Report:

1. Adding Touch Support:

```
virtual bool onTouchBegan(cocos2d::CCTouch *touch, cocos2d::CCEvent *event);
virtual void onTouchMoved(cocos2d::CCTouch *touch, cocos2d::CCEvent *event);
virtual void onTouchEnded(cocos2d::CCTouch *touch, cocos2d::CCEvent *event);
```

And register them in the init function:

```
auto listener = EventListenerTouchOneByOne::create();

// Adding the touch events
listener->onTouchBegan = CC_CALLBACK_2(EngTestScene::onTouchBegan, this);
listener->onTouchMoved = CC_CALLBACK_2(EngTestScene::onTouchMoved, this);
listener->onTouchEnded = CC_CALLBACK_2(EngTestScene::onTouchEnded, this);

_eventDispatcher->addEventListenerWithSceneGraphPriority(listener, this);
```

2. The AStarSearch function is in Utility.h/.cpp

   Here I tried four heuristic functions:

   a) Compare the squared distance to the target point

```
a.getDistanceSq(gTarget) > b.getDistanceSq(gTarget)
```

   b) Compare the sum of delta x and y between current and target point

```
abs(a.x - gTarget.x) + abs(a.y - gTarget.y) > abs(b.x - gTarget.x) + abs(b.y - gTarget.y);
```

   c) First compare the squared distance to the start, if equals, then compare the squared distance to the target

```
float tmpa = a.getDistanceSq(gStart), tmpb = b.getDistanceSq(gStart);
if (floatEquals(tmpa, tmpb)) {
    return a.getDistanceSq(gTarget) > b.getDistanceSq(gTarget);
}
return tmpa > tmpb;
```

   d) Compare the minimum of squared distances between current to start and current to target

```
float starta = a.getDistanceSq(gStart), startb = b.getDistanceSq(gStart);
float targeta = a.getDistanceSq(gTarget), targetb = b.getDistanceSq(gTarget);
return min(starta, targeta) > min(startb, targetb);
```

   And finally, I found that the d is the best one.

   The A* Search is pretty simple, just use Breadth First Search + Priority Queue.

   I have two hash map, one is to record the previous grid, the other is to record how many steps it takes to get this grid.

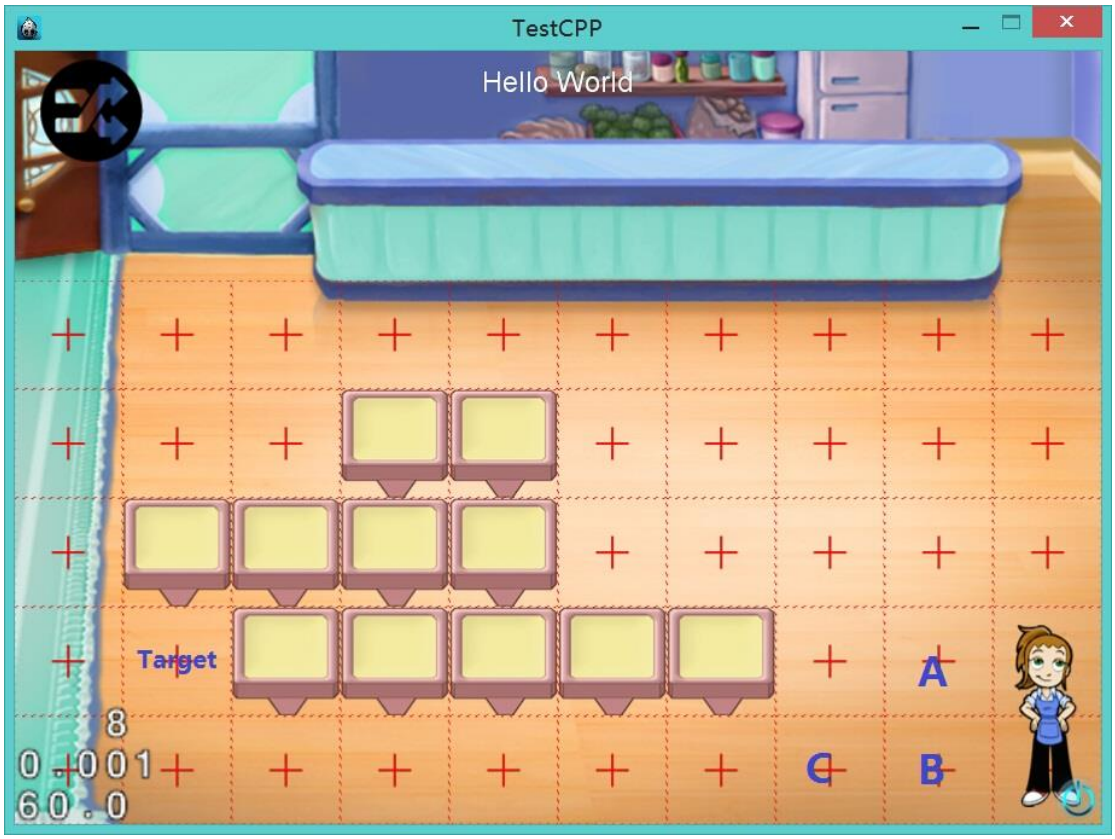
   One thing is worth paying attention:

