Assignment 5

Due November 20, 2023

Overview

In this assignment, you will add AI for the opponent. Normally the opponent will make a snowball, climb up somewhere high, and then throw it at the player. However, there will also be a debugging command to make the opponent go to a specific location.

Most of the assignment will be implementing A* pathfinding for a world where the piles of blocks have different heights, which will be harder to implement than on a simple grid. Sometimes the opponent will have to jump to get between the blocks. In other places, the height difference is large enough that opponent can go down but not jump back up. To find a path across this map, you will create a connection graph for your block map and perform pathfinding on it.

The Connection Graph

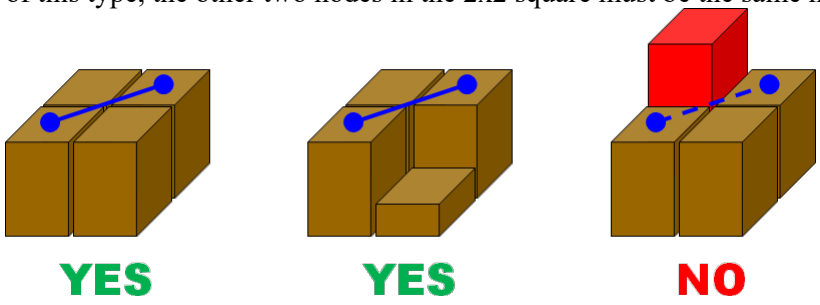
The ***connection graph*** is a graph that shows connectivity between locations on the block map. It will have a node for each cell in the block map. The node will represent the location in the center of the cell at the height of the top of the highest block. If there are no blocks, it will be at ground height. To aid debugging, the nodes will be visualized in the game world as colored dots.

The graph will store directed links indicating which nodes the opponent can reach from which other nodes. Each link will have one of seven connection types, as explained below.

Terminology: The ***4-neighbours*** of a cell on a 2-dimensional grid have one index the same as the cell and the other changed by ±1. The ***diagonal-neighbours*** of a cell have both indexes changed by ±1. Together, the 4-neighbours and the diagonal-neighbours form the set of ***8-neighbours***. These terms are mostly used in image processing.

The seven connection types and the nodes that they lead to are as follows:

- **Adjacent:** A 4-neighbour of the current node at the same height.
- **Up1:** A 4-neighbour of the current node that is exactly 1 unit higher.
- **Down1:** A 4-neighbour of the current node that is exactly 1 unit lower.
- **DownFar:** A 4-neighbour of the current node that is 2 or more units lower.
- **Diagonal:** A diagonal-neighbour of the current node at the same height. To make a connection of this type, the other two nodes in the 2x2 square must be the same height or lower.



- **Jump2:** A node 2 cells away from the current node along one axis. To make a connection of this type, the cell in the middle must be lower. A Jump2 connection will allow the opponent to jump over a pit as wide as one block.

- **Jump3:** Similar to Jump2, except the destination is 3 cells away instead of 2.

It is impossible for the opponent to move directly to a 4-neighbour of the current node that is two or more units higher than it, so there will no links to such nodes. A node at the bottom of a deep pit will have no links leading away from it and a node on top of an isolated spire will have no links leading to it.

Programming Project

Part A: Connection Graph [10 marks]

Requirements

- **[1 mark]** Display something for a connection graph on `[5]` key
- **[2 marks]** Display one node for each cell in the block map grid above the block (or ground)
- **[1 mark]** Display at least one link between two nodes
- **[1 mark]** Display links between adjacent nodes at the same height (4-neighbours)
- **[1 mark]** Display links between diagonal nodes at the same height if neither adjacent node is higher
- **[1 mark]** Display links to adjacent nodes `1` unit of height apart (4-neighbours)
- **[1 mark]** Display links to adjacent nodes `2+` units of height apart (4-neighbours)
- **[1 mark]** Display links to nodes 2 or 3 cells apart in a line if any intervening nodes are lower
- **[1 mark]** Display different links types in different colors

