

Algorithms and performance for games

Simon Kågström

Department of Systems and Software Engineering
Blekinge Institute of Technology
Ronneby, Sweden

<http://www.ipd.bth.se/ska>

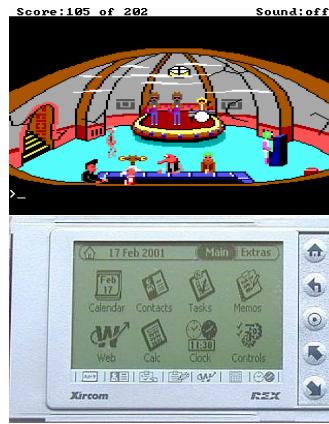


Simon Kågström (BTH)

Algorithms for games

29th June 2006 1 / 96

Post Mortem - REX Adventure



- One of my favourite game series is the Space Quest graphical adventure games
- At the time, I was very much into programming for the REX 6000
- ... So naturally I wanted to make something similar on the REX
- Things like **“Use bottle on coins”** should be possible to represent
- http://spel.bth.se/index.php/Ska:REX_Adventure

Simon Kågström (BTH)

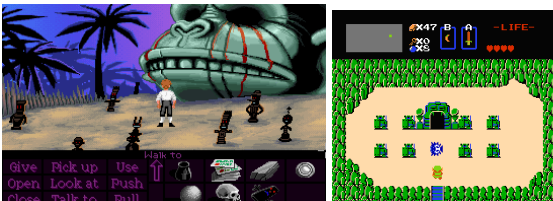
Algorithms for games

29th June 2006 5 / 96

REX Adventure, design

Limitations

- On the REX, you only have 8KB to play with in terms of code and data
- Graphical adventures normally use drawn backdrops
 - Doesn't fit on the REX - (240*120/8), 3KB per image
- Space quest uses keyboard input
 - The REX has a touch screen
- Monkey island-style mouse/touchscreen input, tile-based, Zelda-style graphics

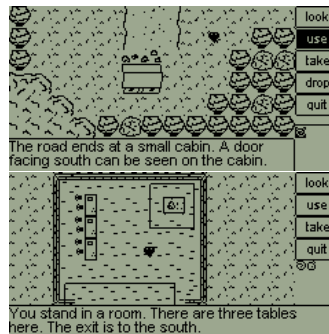


Simon Kågström (BTH)

Algorithms for games

29th June 2006 6 / 96

How it turned out



- I split the game into two parts, a generic library and the actual game
- This overcomes the 8KB limit - 16KB :-)
- The result is basically successful, “use rope on tree”, “look at coins” etc. possible
- The goal of the game is to remove a greedy troll from its rock

Simon Kågström (BTH)

Algorithms for games

29th June 2006 7 / 96

4 things that went good

Description

- The library/game implementation split turned out good, 8KB limit no problem
- The event-based API can produce fairly compact code
- Screens are loaded from textfiles in a simple format
- The game interface feels logical

Code

```
static uint8_t handle_scene_cabin(advgame_t *p_game, adv_event_t event) {
    switch (adv_event_nr(event)) {
        case ADVG_EVENT_ENTER_SCENE:
            /* Add the garden and the door as events */
            advg_init_scene_event(&p_game->scene_events[0], 88, 60, 16, 8, CABIN_DOOR_EVENT);
            ...
        case CABIN_DOOR_EVENT:
            if (p_game->action == ADVG_ACTION_LOOK) /* Player looks at the door */
                advg_display_message(p_game, EV_HOUSE_DOOR_LOOK, 1);
            if (p_game->action == ADVG_ACTION_USE &&
                p_game->obj_use == OBJ_KEY) /* The door is unlocked - enter! */
                advg_goto_scene(p_game, SCENE_INSIDE_CABIN, 120, 60);
            ...
    }
}
```

Simon Kågström (BTH)

Algorithms for games

29th June 2006 9 / 96

4 things that went bad

- The actual game is not very fun
- The graphics drawing is too slow on actual hardware
 - I've begun on an optimized version in assembly
- There is a limit of 16 objects that the player can hold
- Ugly graphics :-)

Simon Kågström (BTH)

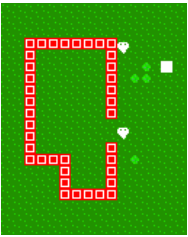
Algorithms for games

29th June 2006 10 / 96

- The book (Chapter 13) presents some basic building blocks: *roaming*, *evading* and *chasing* AI
- Simple behavior can be constructed by assigning weights to different behaviors, e.g.,
 - 50% chasing the player
 - 10% trying to evade the player
 - 30% moving in a pattern
 - 10% random roaming
- Read this chapter through and try the examples
- The rest of the lecture will focus on more general concepts

