Shortest paths, A* and Bresenham's algorithm

Simon Kågström

ska@bth.se

Department of Systems and Software Engineering

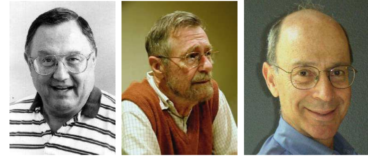
School of Engineering

Blekinge Institute of Technology

P.O. Box 520, SE-372 25 Ronneby, Sweden

Outline

- Where are all gaming algorithms?
- Many "game algorithms" are just applied standard algorithms
- We will look at Dijkstra's algorithm for shortest paths, A* and Bresenham's algorithm for line plotting



- Bresenham? Hart? Dijkstra?
- If time allows, we will watch a movie!

Path finding

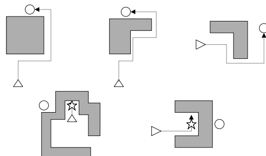
- Path finding can be quite hard

Path finding, II

- Good path finding is crucial for nice-looking AI in many games
- The problem:
  - How to get from a point **A** to a point **B** while avoiding obstacles
  - (and making the path as short as possible)
- Since game AI should look good and realistic, a good path is better than a *shortest* path
- There are many different methods of achieving path finding

Path finding, crash-and-turn

- *crash-and-turn*-based path finding is a simple algorithm to find *a* path:
  1. Start moving in the direction towards the target
  2. If something blocks your way, walk along that (either way)
  3. If it no longer blocks the way then goto 1
- Features:
  - Runs fast, scales well, uses constant memory
  - Looks OK (i.e. it looks like the NPC tries to find a way)
  - **BUT**: With concave obstacles the algorithm can get stuck

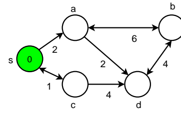


Path finding, navigation meshes

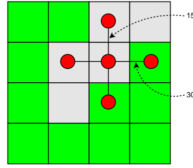
- The standard gaming way around problems: *cheat*
- We can provide hints to the NPCs in the form of navigation meshes
  - Basically provide fixed paths where NPCs can move
- This way we either get away without path finding or with simple ditto (few nodes)
- Typical example could be a racing game where the NPCs try to stay as close as possible to a predefined "route"

Some terminology

- We will look at two kinds of graphs here



1. "Traditional"
   - Nodes V and edges E
   - Edges have weights
2. Tilemap-based
   - Each tile is a node in the graph
   - Edges are implicit between the nodes



- You will use tilemap-based in lab 4

Shortest paths

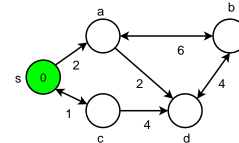
- Types of algorithms
  - Single-source shortest path algorithms: **Dijkstra**, **Bellman-Ford**
  - All-pairs shortest path algorithms: **Floyd-Warshall**
  - Single-pair shortest path algorithms: **A***
- We will focus on single-source and single-pair algorithms
- The single-source algorithms can be generalized to solve the all-pairs and single-pair problems as well
  - It implicitly finds single-pairs
  - Run it for every source to find all-pairs

Single-source shortest paths, how?

- The shortest path algorithms has many things in common
- They work on weighted, directed graphs $G = (V, E)$
  - V is the set of vertices, E the set of edges
- They use a weight function $w : E \to \mathbb{R}$ which associates a weight with an edge between two vertices
- The goal is to traverse the graph from a start vertex s, assigning a *label* to each vertex
  - A label for the *weight* d of the node and it's predecessor π
  - d is an upper bound of the distance from s to the vertex (best so far)
  - π tells us where we came from
  - A *label-setting* algorithm (Dijkstra) sets each label exactly once
  - A *label-correcting* algorithm (Bellman-Ford) may change the label

Single-source shortest paths, how II?

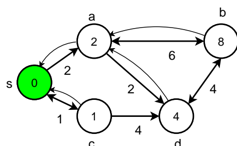
- An example is shown below



- First we have the graph with vertices V and edges E with weights

Single-source shortest paths, how II?

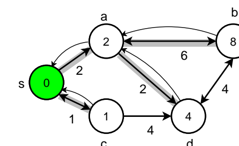
- An example is shown below



- First we have the graph with vertices V and edges E with weights
- A shortest path tree (from s)

Single-source shortest paths, how II?