Suggested Approach

1. Do Tutorial 20. Optionally do Tutorial 21 now as well. It builds on Tutorial 20 and you will need it for Part B.
2. Copy the code to generate the connection graph into your game. Update it to generate the graph from the block map.
    - **Hint:** Convert the connection graph to a class. It should have two constructors. The default constructor initializes it to have one node at the origin and no links. The other constructor takes a block map as its only parameter and constructs a connection graph for it.
3. Test your connection graph. Make sure you can create and display it before continuing.
    - **Hint:** Tutorial 20 draws thicker lines to make the links more visible. To draw thicker lines in your game, use:
      ```
      glLineWidth(2.0);  // line width in pixels
      ```
      If you don't want the other lines to be thicker, set `glLineWidth` back to `1.0` after drawing the links.
    - **Hint:** Tutorial 20 also draws larger, circular points:
      ```
      glEnable(GL_POINT_SMOOTH);  // round points
      glPointSize(10.0);  // point diameter in pixels
      ```
4. When adding a node to the connection graph, place it at the center of the cell instead of the minimum corner.
5. Change the `addOneLink` function to add more types of links. It should now add Adjacent, Up1, Down1, and DownFar links.
6. Add another debugging variable to your game to control whether the connection graph is displayed. Toggle it (once) when `[5]` is pressed.
7. Add an `addJump2Link` function to add a Jump2 link between two nodes if appropriate. Call this function from `addNodeLinks` for the four directions.

- **Hint:** You should pass in the XZ coordinates of both the middle cell and the destination cell. This will allow you to easily use the same function for nodes along both of the X and Z axes.

8. Add a similar `addJump3Link` function. Call this function from `addNodeLinks` for the four directions.
9. Add more colors to display the different types of links. Use the same color for Up1 and Down1 links.
    - **Hint:** You can make the links almost any color because the terrain is textured instead of colored with solid colors. The tutorial only uses blue and purple for the links because the other colors are used for the terrain. Do not make the links white or they will be hard to see against the snow.

Part B: Pathfinding [10 marks]

Requirements

- **[1 mark]** Display something for a path on `[5]` at some time (cycle through nothing, graph, path, nothing)
- **[1 mark]** Display some path as a polyline between two nodes from the connection graph
- **[2 marks]** Path follows links in the connection graph
- **[1 mark]** Display different link types on path in different colors
- **[2 marks]** Generate a path from the opponent to the debugging cube on `[T]` (if both are inside the block map)
- **[1 mark]** When displaying a path, display costs (in some fashion) for all nodes considered in the search
- **[2 marks]** Considered nodes are believable for A* (mostly near path, more near start than end)

Suggested Approach

1. Do Tutorial 21.
2. Copy the new code into your game and adapt it to work with your connection graph from Part A.
    - **Hint:** Continue to use the horizontal distance as your heuristic.
    - **Hint:** When displaying the path, remember to set `glLineWidth`.
3. Update the link costs for the connection types. In general, the opponent should try to avoid big drops and jumping over pits. The specific costs are:

Connection Type	Traversal Cost
Adjacent	`1.0`
Up1	`2.0`
Down1	`1.0`
DownFar	`3.0`
Diagonal	`1.4`
Jump2	`4.0`
Jump3	`5.0`

4. Create a variable for a path for the opponent to follow.
5. When the `[T]` key is pressed, create a path from the opponent to the debugging cube in front of the player. If the opponent or the cube is outside the block map, do not generate a path.
    - **Hint:** Start by finding the XZ coordinates of the cube. Then use a lookup array to find the index of the node corresponding to that cell. In Tutorials 20 and 21, the lookup array is a global variable named `node_indexes`.

6. Change the behaviour for the `[5]` key so it cycles through three different display modes. (This is similar to what the `[2]` key does.) The first mode should be to display nothing. The second mode should display the connection graph. The third mode should display the opponent's path.
    - **Hint:** Make sure you handle displaying the path when there is no path.
7. When displaying the opponent's path, also display the total costs from the most recent search. You did this in the tutorial with the `drawSearchValues` function.

Part C: Path Following [5 marks]

Requirements

- **[2 marks]** Opponent follows path when it exists
- **[1 mark]** Opponent usually walks (sliding along ground is fine)
- **[1 mark]** Opponent jumps onto higher blocks and over gaps (when path goes there)
- **[1 mark]** Opponent gives up on path if stuck

