SSUI Programming Assignment 1 Writeup

released (*Aug 29, 2024*)
due: Sept 12, 2024; 11:59PM ET (Fri) on Canvas

Submission instructions: please zip the project directory with all your files and submit to the Canvas portal.

Task: Implement a Fitt's Law Tester

Preliminaries

Demo: there is a video on Canvas showing off the desired functionality of the completed assignment. It can be found in:

- `Files > Project, Homework, etc. Related > Project1 Resources > ssui-p1-demo.mp4`

Files you should have

ssui-p1

```
- **p1-skel.ts*
- index.html (*use this to launch project after transpiling*)
- tsconfig.json
- among other files
```

If you've set things up correctly, the only thing you should need to touch is going to be the typescript file, which will be titled `p1-skel.ts`

Grading Criteria

```
1. Functional user interface & game for the Fitt's Law Test
2. Commented & well-structured code (rule of thumb: if your code is >= 5 lines and not immediately self evident, explain what it does)
```

This assignment is pretty simple. Get the code to run and play the game as intended, and you will score pretty well overall. To get the highest marks, we do ask that you explain the logic for your underlying code, and so we'll hold back some amount of points for general good code hygiene.

Read the code for `UIObject` and `ScreenObject`

All of the code to be implemented for this assignment depends on understanding of the the parent classes `UIObject` and `ScreenObject`. It is highly recommended that you read through the code for these classes before attempting to write any code.

Please ask on Slack with any questions about these classes!

Tasks for P1

All of the places where you need to implement code will be marked by a

```
// === YOUR CODE HERE ===
```

`class FittsTestUI`

If you look through the code, you'll have base classes like UIObject and ScreenObject which are already defined for you. Please please do read through the code so you're familiar with these classes before going on to implement subclasses which extend these classes.

`FittsTestUI` is the first thing that you'll run into that you'll be expected to implement.

The basic premise of the UI or game state is that it has 4 total states

- start: display a background screen with instructions
- begin_trial: display a Reticle which requires the user to place their mouse cursor on -- user should have their mouse pointer directly in a small circle
- in_trial: display a random sized Target - this should look different than the Reticle
- ended: shows an information screen that the game has ended

(you can find these states in the `switch` statement) you'll be expected to hold, assign, store, modify data to some degree every time there's a state change.

`class Target (extends ScreenObject)`

You'll also be implementing the subclass `Target`, which represents the green blobs that you saw in the demo video.

'newGeom()'

-- where you set the parameters for the bounding box surrounding the object

'draw()'

-- where you write the code to actually how to represent the object on the screen

'pickedBy()'

-- which checks whether your cursor selection is indeed within the Target's dimensions
Remember, the Target is a circle, so it should not count as a valid 'pick' if you're in the bounding box, but not in the circle!

'handleClickAt()'

-- where we advance the game state to the next stage

`class Reticle (extends Target)`

Then we have Reticle, which is the specific target type which we click on to start a trial. We have some similar functions to write for the `Reticle` class - (it looks and acts different!)

'draw()'

-- where you write the code to actually how to represent the object on the screen

'pickedBy()'

-- which checks whether your cursor selection is indeed within the Target's dimensions
Remember, the Target is a circle, so it should **not** count as a valid 'pick' if you're in the bounding box, but not in the circle!

'handleClickAt()'

-- where we advance the game state to the next stage
-- remember we're at the **beginning of the trial** so the game state advancing might be different

`class BackgroundDisplay (extends ScreenObject)`

And like before, we have similar functions for you to implement for `BackgroundDisplay`

'draw()'

-- where you display instructions

'handleClickAt()'

-- advance the game state

Walkthrough / Handout on Canvas!
GO TO FILES

> RESOURCES
> `TYPESCRIPT-SETUP-SSUI.PDF`
Everyone have access to it? I'll let people go through it and take say…. 15 - 20 mins for people to look over it and follow all the steps. Please raise your hand if you're running into any errors.

