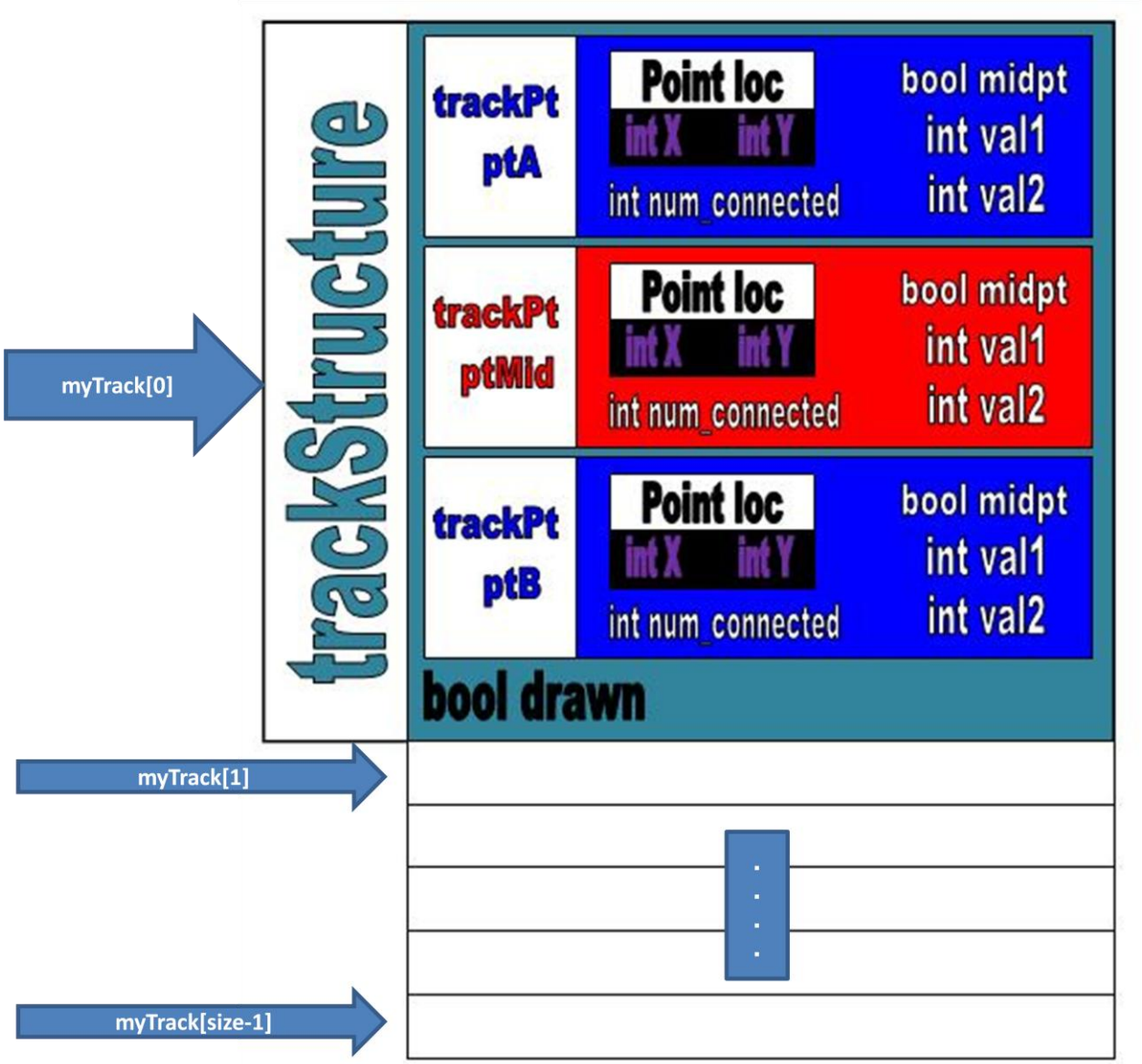
Important Algorithms, How they Work, and Where to Find Them
Track Designer:

I. Data Structure(s) Used—find them in trackObjects.cs
   a. trackPt—a "point" defined by:
      i. loc: a point object containing x and y coordinates on the form. Ellipses are drawn at these points, and their colors depend on whether they are midpoints or turnouts.
      ii. Num_connected: integer value stating how many segments are connected to the point. If the value is 3, the point is a turnout. Otherwise it is a sensor.
      iii. Midpt: Boolean variable saying whether or not the point is a midpoint. Midpoints are the only ones allowed to be dragged by the user, so this variable must be checked to determine if the user can drag the point.
      iv. Val1: index of the segment not to be travelled in a turnout. It is the first segment drawn when a turnout is being drawn. In a sensor or midpoint, this value is set to -1, an invalid index.
      v. Val2: index of the segment to be travelled in a turnout. It is the second segment drawn when a turnout is being drawn (after the "box" is drawn to cut the track). In a sensor or midpoint, this value is set to -1, an invalid index.
   b. TrackStructure—a segment of track defined by:
      i. trackPtA: first of 2 points clicked by the user to define a segment of track.
      ii. trackPtMid: internally calculated point. This is the midpoint between trackPtA and trackPtB. It can only be calculated after the second click.
      iii. trackPtB: second of 2 points clicked by the user to define a segment of track.
      iv. bool drawn: tells whether or not the segment has been drawn. This is used to make sure turnouts are drawn properly. Segments need not be consecutive, so I had to have a way to draw all three adjacent segments when drawing a turnout. I also had to make sure the segment didn't get redrawn later because it could mess up a turnout.



II. Creating segments—find this algorithm in editForm.cs

a. Description: In order to create a segment of track, a user must click two appropriate defining locations. Appropriate locations include:

i. Places within the black lines where NO points are currently located

ii. TrackPts that are NOT midpoints and have 2 or less segments connected to them

If a user fails to click two appropriate locations in a row, the counter variable resets to 0. This allows a user to begin creating a new segment, get "distracted" and drag a midpoint, and then forget they started the new segment rather than having to remember where they started creating the new segment. When the user has successfully clicked two points, they are saved, their midpoint is calculated, and they are added to the segment array.

b. Variables Used:

i. pCount: integer stating how many points have been clicked. Valid values are 0 and 1. After the second valid point is clicked, pCount is set back to 0.
ii. Temp: trackPt whose location tells me whether the user is dragging a midpoint (-1, -1), on an invalid point (-1, Y), or on a valid point (anything else).

c. Room for Improvement: There should be some type of restriction that does not allow a user to make segments cross one another. If this is allowed, there needs to be some type of detection algorithm.

III. Initializing turnouts—find this algorithm in editForm.cs

a. Description: when a trackPt's num_connected variable gets to 3, I know the user has added a turnout. They are then asked to "click the midpoint of the curved segment" followed by "click the midpoint of the straight segment". This is so I can fill in the val1 and val2 variables on the turnout. The user cannot add any more segments until they have successfully initialized the turnout.
b. Variables Used:
i. Add_turnout: Boolean variable that gets set to true when num_segments gets to 3, and does not get set back to false until the 2 midpoints have been clicked.
ii. Index: integer variable telling the index of the segment whose midpoint has been clicked
iii. pCount: integer variable telling how many midpoints have been clicked. Valid values are 0 and 1. After the second valid point is clicked, pCount is set back to 0.
iv. aORb: character telling which endpoint of the segment is the turnout. This tells me in which trackPt to change the values of val1 and val2.
c. Room for Improvement: Message boxes are ugly!! Also, if a user clicks "OK" without reading the directions, they do not know what to do and there is no way for them to find out. A better mechanism for giving directions would be nice.