- An example is shown below



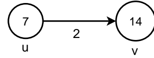
- First we have the graph with vertices V and edges E with weights
- A shortest path tree (from s)
- Labels added to the vertices (predecessor π and weight from s)

Relaxation

- Edges are relaxed during the execution of a single-source shortest path algorithm
- For this, the RELAX function is used

RELAX(u, v, w)

1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$



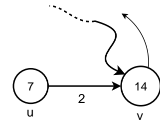
- Relaxation updates the label of v *if* the path from s becomes shorter if passing through u
- In the book, this is the `if currDist(u) > currDist(v)` . . . on page 379

Relaxation

- Edges are relaxed during the execution of a single-source shortest path algorithm
- For this, the RELAX function is used

RELAX(u, v, w)

1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$



- Relaxation updates the label of v *if* the path from s becomes shorter if passing through u
- In the book, this is the `if currDist(u) > currDist(v)` . . . on page 379

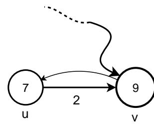
Shortest paths, A* and Bresenham's algorithm – p. 11/39

Relaxation

- Edges are relaxed during the execution of a single-source shortest path algorithm
- For this, the RELAX function is used

RELAX(u, v, w)

1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$



- Relaxation updates the label of v *if* the path from s becomes shorter if passing through u
- In the book, this is the `if currDist(u) > currDist(v)` . . . on page 379

Shortest paths, A* and Bresenham's algorithm – p. 11/39

Initialization

- The initialization of a single-source shortest path algorithm sets all labels'
  - Distance d to infinity
  - Predecessor π to NIL (NULL)

  INITIALIZE-SINGLE-SOURCE(G, s)

  1 **for** each vertex $v \in V[G]$
  2 **do** $d[v] \leftarrow \infty$
  3 $\pi[v] \leftarrow$ NIL
  4 $d[s] \leftarrow 0$
- This is the `for all vertices v ...` on page 378/379

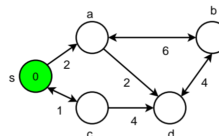
Dijkstras algorithm

- General idea:
  - Two sets, Q, all vertices *to be checked*, S, the set of vertices which *have been determined*
  - Select the node u with the lowest shortest-path estimate
  - Relax all edges leaving u

DIJKSTRA(G, w, s)

1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$
4 **while** $Q \neq \emptyset$
5 **do** $u \leftarrow$ EXTRACT-MIN(Q)
6 $S \leftarrow S \cup \{u\}$
7 **for** each vertex $v \in Adjacent[u]$ and $v \in Q$
8 **do** RELAX(u, v, w)

Dijkstra, II



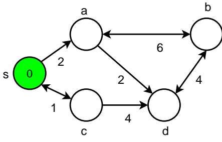
DIJKSTRA(G, w, s)

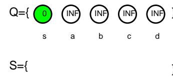
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$
4 **while** $Q \neq \emptyset$
5 **do** $u \leftarrow$ EXTRACT-MIN(Q)
6 $S \leftarrow S \cup \{u\}$
7 **for** each vertex $v \in Adjacent[u]$ and $v \in Q$
8 **do** RELAX(u, v, w)

RELAX(u, v, w)

1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$

Dijkstra, II





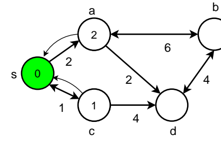
DIJKSTRA(G, w, s)

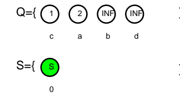
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$
4 **while** $Q \neq \emptyset$
5 **do** $u \leftarrow$ EXTRACT-MIN(Q)
6 $S \leftarrow S \cup \{u\}$
7 **for** each vertex $v \in Adjacent[u]$ and $v \in Q$
8 **do** RELAX(u, v, w)

RELAX(u, v, w)

1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$

Dijkstra, II





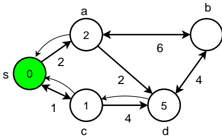
DIJKSTRA(G, w, s)

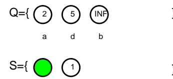
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$
4 **while** $Q \neq \emptyset$
5 **do** $u \leftarrow$ EXTRACT-MIN(Q)
6 $S \leftarrow S \cup \{u\}$
7 **for** each vertex $v \in Adjacent[u]$ and $v \in Q$
8 **do** RELAX(u, v, w)

RELAX(u, v, w)

1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$

Dijkstra, II





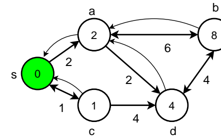
DIJKSTRA(G, w, s)