Programming Assignment 1

RELEASED (*TODAY!*)
due: Sept 15, 2023 (Fri)

P1

Implement a Fitt's Law Tester
(DEMO TIME)

File Structure for P1
SSUI-P1
- **p1.ts (or p1-skeleton.ts)**
- index.html (*use this to launch project*)
- package.json
- style.css
- tasks.json
- tsconfig.json

If we've set things up correctly, the only thing you should need to touch is going to be the typescript file, which will either be titled p1.ts or p1-skeleton.ts

Grading Criteria
1. Functional Fitt's Law Test!
2. Commented code (rule of thumb: if your code is >= 5 and not immediately self evident, explain what it does)

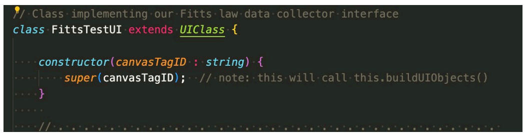
This assignment is pretty simple. Get the code to run and you will score pretty well overall. To get the highest marks, we do ask that you explain the logic for your underlying code, and so we'll hold back some amount of points for general good code hygiene.

Scott wrote the code for this assignment and he set a pretty high bar

Things to Implement for P1

So let's quickly walk through what there is to implement for this assignment.

`class FittsTestUI`

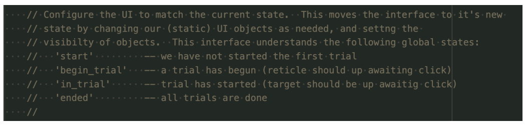


```
// Class implementing our Fitts law data collector interface
class FittsTestUI extends UIClass {

    constructor(canvasTagID : string) {
        super(canvasTagID);  // note: this will call this.buildUIObjects()
    }

    //..................................................
```

If you look through the code, you'll have base classes like UIObject and Screen-Object which are already defined for you. Please please do read through the code so you're familiar with these classes before going on to implement subclasses which extend these classes.

FittsTestUI is probably the first thing that you'll run into that you'll be expected to implement.

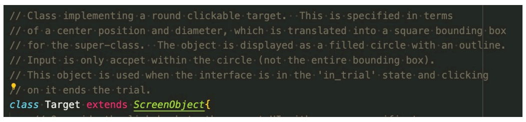
`class FittsTestUI`



```
    // Configure the UI to match the current state.  This moves the interface to it's new
    // state by changing our (static) UI objects as needed, and settng the
    // visibilty of objects.  This interface understands the following global states:
    //   'start'         -- we have not started the first trial
    //   'begin_trial'   -- a trial has begun (reticle should up awaiting click)
    //   'in_trial'      -- trial has started (target should be up awaitig click)
    //   'ended'         -- all trials are done
    //
```

The basic premise of the UI or game state is that it has 4 total states – start, begin trial, in trial, and (which you can find in the switch statement a few more lines down in FittsTestUI) you'll be expected to hold, assign, store, modify data to some degree every time there's a state change.

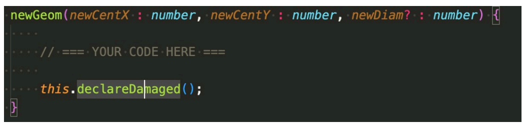
`class Target (extends ScreenObject)`



```
// Class implementing a round clickable target.  This is specified in terms
// of a center position and diameter, which is translated into a square bounding box
// for the super-class.  The object is displayed as a filled circle with an outline.
// Input is only accpet within the circle (not the entire bounding box).
// This object is used when the interface is in the 'in_trial' state and clicking
// on it ends the trial.
class Target extends ScreenObject{
```

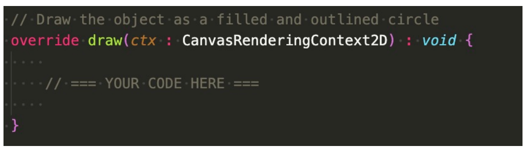
You'll also be implementing the sub-class Target, which represents the green blobs that you saw in the earlier example.

