating which tests would pass or fail will affect your grade at this point.).

60% pass

1. Modular drawing functions
    a. Is the input coordinate greater than the screen boundaries for any function? (Platform, slugs, satchels)
        i. *Pass* - My position in physics tasks has boundaries for each implemented subsystem.
    b. Is the display task being executed in a sufficiently small period of time?
        i. *Pass - the* task is under 20ms / 100ms and has sufficient CPU time for other tasks to run.
2. Physics task

    a. Are any velocity values running away from a reasonable range (EFM_ASSERT(xVel < 1000))
        i. *Pass* - For currently implemented features, velocities are reset consistently, and acceleration calculation is happening correctly.
    b. Are any position values running away from a reasonable range?
        i. *Pass* - Sprites “bounce” off of walls and ceiling of the display and stay within the allowable range of 0-128
    c. Is the physics task taking too long to run?
        i. *Pass* - The physics task is under 5ms, leaving plenty of runtime for other subsystems.
3. State machine tests (STATE MACHINE HASN’T BEEN IMPLEMENTED YET)
    a. Are game structs initialized before the game has started?
        i. *Fail* - No main menu has been implemented
    b. Is the game running during the correct “runGame” state?
        i. *Fail* - No sm implemented.
    c. Does the game correctly change to “gameFail” state once one of the failing end conditions is met?
        i. *Fail* - No game logic has been implemented.
    d. Does the game correctly change to “gameWin” state once one of the victory conditions is met?
        i. *Fail* - No game logic has been implemented.

Week 4:

Carefully specify your functional tests, such that another person could understand how to execute them. Summarize hypothetical-unit and real functional test results. ("NotRun" will make sense for many functional tests that are not possible until you get further)

Functional Tests:

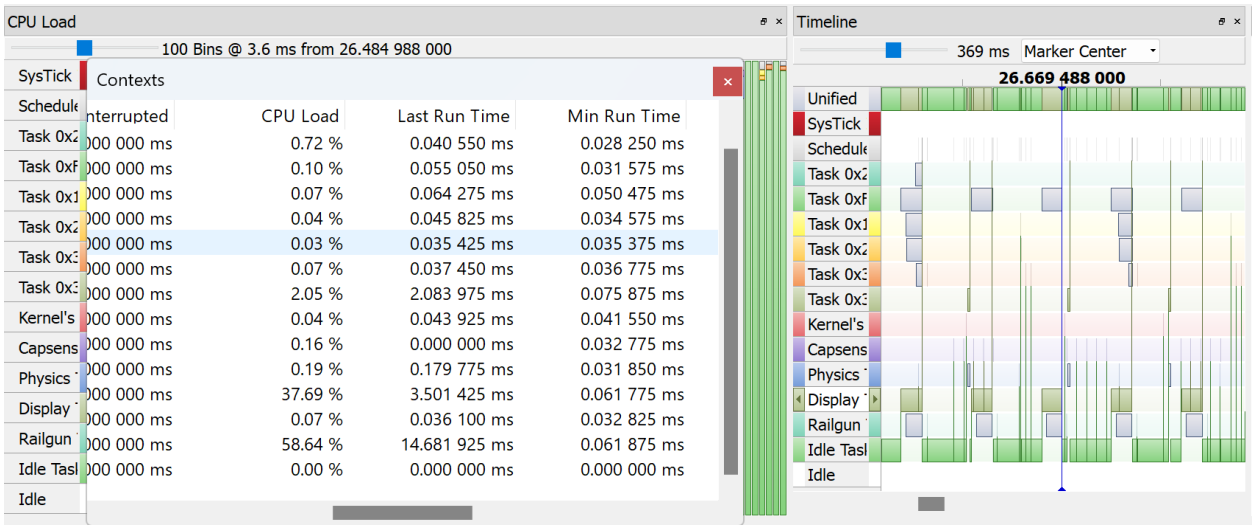
1. To test the slider affecting platform position, place your figner on the far-right side of the capsense slider. You will notice the platform accelerate towards the right and bounce off the right side of the screen. You can move your finger to the far-left side of the capsense and notice the platform will accelerate to the left, causing the platform to slow down, stop briefly, and then start moving to the left. If the platform moves too fast and hits the wall it will explode and the game will end.
2. To test the railgun press and hold BTN1. You will notice the left-hand status bar increasing until you either let go of the button or hit the maximum allowed charge.

