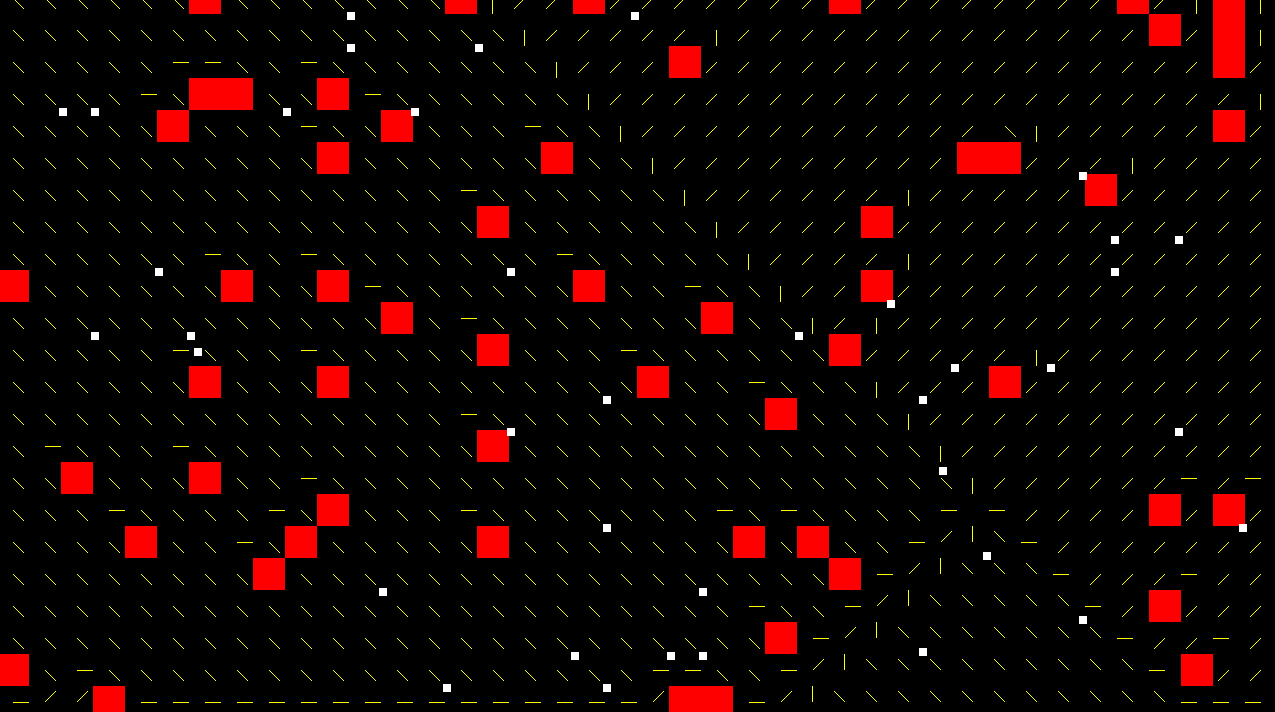
Tutorial – Flow Fields

In this tutorial you will be implementing a simple flow field to move game objects around a 2D grid. The base code will be provided for you, and from there you are expected to expand on it.

Your job is to implement the generation of the flow field and then implement a steering behaviour that samples the flow field to calculate a force for the steering.



Starting Project:

The starter project provided to you is a project that should be added to your aieBootstrap solution.

The project contains code that has three 2D grids of data that overlay each other and store the important data for each grid cell in our virtual space:

* A cost field
* An integration field
* A flow field

The cost field represents costs to travel through the grid cell, with impassable cells being tagged as such. It stores costs as unsigned short. This field is randomly populated by the application, but you may wish to implement an alternative way to populate it.

The integration field is not set up for you. It stores the results of a Dijkstra’s Shortest Path search towards a goal node, using floats, by expanding a search outwards from the goal cell and storing in the integration field the travel cost from the current cell to the goal cell.

Finally the flow field is also not pre-initialised for you. The flow field stores 2D vectors that point towards the neighbouring cell that has the lowest integration cost. The integration costs will be a gradient that descends in value the closer it gets to the goal cell, so the flow field will end up pointing each cell in the direction towards the goal cell.

enum eFlowFieldCosts : unsigned short {

WALKABLE = 1,

IMPASSABLE = 0xffff,

};

enum eFlowFieldSize {

FLOWFIELD\_ROWS = 23,

FLOWFIELD\_COLS = 40,

FLOWFIELD\_CELLSIZE = 32,

};

// contains movement costs through the field

unsigned short m\_costField[FLOWFIELD\_ROWS][FLOWFIELD\_COLS];

// contains actual travel cost to the goal cell

float m\_integrationField[FLOWFIELD\_ROWS][FLOWFIELD\_COLS];

// the vector flow field that travels towards the goal cell

Vector2 m\_flowField[FLOWFIELD\_ROWS][FLOWFIELD\_COLS];

The application also creates a collection of game objects and places them in cells that aren’t impassable.