My A* search eight directions with the different steps, the horizontal and vertical costs two steps while the diagonal costs three steps, thus we can avoid such problem:

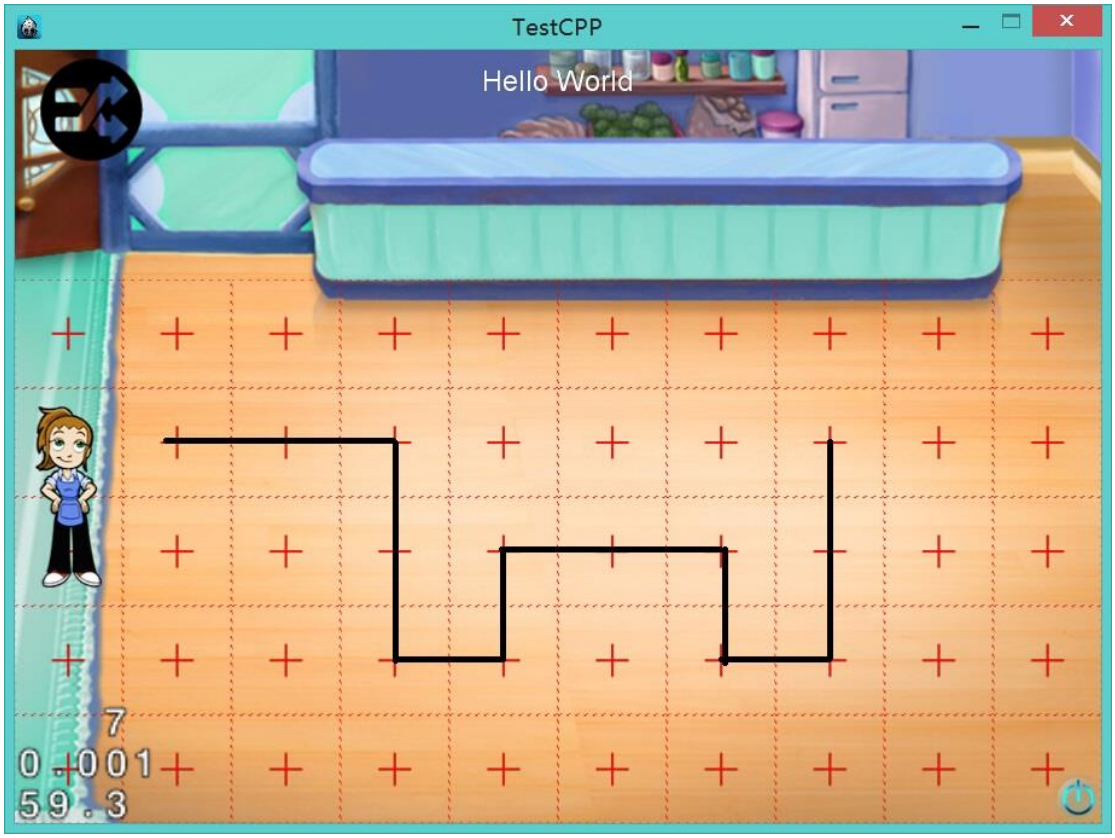
   It takes three steps to get to A, and two steps to get to B.

   If it takes one step both to get to A and B, due to A and C, and all the grids after are closer than the Target, the B won't be searched, so finally our character will go to A, then C, instead of B to C, which seems a little weird.

   (See the picture below.)