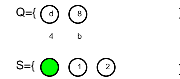
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$
4 **while** $Q \neq \emptyset$
5 **do** $u \leftarrow$ EXTRACT-MIN(Q)
6 $S \leftarrow S \cup \{u\}$
7 **for** each vertex $v \in Adjacent[u]$ and $v \in Q$
8 **do** RELAX(u, v, w)

RELAX(u, v, w)

1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$

Dijkstra, II





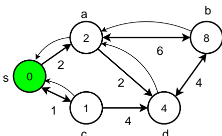
DIJKSTRA(G, w, s)

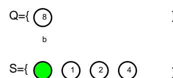
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$
4 **while** $Q \neq \emptyset$
5 **do** $u \leftarrow$ EXTRACT-MIN(Q)
6 $S \leftarrow S \cup \{u\}$
7 **for** each vertex $v \in Adjacent[u]$ and $v \in Q$
8 **do** RELAX(u, v, w)

RELAX(u, v, w)

1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$

Dijkstra, II





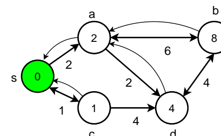
DIJKSTRA(G, w, s)

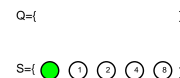
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$
4 **while** $Q \neq \emptyset$
5 **do** $u \leftarrow$ EXTRACT-MIN(Q)
6 $S \leftarrow S \cup \{u\}$
7 **for** each vertex $v \in Adjacent[u]$ and $v \in Q$
8 **do** RELAX(u, v, w)

RELAX(u, v, w)

1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$

Dijkstra, II





DIJKSTRA(G, w, s)

1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$
4 **while** $Q \neq \emptyset$
5 **do** $u \leftarrow$ EXTRACT-MIN(Q)
6 $S \leftarrow S \cup \{u\}$
7 **for** each vertex $v \in Adjacent[u]$ and $v \in Q$
8 **do** RELAX(u, v, w)

RELAX(u, v, w)

1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$

Dijkstra, analysis

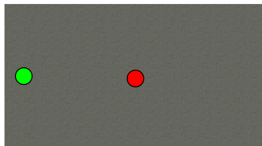
- The complexity of Dijkstras algorithm depends on the choice of set-representation for Q
  1. Each vertex is inserted into S exactly once
  2. Each edge (because of 1) is examined exactly once
- $O(V + E)$, but it now depends on the cost of removing a vertex from Q
  - With an unsorted array or linked list, we get $O(V^2 + E)$ ($O(V)$ to lookup the cheapest node)
  - A binary heap gives $O(ElgV)$, and a fibonacci heap $O(VlgV + E)$
- Negative edges present a problem
  - Dijkstra's algorithm gives an incorrect result
  - In real-world path finding terms, what is a negative edge?
- Negative cycles breaks most algorithms
  - Why?

A*, introduction

- A* is an algorithm for general problem solving
- It is often used for path finding, but is also applicable in other areas
- How do you solve a problem?
  - By exploring possible alternatives
  - Does the alternative solve the problem?
    - If it does, we found a solution
- It usually makes sense to try the "most probable" solution first
  - This is what A* tries to do

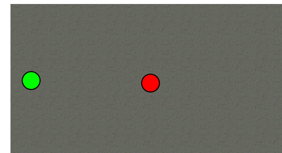
A*, path finding

- Nighttime. We stand in the middle of a giant parking lot without cars



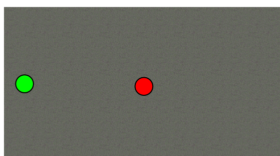
- We want to get from red to green, how would Dijkstra's algorithm solve this?
- How would you solve it (assuming you have a compass)?

A*, path finding



- What do you think A* tries to do?