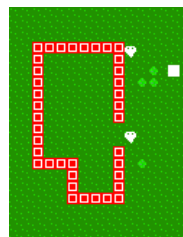
- We will look at *finite state machines* today
 - Pages 155-169 in the "Core techniques and algorithms in game programming" (Daniel Sanches-Crespo) book
- Finite state machines is a simple technique to model behavior graphically and semi-automatically transfer it to code
- The whole subject is best covered in a course about formal languages
 - At BTH we have "Automata and formal languages" (DVC005)
- This is just an introduction applied to games

The situation



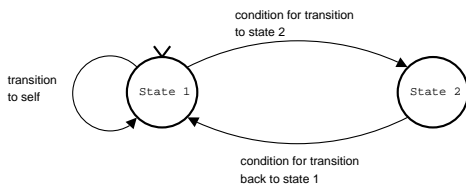
- Imagine a forest castle which contains a large treasure
- The castle, naturally, is heavily guarded
- Guards patrol the walls around the castle and will chase and capture any intruder
- The guards cannot see through walls or bushes

The situation, II



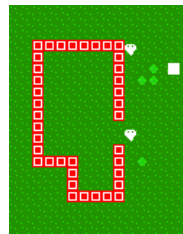
- The guards walk between *waypoints* around the castle
- The guards should behave intelligently
 - If the intruder is out of sight, they should not know where he/she is
 - Obstacle avoidance
- And this should be implemented on slow devices at interactive frame rates
- And of course with nice and clean code!

Finite state machines, graphical representation



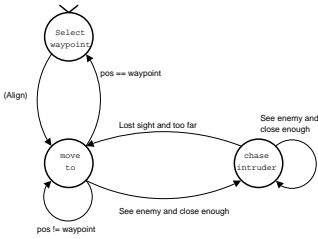
- State machines can be represented graphically
- Nodes are states, edges are transitions between states
- The arrow gives the destination state
- The initial state is given by the v-sign
- The machine can be in exactly *one* state at a time

Modeling the situation



- The guards can be in one of three states:
 - 1 Select which waypoint to go to
 - 2 Move towards a waypoint
 - 3 Chase an intruder

Modeling the situation, II



- The state machine for the guards is shown on the left
- How do the guards avoid obstacles? [Crash-and-turn pathfinding/A*]
- How do we move to a waypoint? [Another FSM]
- How do we know if the intruder is visible? [Bresenham's algorithm]

Simon Kågström (BTH)

Algorithms for games

29th June 2006

18 / 96

Converting the state machine to code

Description

- The FSM is implemented in a switch-statement
- Default actions specify what to do in the state
- Transition specifies the state to change to
- The FSM is called in each frame of the game loop

State machine

```
private void fsm() {
    switch(this.fsmState) {
    case STATE:
        [Default action(s)]
        if [Transition]
            this.fsmState = OTHER_STATE;
        break;
    case OTHER_STATE:
        ...
    }
```

Game loop

```
while (true) {
    /* Run the fsm in each frame */
    movingObject.fsm();
    ...
}
```

Simon Kågström (BTH)

Algorithms for games

29th June 2006

20 / 96

Implementation of the guard FSM

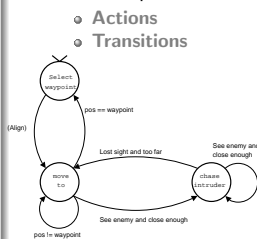
Example

```
switch(this.guardFsm.state) {
case GuardFsm.SELECT_WAYPOINT:
    guardFsm.nextWaypoint();
    guardFsm.state = GuardFsm.MOVE_TO;
    break;
case GuardFsm.MOVE_TO:
    moveTo(guardFsm.curWaypoint.x,
            guardFsm.curWaypoint.y);
    guardFsm.intruder = scanForIntruders();

    if (guardFsm.intruder != null &&
        dist(this, guardFsm.intruder) < 100 &&
        inLineOfSight(guardFsm.intruder))
        guardFsm.state = GuardFsm.CHASE_INTRUDER;
    else if (x == guardFsm.curWaypoint.x &&
             y == guardFsm.curWaypoint.y)
        guardFsm.state = GuardFsm.SELECT_WAYPOINT;
    break;
case GuardFsm.CHASE_INTRUDER:
    moveTo(guardFsm.intruder.x,
            guardFsm.intruder.y);

    if (dist(this, guardFsm.intruder) > 100 ||
        !inLineOfSight(this, guardFsm.intruder))
        guardFsm.state = GuardFsm.MOVE_TO;
    break;
}
```