IV. Dragging midpoints to create curvature—find this in editForm.cs

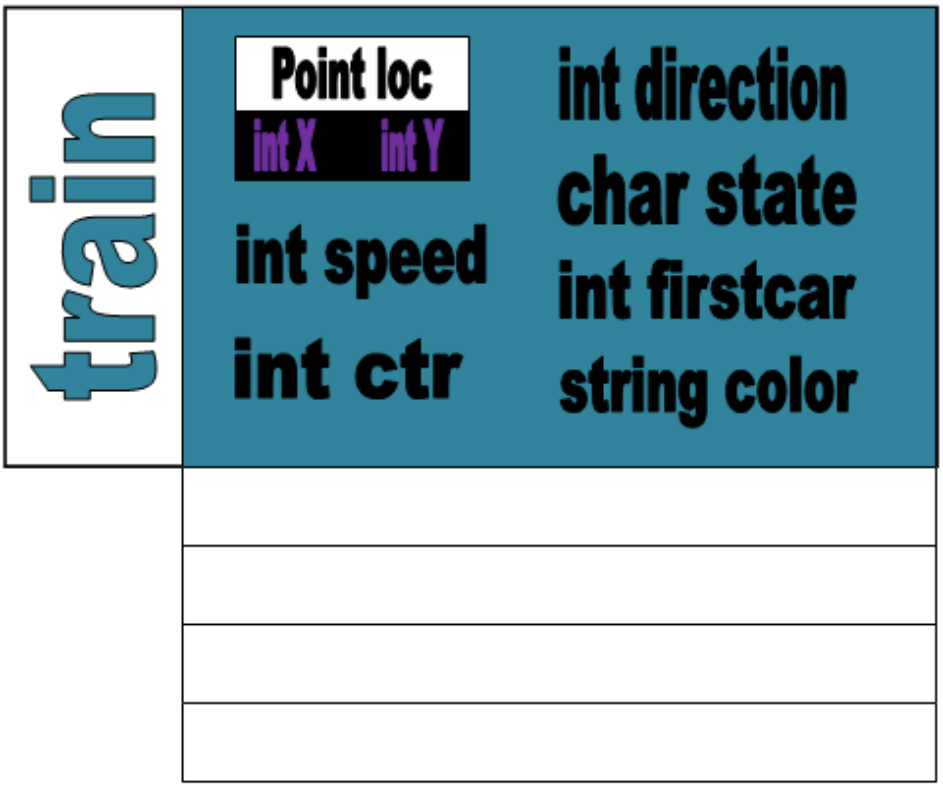
a. Description: when a user creates a segment, a red dot between their two endpoints shows up. This is their midpoint. In order to create the desired track (curvature), users are allowed to drag these midpoints.
b. Variables Used:
i. Temp: trackPt object whose location X and Y values tell me if it's a midpoint or not. If they are both set to -1 by the on_point function, the click was on a midpoint.
ii. Index: integer value that tells me the segment index whose midpoint was clicked on. If no midpoint was clicked, this is -1, an invalid index value. This is used in the mouse_move event. I only want to redraw the screen (showing the midpoint moving) if this index is valid.
iii. Mdown: Boolean variable that tells me if the mouse is down. This is used inside the event handler for the mouse_move event. I only want to redraw the screen (showing the midpoint moving) if the mouse is down.

iv. Addturnout: Boolean variable telling if the user is in the process of initializing a turnout. I do NOT want to perform any of the mouse_move events if they are adding a turnout.