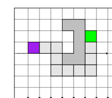
A*, path finding



- What do you think A* tries to do?
  - It tries the "most probable" choice first
  - In this case it tries the direction which minimizes the estimated remaining distance first
  - I.e., just like you it looks to the west first
  - Dijkstra employs a *breadth-first* method, A* uses a *best-first* method

A*, terminology

- A* finds *a* shortest path between purple and green



- Some terminology:
  - *base* (origin), *destination*
  - *movement rules* (How do we get from one node to another)
  - *Heuristics* for the "goodness" of a particular movement
    - E.g. the estimated distance to the goal node
  - *Cost*: The cost to travel from one node to another
    - For instance, cheaper on paved roads then on dirt roads

A*, II

- Basic idea: Look at the most promising candidates first
  - How do we do that?
- The algorithm expands nodes depending on the valid moves
- Each node gets a score (how suitable is it to solve our problem?)
- The score for a node is computed with

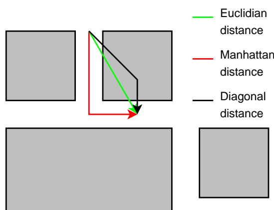
$$f(node) = g(node) + h(node)$$

- Where $g(node)$ is the accumulated cost to get to the node and
- $h(node)$ is the estimated cost to the goal
- The algorithm selects the node with the best score and expands that
- The first expanded node that is the goal node is guaranteed to be the *shortest path*
  - If we provide a (optimistic) heuristic that *underestimates* the destination to the goal node

Heuristics

- There are different ways of approximating $h(node)$:
  1. Manhattan distance, $d_{manhattan} = D(\Delta x + \Delta y)$
     - D is the minimal cost to move between two nodes
     - Manhattan distance is optimistic, so we can guarantee shortest paths
  2. Diagonal distance, $d_{diagonal} = D(max(\Delta x, \Delta y))$
     - If you allow diagonal (8-way) movement
     - Note that D should then have different values for horizontal or vertical vs diagonal movement
  3. We can also use euclidian distance, $d_{euclid} = D\sqrt{\Delta x^2 + \Delta y^2}$
     - If you allow movement along arbitrary angles

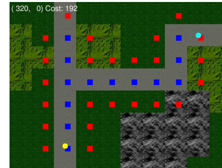
Heuristics, II



- Manhattan distance: $d_{manhattan} = D(\Delta x + \Delta y)$
- Diagonal distance: $d_{diagonal} = D(max(\Delta x, \Delta y))$
- Euclidian distance: $d_{euclid} = D\sqrt{\Delta x^2 + \Delta y^2}$

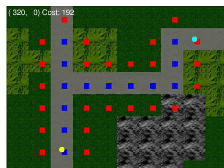
Algorithm overview

- A* uses two sets of nodes, *Open* and *Closed*
  - *Open* contains the nodes that are to be checked
  - *Closed* contains the already checked nodes
- Nodes in the *Closed* set can be re-opened
- The *Open* set should support fast retrieval of the lowest f-node
- Looking up nodes in *Closed* should be fast



Algorithm overview, II

- Basic algorithm (repeat until goal is found):
  1. Get the node with lowest f from *Open*, put in *Closed*
  2. Expand the neighbors of that node, calculate f and insert into *Open*



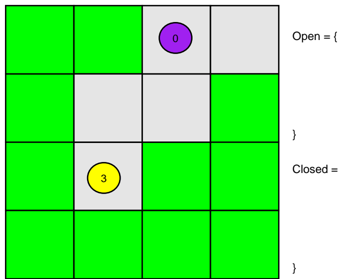
- The red nodes are in the *Open* set, the blue in the *Closed*

Implementation sketch

```
ASTAR-SEARCH(G, start, end)
 1  g[start] ← 0
 2  f[start] ← g[start] + HEURISTIC(start, end)
 3  Open ← Open ∪ {start}
 4  while node ← EXTRACT-MIN(Open)
 5       do if node = end
 6             then return CONSTRUCT-PATH(node)
 7          for each neighbor succ ∈ Adjacent[node]
 8              do newg ← g[node] + COST(node, succ)
 9                 if succ ∈ Open ∪ Closed and g[succ] ≤ newg
10                    then continue with next neighbor
11                 π[succ] ← node
12                 g[succ] ← newg
13                 f[succ] ← g[succ] + HEURISTIC(succ, end)
14                 if succ ∈ Closed
15                    then remove succ from Closed
16                 Open ← Open ∪ {succ}
17          Closed ← Closed ∪ {node}
```

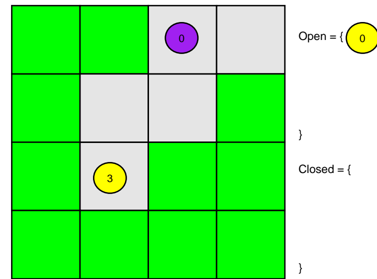
A*, execution

FIXME: Fix these figures!