- The code shows the entire implementation of the guard FSM
- Note the correspondence to the graph
- Code is separated into
 - Actions
 - Transitions



Simon Kågström (BTH)

Algorithms for games

29th June 2006

22 / 96

Implementation of the guard FSM

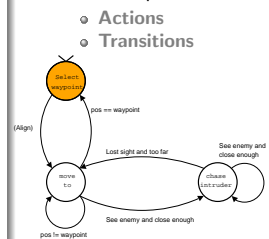
Example

```
switch(this.guardFsm.state) {
case GuardFsm.SELECT_WAYPOINT:
    guardFsm.nextWaypoint();
    guardFsm.state = GuardFsm.MOVE_TO;
    break;
case GuardFsm.MOVE_TO:
    moveTo(guardFsm.curWaypoint.x,
            guardFsm.curWaypoint.y);
    guardFsm.intruder = scanForIntruders();

    if (guardFsm.intruder != null &&
        dist(this, guardFsm.intruder) < 100 &&
        inLineOfSight(guardFsm.intruder))
        guardFsm.state = GuardFsm.CHASE_INTRUDER;
    else if (x == guardFsm.curWaypoint.x &&
             y == guardFsm.curWaypoint.y)
        guardFsm.state = GuardFsm.SELECT_WAYPOINT;
    break;
case GuardFsm.CHASE_INTRUDER:
    moveTo(guardFsm.intruder.x,
            guardFsm.intruder.y);

    if (dist(this, guardFsm.intruder) > 100 ||
        !inLineOfSight(this, guardFsm.intruder))
        guardFsm.state = GuardFsm.MOVE_TO;
    break;
}
```

- The code shows the entire implementation of the guard FSM
- Note the correspondence to the graph
- Code is separated into
 - Actions
 - Transitions



Simon Kågström (BTH)

Algorithms for games

29th June 2006

22 / 96

Implementation of the guard FSM

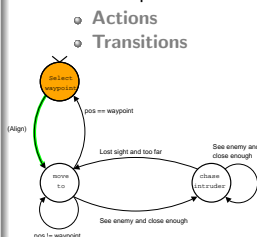
Example

```
switch(this.guardFsm.state) {
case GuardFsm.SELECT_WAYPOINT:
    guardFsm.nextWaypoint();
    guardFsm.state = GuardFsm.MOVE_TO;
    break;
case GuardFsm.MOVE_TO:
    moveTo(guardFsm.curWaypoint.x,
            guardFsm.curWaypoint.y);
    guardFsm.intruder = scanForIntruders();

    if (guardFsm.intruder != null &&
        dist(this, guardFsm.intruder) < 100 &&
        inLineOfSight(guardFsm.intruder))
        guardFsm.state = GuardFsm.CHASE_INTRUDER;
    else if (x == guardFsm.curWaypoint.x &&
             y == guardFsm.curWaypoint.y)
        guardFsm.state = GuardFsm.SELECT_WAYPOINT;
    break;
case GuardFsm.CHASE_INTRUDER:
    moveTo(guardFsm.intruder.x,
            guardFsm.intruder.y);

    if (dist(this, guardFsm.intruder) > 100 ||
        !inLineOfSight(this, guardFsm.intruder))
        guardFsm.state = GuardFsm.MOVE_TO;
    break;
}
```

- The code shows the entire implementation of the guard FSM
- Note the correspondence to the graph
- Code is separated into
 - Actions
 - Transitions



Simon Kågström (BTH)