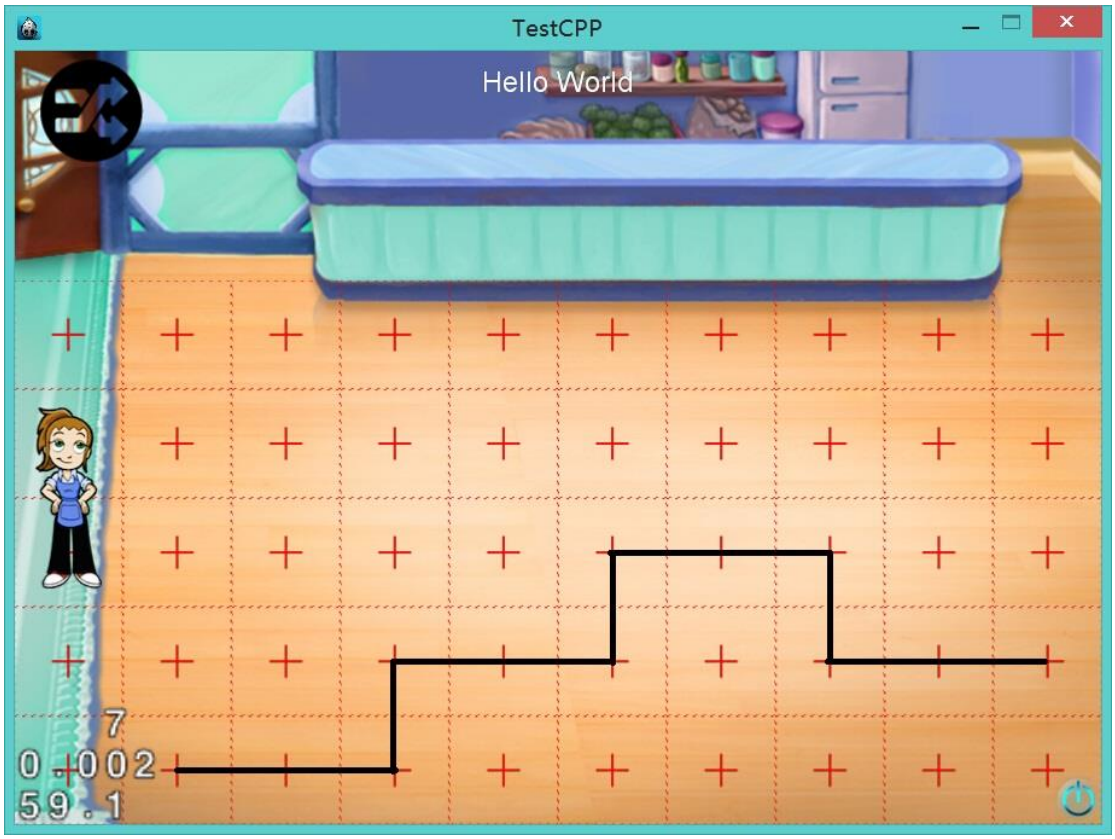

TestCPP
Hello World
Target
A
B
C
8
0.001
60.0

3. To scramble the tables I didn't have many clear ideas, so I picked one that I think would work pretty.
   The basic algorithm is to draw such basic shape:
   As you see in the picture, to draw the tables by horizontal->vertical->horizontal->vertical->…… and so on.