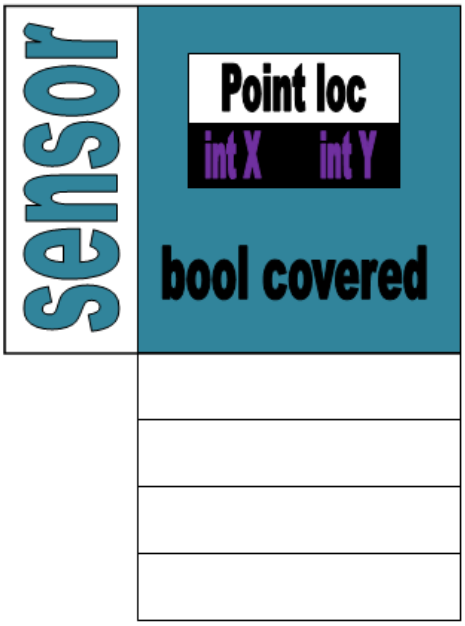
V. Deleting Segments—find this in editForm.cs
   a. Description: A user may not "like" the look of a track they have created, so I allow them to delete any/all segments. When they click the "Delete Segments" button, a checkedListBox dynamically populates based on how many segments are currently in the track. When the user checks one of the boxes, that segment of track becomes highlighted. They are allowed to highlight as many segments as they wish to delete. Then, they must click the "Delete Selected" button to get rid of the selected segments. The segments then shift upwards in the trackStructure array to fill the holes of the deleted segments.
   b. Variables Used:
      i. Temp: trackStructure array used to hold the remaining segments before they get copied back to the real array.
      ii. Add_mode: Boolean variable stating if the user is adding segments. If yes, then the delete stuff does not happen.
   c. Room for Improvement: Currently, I do a lot of shifting. I think this could be made better by simply filling the holes that were created.

Simulator:

I. Data Structure(s) Used:
   a. Train—defined by:
      i. Loc: point object with X and Y coordinates telling the location on the screen
      ii. Speed: integer value ranging from 0 to 10 indicating the speed of the train
      iii. Ctr: integer value that begins at "speed" and gets incremented on each timer tick (if the speed is > 0). When ctr reaches 10, the train gets moved, and ctr gets initialized back to speed.
      iv. Direction: integer value telling the last direction that the train moved.
      v. State: character value indicating the state of the train. 'a'=active, 'c'=crashed, 'd'=derailed, etc. I currently do NOT use this.
      vi. Firstcar: integer value telling the index of the firstcar. This is used for the implementation in which a train consists of more than one car (in which an array of loc would be maintained instead of just one point). The array is then a circular queue in which firstcar is the head car.
      vii. Color: string value stating the color of the train.

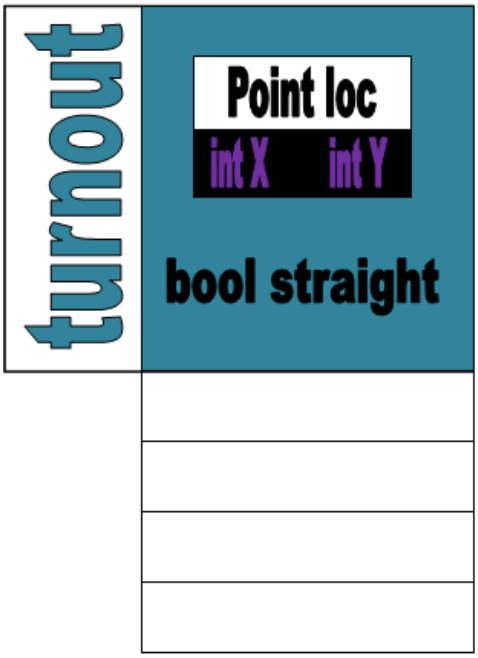


b. Sensor—defined by:
   i. Loc: point with X and Y coordinates telling the location on the screen.
   ii. Covered: Boolean variable stating whether or not the sensor is currently covered. I do NOT currently use this.



c. Turnout—defined by:

i. Loc: point with X and Y coordinates telling the location on the screen.
ii. Straight: Boolean variable stating whether the turnout is currently straight or curved. I do NOT currently use this.