Algorithms for games

29th June 2006

22 / 96

Implementation of the guard FSM

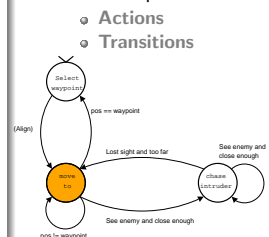
Example

```
switch(this.guardFsm.state) {
case GuardFsm.SELECT_WAYPOINT:
    guardFsm.nextWaypoint();
    guardFsm.state = GuardFsm.MOVE_TO;
    break;
case GuardFsm.MOVE_TO:
    moveTo(guardFsm.curWaypoint.x,
            guardFsm.curWaypoint.y);
    guardFsm.intruder = scanForIntruders();

    if (guardFsm.intruder != null &&
        dist(this, guardFsm.intruder) < 100 &&
        inLineOfSight(guardFsm.intruder))
        guardFsm.state = GuardFsm.CHASE_INTRUDER;
    else if (x == guardFsm.curWaypoint.x &&
             y == guardFsm.curWaypoint.y)
        guardFsm.state = GuardFsm.SELECT_WAYPOINT;
    break;
case GuardFsm.CHASE_INTRUDER:
    moveTo(guardFsm.intruder.x,
            guardFsm.intruder.y);

    if (dist(this, guardFsm.intruder) > 100 ||
        !inLineOfSight(this, guardFsm.intruder))
        guardFsm.state = GuardFsm.MOVE_TO;
    break;
}
```

- The code shows the entire implementation of the guard FSM
- Note the correspondence to the graph
- Code is separated into
 - Actions
 - Transitions



Simon Kågström (BTH)

Algorithms for games

29th June 2006

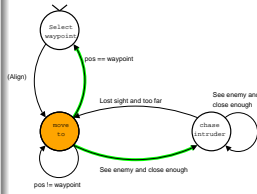
22 / 96

Implementation of the guard FSM

Example

```
switch(this_guardFsm.state) {
case GuardFsm.SELECT_WAYPOINT:
guardFsm.nextWaypoint();
guardFsm.state = GuardFsm.MOVE_TO;
break;
case GuardFsm.MOVE_TO:
moveTo(guardFsm.curWaypoint.x,
guardFsm.curWaypoint.y);
guardFsm.intruder = scanForIntruders();
if (guardFsm.intruder != null &&
dist(this, guardFsm.intruder) < 100 &&
inLineOfSight(guardFsm.intruder))
guardFsm.state = GuardFsm.CHASE_INTRUDER;
else if (x == guardFsm.curWaypoint.x &&
y == guardFsm.curWaypoint.y)
guardFsm.state = GuardFsm.SELECT_WAYPOINT;
break;
case GuardFsm.CHASE_INTRUDER:
moveTo(guardFsm.intruder.x,
guardFsm.intruder.y);
if (dist(this, guardFsm.intruder) > 100 ||
!inLineOfSight(this_guardFsm.intruder))
guardFsm.state = GuardFsm.MOVE_TO;
break;
}
```

- The code shows the entire implementation of the guard FSM
- Note the correspondence to the graph
- Code is separated into
 - Actions
 - Transitions

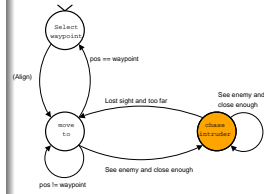


Implementation of the guard FSM

Example

```
switch(this_guardFsm.state) {
case GuardFsm.SELECT_WAYPOINT:
guardFsm.nextWaypoint();
guardFsm.state = GuardFsm.MOVE_TO;
break;
case GuardFsm.MOVE_TO:
moveTo(guardFsm.curWaypoint.x,
guardFsm.curWaypoint.y);
guardFsm.intruder = scanForIntruders();
if (guardFsm.intruder != null &&
dist(this, guardFsm.intruder) < 100 &&
inLineOfSight(guardFsm.intruder))
guardFsm.state = GuardFsm.CHASE_INTRUDER;
else if (x == guardFsm.curWaypoint.x &&
y == guardFsm.curWaypoint.y)
guardFsm.state = GuardFsm.SELECT_WAYPOINT;
break;
case GuardFsm.CHASE_INTRUDER:
moveTo(guardFsm.intruder.x,
guardFsm.intruder.y);
if (dist(this, guardFsm.intruder) > 100 ||
!inLineOfSight(this_guardFsm.intruder))
guardFsm.state = GuardFsm.MOVE_TO;
break;
}
```

- The code shows the entire implementation of the guard FSM
- Note the correspondence to the graph
- Code is separated into
 - Actions
 - Transitions