'NEWGEOM'



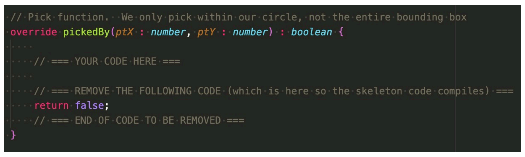
```
newGeom(newCentX : number, newCentY : number, newDiam? : number) {

    // === YOUR CODE HERE ===

    this.declareDamaged();
}
```

'DRAW'



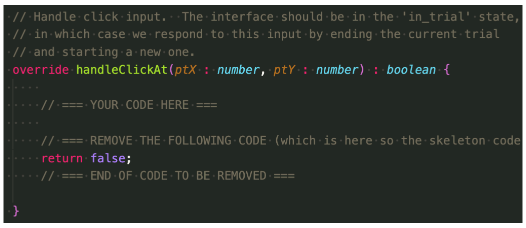
```
// Draw the object as a filled and outlined circle
override draw(ctx : CanvasRenderingContext2D) : void {

    // === YOUR CODE HERE ===

}
```

'PICKEDBY'



```
// Pick function.  We only pick within our circle, not the entire bounding box
override pickedBy(ptX : number, ptY : number) : boolean {

    // === YOUR CODE HERE ===

    // === REMOVE THE FOLLOWING CODE (which is here so the skeleton code compiles) ===
    return false;
    // === END OF CODE TO BE REMOVED ===
}
```

'HANDLECLICK'



```
// Handle click input.  The interface should be in the 'in_trial' state,
// in which case we respond to this input by ending the current trial
// and starting a new one.
override handleClickAt(ptX : number, ptY : number) : boolean {
    
    // === YOUR CODE HERE ===
    
    // === REMOVE THE FOLLOWING CODE (which is here so the skeleton code
    return false;
    // === END OF CODE TO BE REMOVED ===

}
```

There'll be four total functions for you to implement in 'class Target'

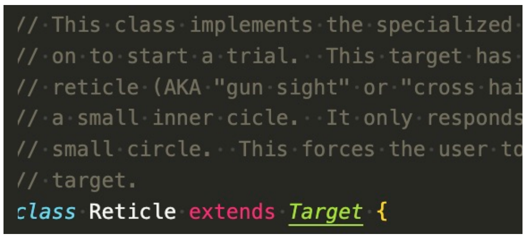
newGeom – where you set the parameters for the bounding box surrounding the object

draw – where you write the code to actually how to represent the object on the screen

pickedBy – which checks whether your cursor selection is indeed within the Target's dimensional bounds – remember, our Targets are circles!

handleClick – where we advance the game state to the next stage

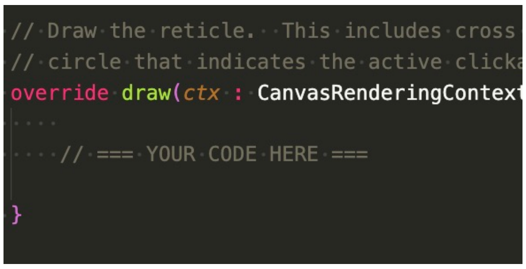
`CLASS RETICLE (EXTENDS TARGET)`



```
// This class implements the specialized
// on to start a trial.  This target has
// reticle (AKA "gun sight" or "cross hai
// a small inner cicle.  It only responds
// small circle.  This forces the user to
// target.
class Reticle extends Target {
```

Then we have Reticle, which is the specific target type which we click on to start a trial

'DRAW'



```
// Draw the reticle.  This includes cross
// circle that indicates the active clicka
override draw(ctx : CanvasRenderingContext
    
    // === YOUR CODE HERE ===

}
```

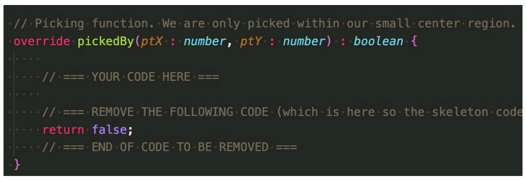
'PICKEDBY'

'HANDLECLICKAT'

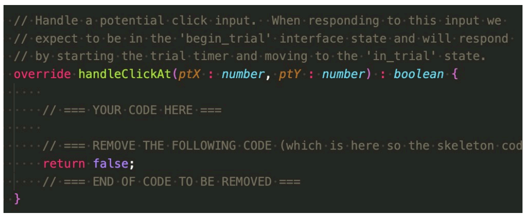
we have some similar functions to write for the Reticle Class - namely draw (it looks different!)