Suggested Approach

1. Add an AI update function for the opponent. Call it before the physics update.
2. Create a function to apply horizontal acceleration to the opponent towards a node.
    - **Note:** Applying an acceleration changes the opponent's velocity rather the opponent's position.
    - **Note:** Applying an acceleration is done similarly to how the player accelerates when `[W]` is pressed.
3. Create a function to make the opponent follow a path if possible. First, the function should check if the opponent is within `0.25 m` of the next path node. If so, remove that node from the front of the path. Then accelerate the opponent horizontally towards the next path node.
    - **Hint:** Use the function from the previous step.
4. In the AI update, check if the opponent is inside the block map. If not, the opponent should accelerate towards the center of the block map. If the opponent is in the block map, check if the opponent has a path. If so, the opponent should follow the path.
    - **Note:** At this point, your opponent should be able to follow paths that do not require jumping.
5. Store a yaw rotation for your opponent. In the AI update, check if your opponent is stopped. If not, update the yaw based on the opponent's horizontal velocity.
    - **Hint:** The `cbabe_stand.obj` and `guard_stand.obj` models are oriented to face in the +X direction internally. Thus, you will have to rotate them `90°` from the direction the `<cmath>` library returns:

      ```
      yaw_radians = atan2(velocity.x, velocity.z) - 90.0;
      ```

      OR

      ```
      yaw_radians = velocity.getRotationYSafe() - 90.0;
      ```
6. Improve your path-following function to make the opponent jump when needed. If the opponent is on the ground and the next path link has type Up1, Jump2, or Jump3, the opponent should jump. Give the opponent a vertical velocity of `5.5 m/s` upwards. Do not change the opponent's horizontal velocity.
    - **Note:** At this point, your opponent should be able to follow all paths unless a pit is encountered.
7. Create a counter for how long the opponent has been moving along a single path link. Reset it to `0.0 s` whenever the opponent reaches a new link. If it ever reaches `3.0 s`, assume the opponent has gotten stuck somehow and destroy the path.
    - **Note:** One way the opponent could get stuck is to accidentally jump past the destination node and fall. Another way is to be hit by a snowball at a bad moment.

Part D: Opponent AI [5 marks]

Requirements

- **[1 mark]** Opponent gains snowballs somehow, sometime
- **[1 mark]** Show if opponent has a snowball somehow
- **[1 mark]** Opponent stands still and makes a snowball on snow
- **[1 mark]** Opponent with snowballs makes a path to a high-ish place
- **[1 mark]** Opponent at high-ish place throws snowball at player

Suggested Approach

1. Add a variable to store how many snowballs the opponent has. This will always be `0` or `1`.
2. Add a function to determine the depth of the snow is at a position. For example, if the snow is `2.3 m` above ground level and there are two blocks (total of `2.0 m`), the snow is `2.3 - 2.0 = 0.3 m` deep.
    - **Hint:** If you are using the provided solution, the somewhat-misnamed `Map::calculateSnowHeight` function already returns the snow depth. However, its arguments make it difficult to call (especially the `Neighbourhood` value). Make a new function similar to `Map::isOnSnow` that takes a `Vector3` position as a parameter and calls `getSnowHeight`.
3. Add a function to determine if a node is high-ish. A node is high-ish if (a) it is on top of at least one block (not on the ground), and (b) none of that block's 8-neighbours are higher.
    - **Note:** A high-ish node could be on a local peak or it could be on a flat area.
4. Add a function to find all the nodes that (a) have at least `0.2 m` of snow on them and (b) are not high-ish. Store these nodes in an array or `vector`. These are the nodes where the opponent will make snowballs. Call this function from `init`.
    - **Note:** We want at least `0.2 m` snow instead of "any snow" as a safety margin. The opponent might not stop in the middle of the block, but the opponent should be able to make a snowball anyway. If there is `0.2 m` of snow in the center, there is probably at least some snow elsewhere on the same block.
    - **Note:** We do not want the opponent to make and throw snowballs from the same node. If the opponent can do this, he/she will never have any reason to move. That would make the game less interesting.
5. Expand your function from the previous step to also find all the nodes that (a) are high-ish and (b) have snow at least `0.2 m` below the block surface. These are the nodes where the opponent will throw snowballs from. Store these nodes in a different array or `vector`.
    - **Note:** Again, we want the `0.2 m` as a safety factor. The opponent should not be able to make snowballs there even if he is a bit off-center.
6. Expand the opponent AI function. If the opponent does not have a path or snowball, check if the opponent is standing on snow. If so, the opponent should stay still and make a snowball. This takes `3.0` seconds, just like for the player. If there is no snow, the opponent should make a path to a node where a snowball can be made.
    - **Hint:** Make a counter for the opponent's progress in making a snowball. Update it with the physics updates, as you do for the player's progress. In this case, the AI should tell the opponent to stand still on the snow.
    - **Note:** There will be many nodes suited for making snowballs. You can pick one of them in any way that you want. One way is to choose a node at random:

```
unsigned int index = rand() % snowy_nodes.size();
unsigned int destination = snowy_nodes[index];
```