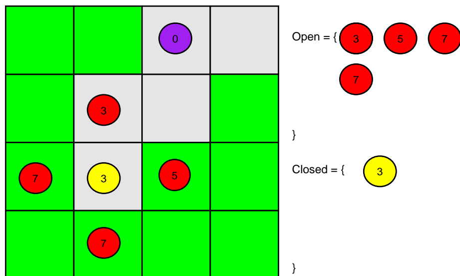
A*, execution

FIXME: Fix these figures!



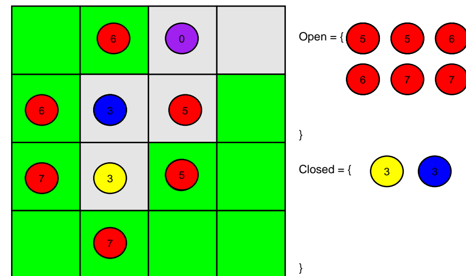
A*, execution

FIXME: Fix these figures!



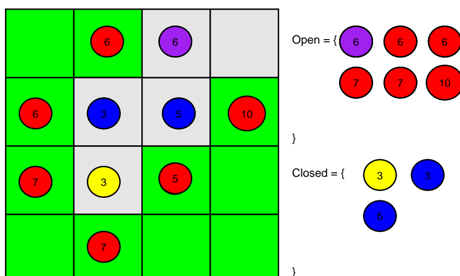
A*, execution

FIXME: Fix these figures!



A*, execution

FIXME: Fix these figures!



A*, some implementation notes

- Some tips for the lab implementation:
  1. You can store the g-value $g[node]$ for a node in the node, i.e., `node->g`
     - (Same with f)
  2. HEURISTIC can be implemented as Manhattan distance between the nodes (i.e., distance between the tiles)
  3. The predecessor π is a pointer to the previous node
  4. You might find the `private` void pointer in the `finder_node_t` useful
  5. The *Open* set should provide fast extraction of the lowest-f node and quick insertion (look in STL for possible set representations)
  6. Have a look at the example implementation in `pathFinder.cc`

Performance aspects

- Unfortunately, A* *can* quickly eat up all your CPU cycles, ruin your game predictability, waste all your memory and chase away all your potential game players
- The performance of A* will vary with the structure of your map
  - No/few obstacles: good performance
  - Short distance start to goal: good performance
  - Labyrinths: bad performance
  - Long distance start to goal: bad performance
- Calculating a new path for your NPCs every frame will be prohibitly expensive, and probably make your game unplayable
- We might need some optimizations

A*, variations

- There are many variations on A* (see `http://www-cs-students.stanford.edu/~amitp/gameprog.html`)
- **Multithreading**: Calculate paths in a background thread continuously
- **Early exit**: Exit with a partial path from A*, which might be good enough
- **Interruptible**: Store the state and continue later
- **Group movement**: In strategy games, instead of running A* for every NPC, run it for one and have the others follow this one
  - The *boids* algorithm can be helpful then.
- **Beam search**: Fixed size of the *Open* set, discarding the worst node when adding a new node
  - The *Open* set must be sorted

A*, variations II

- **Dynamic weighting**: Less weight to the heuristic closer to the goal
  - $f = g + w(p) * h$, where $w \geq 1$ and w decreases closer to the goal
  - The search will first focus on getting closer, while trying harder in the end
- **Iterative deepening**: Cutoff the search after a certain f-value, increasing cutoff after a while
- **Region-based A***: Divide your playfield into connected convex areas, running A* between them and crash-and-turn within them
- ...

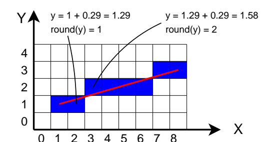
FIXME: include Bresenham?

Drawing lines

- Bresenham's algorithm draws contiguous lines on pixel-based screens
  - As a side-note, there were (are?) computers and screens which were not based on a pixel-matrix (google for Vectrex)
- The line-equation $y = mx + h$, where m is the slope, h is the y-offset ("height"), can be used for drawing lines (the DDA algorithm)
- m is defined as $m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$ where $0 \leq m \leq 1$
- With this, we can then iterate through $x_1 < x < x_2$ and increase y by m each iteration (and place pixels there)

The DDA-algorithm for drawing lines

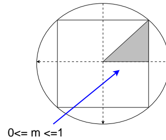
```
y = y1;
for (int x=x1; x < x2; x++) {
  y += m;
  putpixel(x, round(y));
}
```