II. Adding a train—find this in trackDesigner.cs and addTrain.cs
   a. Description: In the simulator form, the addTrain button becomes enabled after a user defines a track consisting of 1 or more segments. When they click the button, a message box directs them to initialize the train's location by clicking on a sensor. Another form gets opened as well, but everything remains disabled until they have clicked a proper location. The user is allowed to choose the color, speed and direction of the train on this other form.
   b. Variables Used:
      i. addingTrain: Boolean variable used to determine how to handle the click event. If the user is adding a train, the click event looks to see if they have clicked on a sensor or not.
      ii. trainPoint: Point object (with X and Y coordinates) used to determine if the user has clicked on a sensor or not. This location is also used to create a new train if it is a proper one (in other words if it is on a sensor).
      iii. trainColor, trainDir, trainSpeed: string, integer, integer values used to create a new train. These are passed as properties from the addTrain form after the "Create Train" button is clicked.
   c. Room for Improvement: This is another case where I use a message box to give the user direction. I would like to see this changed as I am not a huge fan of message boxes.

III. Moving the train—find this in trackDesigner.cs

a. Description: Every train has a data member that holds an integer representing its current direction. On the appropriate timer tick (depends on the speed of the train), a train asks whether it can continue in that direction. The answer is yes if a step in that direction lands the train on track color (black, blue, or green). Otherwise, the answer is no. If the answer is no, it looks one step ahead in a clockwise direction and one step ahead in a counterclockwise direction. If it can go in either of those, it will. Otherwise, it gets stuck.
b. Variables Used:
    i. Dir: integer value inside each train object stating the train's current heading.
    ii. Compass: array of points that tell the "steps" for each of the eight directions. For example, direction 0 is "east", and compass[0]=[1,0] which is one pixel to the right.
    iii. DELTA: constant integer that tells how large of a step to make. Currently I have this set to 1, so a step is 1 pixel. The train becomes a little more unpredictable and a little faster if delta is larger.
    iv. Pixel: color variable that holds the color of the pixel at a given location. In the case of this algorithm, it can hold the color of the next step in the same direction, the next step in a counterclockwise direction, or the next step in a clockwise direction.
c. Room for Improvement: There is a lot of room for improvement with this algorithm. The train appears to jerk back and forth within the track. This is because the train's location is actually just the center pixel of the dot it is represented by. Since the track is more than one pixel wide, it can move around within the track. The train can also get stuck in a track that looks plausible. A refinement to the compass or some sort of recursive algorithm might make this better. The problem is that it needs to be FAST. The timer currently ticks ten times per second, so this algorithm cannot take a long time to run—especially if there is more than one train on the track.

IV. Populating the status form—find this in simulatorStatus.cs
a. Description: When the simulator is turned on, a status form opens showing the state of the simulator—all turnouts, sensors and trains. This form is dynamically populated based on how many of each there are.
b. Variables Used:
    i. tCount: integer variable that tells how many turnouts there are. One radiobutton is created for each turnout.
    ii. sCount: integer variable that tells how many sensors there are. One label is created for each sensor.
    iii. trCount: integer variable that tells how many trains there are. One radiobutton is created for each train, and the text is colored in the same color as the train.

V. Changing the state of a turnout—find this in simulatorStatus.cs
a. Description: When the simulator is running, the user is allowed to change the state of any turnout by checking its radioButton and clicking the "Change Turnout"

button. All this entails is swapping the val1 and val2 variables inside the trackPt. The form then raises an event in trackDesigner.cs to redraw the screen. The new state will be visible because of the val1/val2 swap in the trackPt.

b. Variables Used:
    i. rbTemp: radioButton object used to determine which radioButton was checked when the button was clicked. This tells which turnout needs to be changed.
    ii. P: point object that holds the location of the turnout. This is used to match the trackPt to the turnout—they have the same location.

c. Room for Improvement: Turnouts are not always drawn properly. The only way to ensure that they are drawn properly using val1 and val2 is to ensure that only one endpoint of a segment at max can be a turnout. This is easier said than done. It is not as simple as saying "this segment has a turnout" because I do not put restrictions on segments to be consecutive. If I try to add a turnout to one segment, that also affects two other segments because that trackPt is an endpoint of two other segments. I could add a data member to the trackPt object saying whether it is already part of a segment that has a turnout. This would require updating both trackPts of all three segments connected to the turnout when a turnout is added. It seems like there has to be a better way.