When you let go of the bar, you will also see that the generator's total charge will decrease and quickly charge back up.

3. To test the satchel charges, you can either wait for a satchel to randomly hit the platform or intentionally put yourself in its path. When the satchel hits you, you will notice that the game ends because you lost.
4. To test the game state, shoot the foundation of the castle enough to have LED1 start to blink. After it blinks for a moment it will be solid amber, indicating the evacuation is complete. Once the LED is solid amber that indicates you have won and the game will end.

Week 5:

- RT tasks: What are your priorities, execution times vs. deadlines as seen with Segger SystemView? (Include screenshots) Conflicts seen that kept it from operating as you planned? (2)



Interrupted	CPU Load	Last Run Time	Min Run Time
000 000 ms	0.72 %	0.040 550 ms	0.028 250 ms
000 000 ms	0.10 %	0.055 050 ms	0.031 575 ms
000 000 ms	0.07 %	0.064 275 ms	0.050 475 ms
000 000 ms	0.04 %	0.045 825 ms	0.034 575 ms
000 000 ms	0.03 %	0.035 425 ms	0.035 375 ms
000 000 ms	0.07 %	0.037 450 ms	0.036 775 ms
000 000 ms	2.05 %	2.083 975 ms	0.075 875 ms
000 000 ms	0.04 %	0.043 925 ms	0.041 550 ms
000 000 ms	0.16 %	0.000 000 ms	0.032 775 ms
000 000 ms	0.19 %	0.179 775 ms	0.031 850 ms
000 000 ms	37.69 %	3.501 425 ms	0.061 775 ms
000 000 ms	0.07 %	0.036 100 ms	0.032 825 ms
000 000 ms	58.64 %	14.681 925 ms	0.061 875 ms
000 000 ms	0.00 %	0.000 000 ms	0.000 000 ms

I am overall very happy with the execution time and CPU util of my project. Even when many things were being tracked on the screen/physics, the updates were still very quick, and the game never stopped working due to a performance issue. This is also due to me trying to keep loops to the minimum

Task	Priority	CPU Util %
Idle	63	0
Capsense	25	0.16
Display	31	37.7
Physics	30	0.19
Railgun	32	58.64
Satchels	34	<.1
Generator	35	<.1
Collision	36	<.1
Shield	37	<.1
LEDs	38	<.1

and employing optimizations whenever I could. Priorities need to be reevaluated once near the beginning because the cap sense was having weird bugs if its priority wasn't very high. After that, I set arbitrary priorities, and the game worked well from there.

- Code Space: how much, and evaluative comments (2)

Flash used: 130720 / 1024000 (12%)

RAM used: 70968 / 256000 (27%)

I am somewhat happy with these numbers as it shows that I wasn't up against any wall for code space, but I was expecting it to be a little lower.

- Evaluation of your approach(es) to the physics update requirement. Explain your rationale for grouping the physics updates and edge case handling the way you did. What limitations did you have to deal with that were not obvious at first? (3)

My physics update task would pend a mutex for a given subsystem, the railgun shots (slugs), for example, and change the velocity and position based off a constant time delta that the task was set to. The physics task was set to update every 100ms, so every 10th of a second each slug would receive an update and once the mutex was posted and the task suspended, then the display task could look at that public data and draw figures accordingly. I had very little trouble implementing the physics as it was just simple kinematics, and I am sure that I could've implemented some air drag, friction, and more if needed. I had a mental image of how each subsystem would interact, and that image was solid the entire time, so instead of focussing on this as much of a programming project, I tried to practice some of my lacking techniques and try to match the project descriptions. Some things I focussed on was keeping accurate header files, keeping includes clean, having verbose variable names, and keeping everything modular and referencing modular constants.

