Marcos Remy

GLUT and OpenGL were used to create the Snake game using C++.

```
    //  as static [!??].
    const GLfloat   DH = (GRID_PANE_WIDTH - 2.f)/ numCols,
                    DV = (GRID_PANE_HEIGHT - 2.f) / numRows;
    const GLfloat segMove[4][2] = {
                                    {0, DV},    //  NORTH
                                    {DH, 0},    //  WEST
                                    {0, -DV},   //  SOUTH
                                    {-DH, 0}};  //  EAST

    glPushMatrix();

    //  The first segment is different

    glColor4fv(traveler.rgba);

    glTranslatef((traveler.segmentList[0].col + 0.5f)*DH,
                 (traveler.segmentList[0].row + 0.5f)*DV, 0.f);

    if (traveler.segmentList.size() > 1)
    {

        for (unsigned int currSegIndex=0; currSegIndex<traveler.segmentList.size()-1; currSegIndex++)
        {

            int dirInt = static_cast<int>(traveler.segmentList[currSegIndex].dir);

            //if currentsegment is alive color is good else color is black

            glColor4fv(traveler.segmentList[currSegIndex].rgba);

            //  draw a segment to the center of the next square
            glBegin(GL_LINES);
                glVertex2f(0, 0);
                glVertex2f(segMove[dirInt][0],
                           segMove[dirInt][1]);
            glEnd();
```

Each snake is drawn using OpenGL segments which connect to each other on the grid with the head leading the body.

```cpp
void drawTravelers(void)
{

    //----------------------------
    //  You may have to synchronize things here
    //----------------------------

    std::unique_lock lk(syncLock);

    for (unsigned int k=0; k<travelerList.size(); k++) {

        //Traveler Threading:
        {

            std::lock_guard lk( *travelerList[k].tMutex);
            travelerList[k].ready = true;
            travelerList[k].tCV->notify_one();

        }



        //Check if traveler is alive then render it:
        if (!travelerList[k].finished) {

            drawTraveler(travelerList[k]);

        } else {

            if (travelerThreads[k].joinable()) {

                travelerThreads[k].detach();
                numLiveThreads--;
            }

        }
```

Each snake operates on its own thread and each thread is terminated once it finds the end location. It starts by acquiring a lock guard on the mutex associated with traveler's thread that way other threads can't access or modify the traveler's data at the same time. The ready flag is then set so it can modify its own data. Finally, the condition variable notifies the next thread that it's able to modify data now.

```
void updateTraveler(int k) {

    while (running) {

        std::unique_lock lk(*travelerList[k].tMutex);
        travelerList[k].tCV->wait(lk, [&]() {return travelerList[k].ready;}); //Wait for render thread to be
            done


        syncLock.lock();

        //Update Travelers Segments:
        if (travelerList[k].segAdd < travelerList[k].pathSegments.size()) {

            travelerList[k].segmentList.push_back( travelerList[k].pathSegments.at(travelerList[k].segAdd) );

            if (travelerList[k].segAdd > travelerList[k].numSegs) {

                travelerList[k].segmentList[travelerList[k].segDel].rgba[0] = 0.0f;
                travelerList[k].segmentList[travelerList[k].segDel].rgba[1] = 0.0f;
                travelerList[k].segmentList[travelerList[k].segDel].rgba[2] = 0.0f;

                travelerList[k].segDel++;

            }

            grid[travelerList[k].pathSegments.at(travelerList[k].segAdd).row][travelerList
                [k].pathSegments.at(travelerList[k].segAdd).col] = SquareType::TRAVELER;

            travelerList[k].segAdd++;

        } else {

            if (travelerList[k].pathSegments.size() > 0) {

                if (travelerList[k].finished == false) {

                    travelerList[k].finished = true;
                    numTravelersDone++;

                }

            }

        }

        syncLock.unlock();

        travelerList[k].ready = false;
        lk.unlock();
        travelerList[k].tCV->notify_one();

        }

}
```

This is the code which each thread runs, it adds segments to the traveler's segmentList vector if it's within the size. Then the grid's data is updated so that the game can render the new information on the grid being the new segment.

```
// create the start node and push into list of open nodes
pNode1 = new Node(locStart, 0, 0);
pNode1->calculateFValue(locFinish);
q[qi].push(*pNode1);

// A* search
while(!q[qi].empty()) {
    // get the current node w/ the lowest FValue
    // from the list of open nodes
    pNode1 = new Node( q[qi].top().getLocation(),
                 q[qi].top().getGValue(), q[qi].top().getFValue());

    row = (pNode1->getLocation()).row;
    col = pNode1->getLocation().col;

    // remove the node from the open list
    q[qi].pop();
    openNodes[row][col] = 0;

    // mark it on the closed nodes list
    closedNodes[row][col] = 1;

    // stop searching when the goal state is reached
    if(row == locFinish.row && col == locFinish.col) {

    // generate the path from finish to start from dirMap
        string path = "";
        while(!(row == locStart.row && col == locStart.col)) {

            j = dirMap[row][col];
            c = '0' + (j + NDIR/2) % NDIR;
            path = c + path;
            row += iDir[j];
            col += jDir[j];

        }
```

```
        // garbage collection
        delete pNode1;

        // empty the leftover nodes
        while(!q[qi].empty()) q[qi].pop();
        return path;

    }

    // generate moves in all possible directions
    for(i = 0; i < NDIR; i++) {
        iNext = row + iDir[i];
    jNext = col + jDir[i];

    // if not wall (obstacle) nor in the closed list
        if(!(iNext < 0 || iNext > numRows - 1 || jNext < 0 || jNext > numCols - 1 ||
        aGrid[iNext][jNext] == 1 || closedNodes[iNext][jNext] == 1)) {

    // generate a child node
            pNode2 = new Node( Location(iNext, jNext), pNode1->getGValue(), pNode1->getFValue());
            pNode2->updateGValue(i);
            pNode2->calculateFValue(locFinish);

            // if it is not in the open list then add into that
            if(openNodes[iNext][jNext] == 0) {
                openNodes[iNext][jNext] = pNode2->getFValue();
                q[qi].push(*pNode2);
                // mark its parent node direction
                dirMap[iNext][jNext] = (i + NDIR/2) % NDIR;
            }

    // already in the open list
            else if(openNodes[iNext][jNext] > pNode2->getFValue()) {
                // update the FValue info
                openNodes[iNext][jNext] = pNode2->getFValue();

                // update the parent direction info,  mark its parent node direction
```

```
                dirMap[iNext][jNext] = (i + NDIR/2) % NDIR;

                // replace the node by emptying one q to the other one
                // except the node to be replaced will be ignored
                // and the new node will be pushed in instead
                while(!(q[qi].top().getLocation().row == iNext &&
                    q[qi].top().getLocation().col == jNext)) {
                    q[1 - qi].push(q[qi].top());
                    q[qi].pop();
                }

        // remove the wanted node
                q[qi].pop();

                // empty the larger size q to the smaller one
                if(q[qi].size() > q[1 - qi].size()) qi = 1 - qi;
                while(!q[qi].empty()) {
                    q[1 - qi].push(q[qi].top());
                    q[qi].pop();
                }
                qi = 1 - qi;

        // add the better node instead
                q[qi].push(*pNode2);
            }

            else delete pNode2;
        }
    }
    delete pNode1;
}
// no path found
return "";


id updateTraveler(int k) {
```

Here’s the A* algorithm which I adapted to work with the game. It finds the shortest path between two points on the game’s grid. It does this by taking into account the distance between nodes and any obstacles in the way. Each node is represented with a value and the ones with the lower values are added to the path.