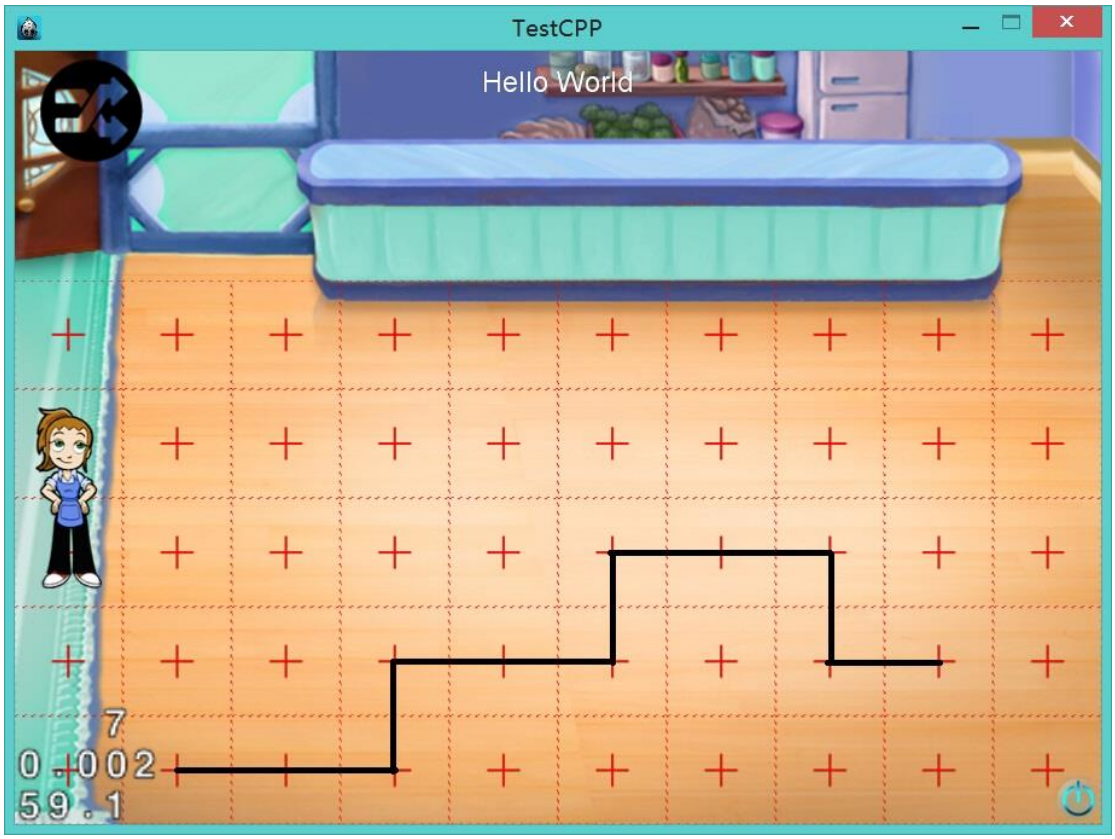


So how to make sure that it's not make an inaccessible area?
What I was designing is, starting from the column 1 instead of 0, and make sure except the first horizontal placement, no other would touch the borders of the map.

So here's a problem, it's easy to control the verticals, and we start from the column 1, so left border won't be touched, but what about the right border? Like below:

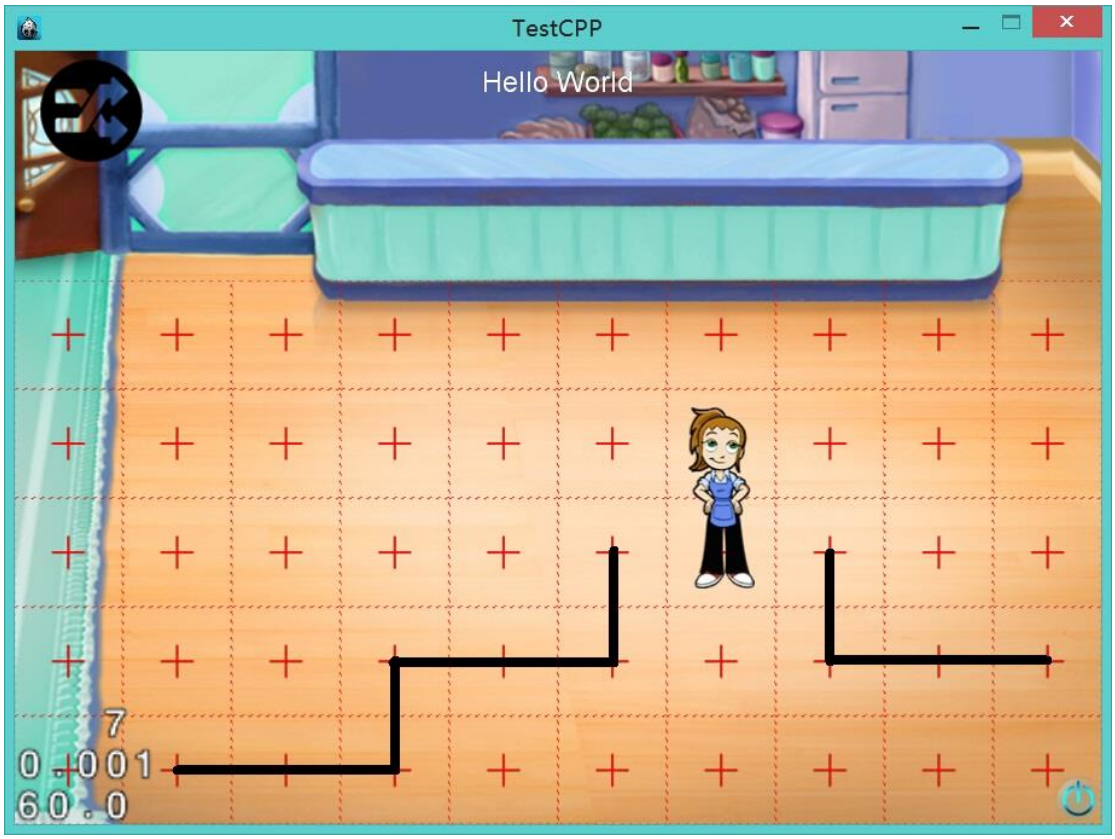


Well, it’s not happening in my code. ☺
Let me analyze the extreme conditions:
The horizontal one goes rand() % 2 + 1, which means its range is [1, 2]
The vertical one goes rand() % (max(distances to border) - 1) + 1, the range is [1, 3]
And total length would be sigma(horizontal) + sigma(vertical),
In order to stretch the length most, we have to minimize the y, and maximize the x.
There are 11 tables in total:



This is the extreme condition, it won't touch right border.

Wait, there's another case:



What if the character is on the road? Then the right border could be touched?

We move the character to the zero point before scrambling the tables, ☺

Thanks,
Yufan Lu