- Scaling of variable spaces (think back to the lecture including "corner testing"): what ranges did you find playable? (1)
    - Probing questions: Which configuration data did you find unnecessary? What do you wish was more constrained instead of being undefined?

```
// Mutable Consts
    game.mutable_consts.max_satchel_xvel = 50;
    game.mutable_consts.max_satchels = MAX_SATCHELS;
    game.mutable_consts.max_slugs = MAX_SLUGS;
    game.mutable_consts.satchel_origin_height = 40;
    game.mutable_consts.satchel_floor = 127;
    game.mutable_consts.limitType = periodicThrowTime;

// Slug config
    for (int i = 0; i < game.mutable_consts.max_slugs; i++)
    {
        game.app_slug_data[i].mass = 2.5;
        game.app_slug_data[i].width = 3;
        game.app_slug_data[i].slugID = i;
        game.app_slug_data[i].live = false;
        game.app_slug_data[i].hasLeftPlatform = false;
    }
// Platform config
    game.app_platform_data.width = 10; // 10m width
    game.app_platform_data.centerPos = 64; // Begin centered
    game.app_platform_data.mass = 100; // 100 KG
    game.app_platform_data.forceConstant = 12.5; // 100 N
    game.app_platform_data.floor = 120; // Good spot for it to live
    game.app_platform_data.theta = 45;
    game.app_platform_data.railgunLength = 4;
    game.app_platform_data.height = game.app_platform_data.floor
            - (game.app_platform_data.railgunLength * sin(game.app_platform_data.theta));
    game.app_platform_data.maxBounceVelocity = 200;
```

```
// Railgun Config
    game.app_railgun_data.theta = game.app_platform_data.theta; // Set railgun theta to match platform
    game.app_railgun_data.chargeRate = 1000;
    game.app_railgun_data.currentCharge = 0;
    game.app_railgun_data.maxCharge = 5000;
// Satchel Config
    for (int i = 0; i < game.mutable_consts.max_satchels; i++)
    {
        game.app_satchel_data[i].width = 3;
        game.app_satchel_data[i].height = 3;

    }

// Generator / Capacitor
    game.app_generator_data.energyCapacity = 25000;
    game.app_generator_data.chargeRate = 5000;
    game.app_generator_data.currentCharge = 0;
    game.app_generator_data.tauGenerator_ms = TAU_GENERATOR_MS;
// Shield config
    game.app_shield_data.activationEnergy = 2500;
    game.app_shield_data.energyDrainPerSecond = 1.5 * game.app_generator_data.chargeRate;
    game.app_shield_data.active = false;
    game.app_shield_data.maxRange = 20;
// Statemachine setup
    game.app_sm.states = menu;
```

```
// Castle Config
    game.app_castle_data.castleFloorHeight = 40;
    game.app_castle_data.foundationDepth = 128 - game.app_castle_data.castleFloorHeight;
    game.app_castle_data.foundationMaxHealth = 3;
    game.app_castle_data.foundationHealth = game.app_castle_data.foundationMaxHealth;
    game.app_castle_data.evactuationTime = 5;
// Menu Config
    game.app_menu_data.numOptions = 3;
    game.app_menu_data.currSelection = 0;
```

I used nearly every constant that was suggested for use and didn't really find any to be unnecessary. I used my own values for many of the constants because I didn't get around to making the game scalable for any height/width and only used the 128x128 meter square. I am very happy with the playability of my game and I

did the math at one point and found many of the parameters to be a similar ratio to Jon's provided data.

- What would be your next steps if you had just another 2 weeks to work on your project, and why? (2)

If I had another 2 weeks, I would tackle the game scaling issue and polish the menu screens. I envisioned having EVERY variable be mutable in the menu so you can customize and break the game however the player saw fit. I thought it might have also been cool to add a breakout board for a joystick or extra buttons for extra functionality, such as tilting the railgun elevation angle or getting more precise control over the platform. A final thing I would've wanted to experiment with would be keeping a leaderboard of all the completion times and having ending "credits" that would scroll each of the winning scores similar to that of an arcade cabinet.

Extra credit points:

- I added functionality for 3 total satchel modes (alwaysOne, maxInFlight, and periodicThrow)
- I added a main menu with the ability to change what satchel mode was in use for the game.