The application also has a function for randomising the cost field, a function for accessing passable neighbour cells around a specified cell, and then two methods you must implement: **performIntegration()** and **generateFlowfield():**

// percentage of grid taken up by obstacles

float m\_obstaclePercentage = 0.05f;

// randomly generates obstacles and places entities safely

void randomiseLevel(float obstaclePercentage);

// the neighbours parameter will be filled with 2x int for

// every neighbour ([y][x] within the field)

// ignores impassable neighbours

void getCellNeighbours(int x, int y, std::vector<int>& neighbours);

// generate dijkstras to goal cell

void performIntergration(int goalX, int goalY);

// calculate flow based on gScores

void generateFlowfield();

As the comments indicate the **performIntegration()** method is meant to perform a Dijkstra’s search from the specified goal cell and update the integration field travel costs.

The **generateFlowfield()** method is then meant to use the integration field to populate the flow field with directions.

The application is already set up to draw the flow field and game objects, and during update it detects a mouse click to generate a target cell that is then used to update the flow field and integration fields:

// randomise level

if (input->wasKeyPressed(aie::INPUT\_KEY\_R))

randomiseLevel(m\_obstaclePercentage);

// pick goal cell

if (input->isMouseButtonDown(aie::INPUT\_MOUSE\_BUTTON\_LEFT)) {

// find cell under mouse

int x = 0, y = 0;

input->getMouseXY(&x, &y);

// convert screen space to "grid space"

x /= FLOWFIELD\_CELLSIZE;

y /= FLOWFIELD\_CELLSIZE;

if (m\_integrationField[y][x] != eFlowFieldCosts::IMPASSABLE) {

performIntergration(x, y);

generateFlowfield();

}

}

Your job is to first generate the integration field,

Performing Integration:

The integration field is simply a modified Dijkstra’s.

void FlowFieldsApp::performIntergration(int goalX, int goalY) {

// perform a dijkstra's search towards goal cell,

// starting from goal cell spreading outwards

}

We start by resetting the cost of the integration field to be a large number, for example, to **FLT\_MAX**.

Next we need to create an open list for our search. Since we are filling a grid we can store a pair of int within the open list, using a **std::list** containing **std::pair<int, int>.** We push the goal cell’s coordinates onto the list and then set the integration field at that location to have a cost of 0 since it is the goal node.

Since we aren’t using links between graph nodes we need a way to access the neighbouring cells, which is what the method **getCellNeighbours()** does. We pass it the current cell’s coordinates and then a **std::vector** containing 2 int for every neighbour. Since we will be doing this for every cell on the open list we might as well reuse a vector that is already sized to create enough int’s.

With those added to the function we could begin setting up the while loop that Dijkstra’s uses:

You may have noticed that the integration field is being indexed using Y and then X. This is because we’re storing the data in the fields by **ROW** then **COLUMN**.

void FlowFieldsApp::performIntergration(int goalX, int goalY) {

// perform a dijkstra's search towards goal cell,

// starting from goal cell spreading outwards

// reset integration field

for (int r = 0; r < FLOWFIELD\_ROWS; ++r)

for (int c = 0; c < FLOWFIELD\_COLS; ++c)

m\_integrationField[r][c] = FLT\_MAX;

// push goal into open list

std::list<std::pair<int,int>> openList;

openList.push\_front({ goalX, goalY });

m\_integrationField[goalY][goalX] = 0;

std::vector<int> neighbours(16);

// do search

while (openList.empty() == false) {

}

}

Within the while loop, just like Dijkstra’s, we pop a cell from the open list and then query its neighbours. The neighbour **std::vector** will have 2 int for every neighbour, representing the [y][x] indices within the fields.

We determine a potential cost to the neighbour by combining the popped cell’s integration score with the neighbours cost field value. If that value is less than the neighbours current integration score then we update its score. If it is also not within the open list we add it to the list, or add it back to the list in the cases where we found a quicker way to get there.

The following code goes within the while loop at the end of the last snippet:

And that’s it for integration. Next we generate the flow field.

auto current = openList.front();

openList.pop\_front();

// get neighbours of popped cell and iterate over them

getCellNeighbours(current.first, current.second, neighbours);

unsigned int neighbourCount = neighbours.size();

for (unsigned int i = 0; i < neighbourCount; i += 2) {

// calculate new travel cost

// current is a std::pair<int,int> representing x and y

float cost = m\_integrationField[current.second][current.first] +

m\_costField[neighbours[i + 1]][neighbours[i]];

// compare if score was lower

if (cost < m\_integrationField[neighbours[i + 1]][neighbours[i]]) {

// create a new std::pair for the neighbour

std::pair<int, int> neighbour = { neighbours[i],

neighbours[i + 1] };

// check if they should be added to the open list

auto iter = std::find(openList.begin(),

openList.end(), neighbour);

if (iter == openList.end()) {

openList.push\_back(neighbour);

}

// update the cost

m\_integrationField[neighbours[i + 1]][neighbours[i]] = cost;

}

}

Calculating the Flow Field:

You now need to implement the **generateFlowfield()** method:

void FlowFieldsApp::generateFlowfield() {

// for each grid cell, sample neighbour costs and

// setup flow direction to lowest cost neighbour

}

There are various ways to generate a flow field, but one simple way is to simply iterate over the integration field. For each cell, find which neighbour cell has the lowest integration score. Whichever it is we set the flow field vector to point towards it and normalise it. It’s that simple.