7. Display whether the opponent has a snowball. One way to do this is in the HUD. Another way is to display a snowball model with the opponent model.
    - **Hint:** The `cbabe_stand.obj` model can hold a snowball of radius `0.1 m` at `(0.25, 0.1, 0.2)` in local coordinates. The `guard_stand.obj` model can hold one at `(0.2, 0.4, 0.18)`. For `cbabe_jump.obj`, the position is `(0.45, 0.75, 0.35)` and for `guard_jump.obj` it is `(0.5, 0.8, 0.45)`.
8. Expand the opponent AI function again. If the opponent does not have a path, but does have a snowball, check if the opponent is standing at a high-ish node. If not, the opponent should make a path to a node from which a snowball can be thrown.
9. Add a function to calculate the pitch (vertical angle) for the opponent to throw a snowball to hit a target. The formula is:

$$\theta = -tan^{-1}\left(\frac{v^2 - \sqrt{v^4 + F_g\left(2yv^2 - F_g x^2\right)}}{F_g x}\right)$$

    where:

    x is horizontal distance (use `Vector3::getDistanceXZ()`, always non-negative)
    y is vertical distance (positive for above, negative for below)
    v is starting speed (always positive)
    F_g is the magnitude of the acceleration due to gravity (negative, since it points down)
    θ is the pitch (in radians, positive is upwards)

    Start by calculating the value that will go inside the square root. If that value is negative, the target is too far away to hit. Return a special value to indicate this. Otherwise, determine and return θ.
    - **Hint:** The tan^{-1} function is available in the `<cmath>` library as `atan`.
    - **Hint:** Remember to convert your angle to degrees (if needed).
    - **Note:** This is adapted from the formula on Wikipedia. For the derivation, see this forum post. In the original, F_g was represented as a positive number.
    - **Note:** You can also solve the equation for v_x, but you will need cos^{-1} (which is known as `acos`) to convert it to an angle and there is no way to know if it should be positive (aim up) or negative (aim down).
10. Add a function to make the opponent throw a snowball at the player with a speed of `15.0 m/s`. The function should return a bool indicating whether or not it succeeded. Start by calculating the required pitch. If the throw is impossible (the special value from previous step) or the pitch is less than `-60°` or greater than `60°`, give up and report failure. Otherwise, throw the snowball at the player, change the opponent's yaw to the direction in which the snowball was thrown, and report success.
    - **Hint:** You can calculate the desired yaw based on the difference in the positions:

```
Vector3 relative = target_position - throw_position;
double yaw_radians = relative.getRotationYSafe();
```

11. Expand the opponent AI function yet again. If the opponent has no path, has a snowball, and is at a high-ish node, the opponent should attempt to throw the snowball at the player. If the throw is impossible because of the pitch angle, the opponent should calculate a path to a new node where another attempt can be made to throw a snowball.
12. Add additional collision checking to detect when flying snowballs hit the player. If a snowball does so, destroy it and knock the player away but do not increase the hit count.
13. Add a member variable to the flying snowball type saying who threw it. Set it when you create a snowball. Snowballs thrown by the player should not hit the player and snowballs thrown by the opponent should not hit the opponent.

Compiling and Running [10 marks]

Requirements:

- **[5 marks]** Program compiles without (significant) compiler errors. Sometimes standard libraries include each other differently on different compilers. If so, the marker will add them as needed.
- **[2 marks]** Program starts without crashing
- **[3 marks]** Program does not crash during testing

Submission

- Submit a complete copy of your source code
  - Include a copy of the `ObjLibrary` (if you used it)
- Don't submit a compiled version
- Don't submit intermediate files, such as:
  - `Debug` folder
  - `Release` folder
  - `ipch` folder
  - `*.ncb`, `*.sdf`, or `*.db` files
  - `*.o` files
- Submit models and textures if they are different from the ones provided on the course website
  - You don't need to submit the files from `Resources.zip` on course website, such as the skybox texture. The marker has those and will add them to your game.