# Iteration Two

Time you enjoy wasting is not wasted time.

## The goal

To add a time limit for each level

### Tasks

|  |  |  |
| --- | --- | --- |
| **Task** | **Time Estimate** | **Actual Time Taken** |
| Analysis and Planning | 90 | 90 |
| Coding | 90 | 60 |
| Testing | 30 | 30 |

### Analysis and Planning

The requestAnimationFrame() function is a window function that passes a callback function to be called when the browser is ready to process another frame. When the browser runs the callback function it passes the argument of a high-resolution timestamp, a period of time measured in milliseconds since the page opened.

The global function runAnimation() passes a function called frame() to requestAnimationFrame(). frame() accepts this high-resolution timestamp and converts it to the time elapsed since the last frame was processed, or 100 milliseconds, whichever is the smaller value. It also divides the timestamp by 1000 so it is measured in seconds (e.g. The largest possible value is 0.1 seconds).

I originally thought to track time directly in the state object, but quickly realised this would require heavy refactoring of code to pass the time as an argument each time the state is updated. Instead I will set a countdown value as a prototypal variable, which will remain constant across State instances, and use getters and setters to access it directly.

The countdown timer will be initialised as a number before the rendering of each level. This number will be the number of seconds allowed for the given level, which will be defined in the level plans. The timestep will be subtracted from this in a function called State.countDown() which will accept the timestep as an argument. This method will be called from within state.update() and return true if there is time left, false if there is not. If there is no time left the status will be set to lost, ending the level.

To display the countdown timer I will add methods to the CanvasDisplay class to convert the time left to minutes seconds and milliseconds (if there are less than 15 seconds left). These values will be drawn to the canvas after all other game objects, making it an overlay. This method will be called drawOverlay().

### Class Diagram Before:



### Class Diagram After:



### Design

## Activity Diagram



## Planning a Complex Algorithm

Define the problem

*Want to add a timer*

Inputs to the routine

*A timestep, measured in seconds*

Outputs from the routine

*A Boolean value. False if there is no time left, True otherwise*

Pre-conditions

*State timer is defined*

Post-conditions

*State timer will be decreased by the amount given by the timestep. State timer will not go below zero*

Name the Routine

*State.countDown()*

Think about error handling

*If timestep will take state timer below zero, state timer will be set to zero and the function will return false. Functionality will be tested in browser*

### Pseudocode

State.countDown(timestep) {

Decrease my timer by timestep  
 if my timer is less than 0 {

Set timer to 0  
 return false

}  
 return true

}

### Evaluation

I originally planned to attach the timer to the level class because it is a constant throughout the playing of a level. I had this working before I decided I would prefer to attach it to the State prototype, using get and set to make it directly accessible.

During the testing of that code I learned that attaching variables to the State prototype caused ghosting, where the new variables showed up in unexpected places such as other classes. To fix this problem I defined an empty Silo class to act as the parent of State, which isolated the prototypal attributes of State from other classes.

I put the updated code through standardJS and there were no styling defects.

### standardJS Before