pickedBy - we're more selective about how close we want the cursor to be before we return True

handleClickAt - remember we're at the beginning of the trial so the game state advancing might be different

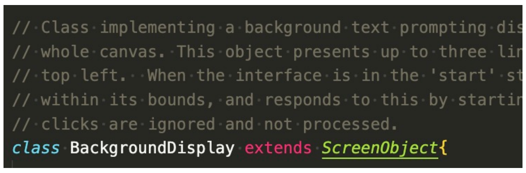


```
// Picking function. We are only picked within our small center region.
override pickedBy(ptX : number, ptY : number) : boolean {
    
    // === YOUR CODE HERE ===
    
    // === REMOVE THE FOLLOWING CODE (which is here so the skeleton code
    return false;
    // === END OF CODE TO BE REMOVED ===
}
```



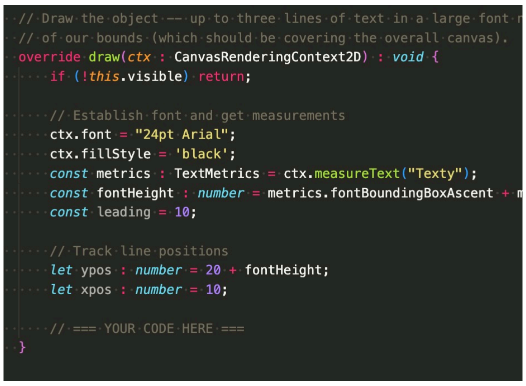
```
// Handle a potential click input.  When responding to this input we
// expect to be in the 'begin_trial' interface state and will respond
// by starting the trial timer and moving to the 'in_trial' state.
override handleClickAt(ptX : number, ptY : number) : boolean {
    
    // === YOUR CODE HERE ===
    
    // === REMOVE THE FOLLOWING CODE (which is here so the skeleton cod
    return false;
    // === END OF CODE TO BE REMOVED ===
}
```

`CLASS BACKGROUNDDISPLAY (EXTENDS SCREENOBJECT)`



```
// Class implementing a background text prompting dis
// whole canvas. This object presents up to three lin
// top left.  When the interface is in the 'start' st
// within its bounds, and responds to this by startin
// clicks are ignored and not processed.
class BackgroundDisplay extends ScreenObject{
```

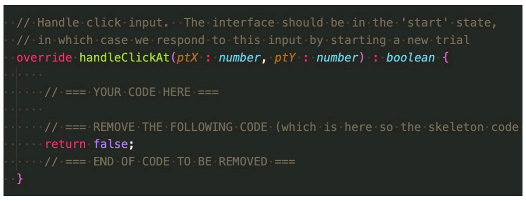
‘DRAW’



```
  // Draw the object -- up to three lines of text in a large font r
  // of our bounds (which should be covering the overall canvas).
  override draw(ctx : CanvasRenderingContext2D) : void {
      if (!this.visible) return;

      // Establish font and get measurements
      ctx.font = "24pt Arial";
      ctx.fillStyle = 'black';
      const metrics : TextMetrics = ctx.measureText("Texty");
      const fontHeight : number = metrics.fontBoundingBoxAscent + m
      const leading = 10;

      // Track line positions
      let ypos : number = 20 + fontHeight;
      let xpos : number = 10;

      // === YOUR CODE HERE ===
  }
```

‘HANDLECLICKAT’

and like before, we have similar functions for you to implement.



```
  // Handle click input.  The interface should be in the 'start' state,
  // in which case we respond to this input by starting a new trial
  override handleClickAt(ptX : number, ptY : number) : boolean {
      
      // === YOUR CODE HERE ===
      
      // === REMOVE THE FOLLOWING CODE (which is here so the skeleton code
      return false;
      // === END OF CODE TO BE REMOVED ===
  }
```

That’s it! 👋

Be sure to join the Slack

and please feel free to ask any questions, I’ll stick around until the end of lab time.