If no neighbours are found then the flow field vector should be zero.

We again use a **std::vector<int>** for collecting neighbours.

Check the next code snippet to see how this task can be completed.

void FlowFieldsApp::generateFlowfield() {

// for each grid cell, sample neighbour costs and

// setup flow direction to lowest cost neighbour

std::vector<int> neighbours;

int lowestX, lowestY;

float lowestScore;

bool foundNeighbour = false;

for (int r = 0; r < FLOWFIELD\_ROWS; ++r) {

for (int c = 0; c < FLOWFIELD\_COLS; ++c) {

// reset field

m\_flowField[r][c].x = 0;

m\_flowField[r][c].y = 0;

// reset lowest score

lowestScore = FLT\_MAX;

// gather neighbours

getCellNeighbours(c, r, neighbours);

unsigned int neighbourCount = neighbours.size();

// only set a vector if neighbours found

foundNeighbour = false;

if (neighbourCount > 0) {

for (unsigned int i = 0; i < neighbourCount; i += 2) {

int nx = neighbours[i];

int ny = neighbours[i + 1];

// is it lowest?

if (m\_integrationField[ny][nx] < lowestScore) {

lowestScore = m\_integrationField[ny][nx];

lowestX = nx;

lowestY = ny;

foundNeighbour = true;

}

}

// if valid neighbour found, normalise direction

if (foundNeighbour) {

float mag = float((lowestX - c) \* (lowestX - c) +

(lowestY - r) \* (lowestY - r));

if (mag > 0) {

mag = sqrt(mag);

m\_flowField[r][c].x = (lowestX - c) / mag;

m\_flowField[r][c].y = (lowestY - r) / mag;

}

}

}

}

}

}

If you have followed along then you will now have successfully implemented the code needed to update a flow field based off of a cost field.

Now we can make use of the field within a steering behaviour.

Creating the FlowForce Steering Behaviour:

You will need to have completed the SteeringBehaviour lessons to be able to complete this.

We can simply create a new force object, **FlowForce**, that inherits from **SteeringForce**. Once implemented we simply add it to a **SteeringBehaviour** and add the behaviour to all of our game objects which already have velocity settings applied.

class FlowForce : public SteeringForce {

public:

FlowForce() {}

virtual ~FlowForce() {}

void setFlowField(Vector2\* flowfield, int rows, int cols, float cellSize) {

m\_flowfield = flowfield;

m\_rows = rows;

m\_cols = cols;

m\_cellSize = cellSize;

}

virtual Force getForce(GameObject\* gameObject) const;

protected:

int m\_rows, m\_cols;

float m\_cellSize;

Vector2\* m\_flowfield = nullptr;

};

The force needs to have access to the flow field and needs to know some information about it, such as cell size and how many columns there are. This is so that we can take the position of a game object and convert it to a grid coordinate so that we can access the velocity of the cell the game object is within.

Implementing **getForce()** simply becomes a matter of determining which cell it is and then return the flow field direction.

During our application’s **startup()** we simply set up the FlowForce with the flow field’s data.

Force FlowForce::getForce(GameObject\* gameObject) const {

if (m\_flowfield == nullptr)

return{ 0,0 };

// find cell we're in

float x, y;

gameObject->getPosition(&x, &y);

int ix = int(x / m\_cellSize);

int iy = int(y / m\_cellSize);

// need a 1D index

int index = iy \* m\_cols + ix;

float maxForce = 0;

gameObject->getBlackboard().get("maxForce", maxForce);

return{ m\_flowfield[index].x \* maxForce, m\_flowfield[index].y \* maxForce };

}

You will need to ensure the steering behaviour is attached to the game objects, but that should be all there is to getting basic flow fields working!

Exercises:

1. This implementation suffers from a few small issues, which include:  
   Diagonal travel cost is considered the same as vertical / horizontal  
   Diagonals sometimes should be blocked by obstacles but aren’t.  
   Attempt to solve these issues.
2. Other methods exist for populating the flow field. One involves sampling the gradient of the integration field in only vertical and horizontal directions, ignoring diagonals.

|  |  |  |
| --- | --- | --- |
|  | **T** |  |
| **L** | **C** | **R** |
|  | **B** |  |

We then set the flow field vector at **C** using the following formula:

m\_flowField[y][x].x = L – R

m\_flowField[y][x].y = B – T

We still normalise the vector afterwards.

Try this method. What do you find? What improvements could be made?

1. Sometimes the FlowForce steering behaviour needs access to all vectors around the current one and blends them together. Try to implement this feature.