- In the figure:
  - $\Delta x = 8 - 1 = 7$
  - $\Delta y = 3 - 1 = 2$
  - $m = \frac{\Delta y}{\Delta x} = \frac{2}{7} \approx 0.29$

The DDA algoirthm, problems

```
y = y1;
for (int x=x1; x < x2; x++) {
  y += m;
  putpixel(x, round(y));
}
```



- The DDA algorithm only works for slopes $0 \le m \le 1$
- Easy to modify:
  - For slopes $m > 1$, iterate over y instead (updating x in the same way)
  - If $x_1 > x_2$, swap x_1 and x_2
- However: m is float, which are generally more expensive to work with than integers

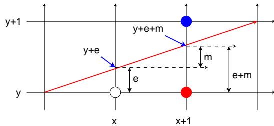
Bresenham's algorithm

- Who is Bresenham really and how does his algorithm work?
  - Bresenham's algorithm was published in the paper "Algorithm for Computer Control of a Digital Plotter" 1965

`http://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html`

- Bresenham's algorithm rasterizes lines using only cheap integer operations
- DDA can paint each pixel many times, Bresenham only once

Bresenham, II



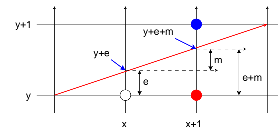
- If we increase x each iteration, we *must* place the next pixel at $(x+1, y)$ or $(x+1, y+1)$
- $(x+1, y)$ is selected if $\epsilon + m < 0.5$, $(x+1, y+1)$ otherwise
- ϵ, the error from the y coordinate of the pixel, is $-0.5 < \epsilon < 0.5$. This is updated in every iteration as

$$\epsilon_{new} \leftarrow \begin{cases} (y+\epsilon+m) - y, & \epsilon + m < 0.5 \\ (y+\epsilon+m) - (y+1), & otherwise \end{cases}$$

Bresenham, III

- By multiplying by $2\Delta x$ we change the inequality $\epsilon + m < 0.5$ to $2\epsilon\Delta x + 2\Delta y < \Delta x$ (remember that m is $\frac{\Delta y}{\Delta x}$)
- We can then substitute $\epsilon\Delta x$ for ϵ' which gives the inequality $2(\epsilon' + \Delta y) < \Delta x$
- The update rule on ϵ' form become

$$\epsilon'_{new} \leftarrow \begin{cases} \epsilon' + \Delta y, & 2(\epsilon' + \Delta y) < \Delta x \\ \epsilon' + \Delta y - \Delta x, & otherwise \end{cases}$$



Bresenham, implementation

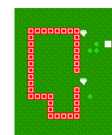
- The algorithm is shown below for $0 \le m \le 1$

BRESENHAM(x_1, y_1, x_2, y_2)
1 $y \leftarrow y_1$
2 $\epsilon' \leftarrow 0$
3 **for** $x \leftarrow x_1$ **to** x_2
4 **do** PUT-PIXEL(x, y)
5 **if** $2(\epsilon' + \Delta y) < \Delta x$
6 **then** $\epsilon' \leftarrow \epsilon' + \Delta y$
7 **else** $y \leftarrow y + 1$
8 $\epsilon' \leftarrow \epsilon' + \Delta y - \Delta x$

- Note that we get by with *one* multiplication with a constant only
- `utils.c` contains an efficient implementation of Bresenham's algorithm
  - This works for arbitrary slopes, which complicates things

Bresenham, applications

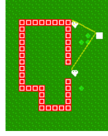
- Line-drawing is only one application of Bresenham's algorithm, it can be used for other things as well
- "Path finding"
- Line-of-sight



- The implementation in `utils.c` takes a callback which handles the plotting
- What the algorithm does at each step (e.g., plotting pixels) is specified in the callback

Bresenham, applications

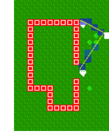
- Line-drawing is only one application of Bresenham's algorithm, it can be used for other things as well
- "Path finding"
- Line-of-sight



- The implementation in `utils.c` takes a callback which handles the plotting
- What the algorithm does at each step (e.g., plotting pixels) is specified in the callback

Bresenham, applications

- Line-drawing is only one application of Bresenham's algorithm, it can be used for other things as well
- "Path finding"
- Line-of-sight



- The implementation in `utils.c` takes a callback which handles the plotting
- What the algorithm does at each step (e.g., plotting pixels) is specified in the callback

Questions?

Questions?