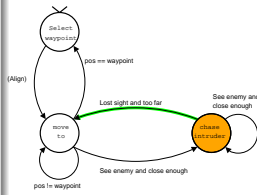


Implementation of the guard FSM

Example

```
switch(this_guardFsm.state) {
case GuardFsm.SELECT_WAYPOINT:
guardFsm.nextWaypoint();
guardFsm.state = GuardFsm.MOVE_TO;
break;
case GuardFsm.MOVE_TO:
moveTo(guardFsm.curWaypoint.x,
guardFsm.curWaypoint.y);
guardFsm.intruder = scanForIntruders();
if (guardFsm.intruder != null &&
dist(this, guardFsm.intruder) < 100 &&
inLineOfSight(guardFsm.intruder))
guardFsm.state = GuardFsm.CHASE_INTRUDER;
else if (x == guardFsm.curWaypoint.x &&
y == guardFsm.curWaypoint.y)
guardFsm.state = GuardFsm.SELECT_WAYPOINT;
break;
case GuardFsm.CHASE_INTRUDER:
moveTo(guardFsm.intruder.x,
guardFsm.intruder.y);
if (dist(this, guardFsm.intruder) > 100 ||
!inLineOfSight(this_guardFsm.intruder))
guardFsm.state = GuardFsm.MOVE_TO;
break;
}
```

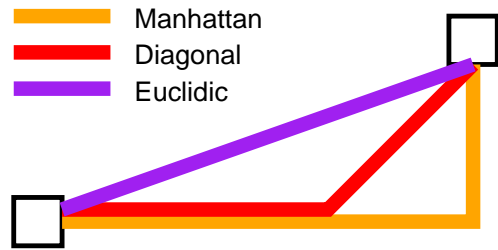
- The code shows the entire implementation of the guard FSM
- Note the correspondence to the graph
- Code is separated into
 - Actions
 - Transitions



Implementing move to

Description

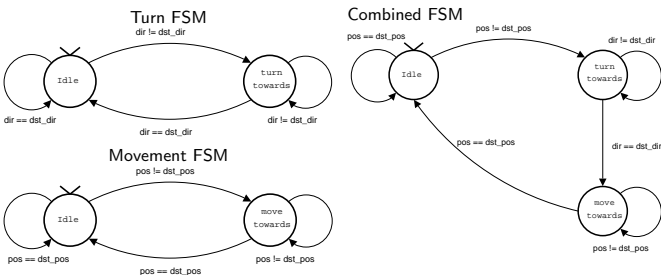
- Manhattan distance (example below)
- Euclidian distance guarantees shortest paths
- Diagonal movement (watch out for speed!)



Implementing move to, II

Description

- Movement can also be implemented in state machines
- Direction and actual movement can be split in two parallel FSMs
- The movement FSM only sets the speed, the turn FSM sets the direction



Implementing inLineOfSight

Description

- An easy way of determining if an object is visible is using *Bresenham's algorithm*
- The algorithm is used to draw solid lines



Example

```
class Guard implements BresenhamCallback {
...
public boolean bresenhamStep(int x, int y) {
if (this.playfield.isObstacle(x,y))
return true;
return false;
}
public boolean inLineOfSight(int x, y) {
return !Bresenham.run(this.getX(), this.getY(), x, y);
}
}
```

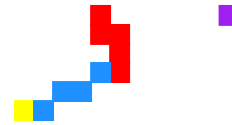
Implementing inLineOfSight, II

- The animation shows how `inLineOfSight` is implemented using Bresenham's algorithm



Implementing inLineOfSight, II

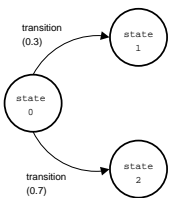
- The animation shows how `inLineOfSight` is implemented using Bresenham's algorithm



Example

```
class Guard implements BresenhamCallback {
    ...
    public boolean bresenhamStep(int x, int y) {
        if (this.playfield.isObstacle(x,y))
            return true;
        return false;
    }
    public boolean inLineOfSight(int x, y) {
        return !Bresenham.run(this.x, this.y, x, y);
    }
}
```

More advanced concepts

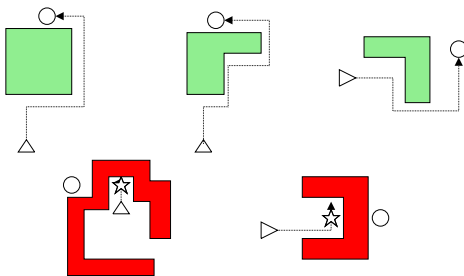


- Non-determinism
 - Multiple transitions with weights
 - 30% chance $state_0 \rightarrow state_1$
 - 70% chance $state_0 \rightarrow state_2$
- Parallel state machines
 - With too many states and transitions, state machines become very complex
 - Better then to separate behavior into multiple state machines

Crash-and-turn path finding

- If we have only convex obstacles, path finding becomes simple
- We can then use *crash-and-turn*-based path finding:
 - Start moving in the direction towards the target
 - If something blocks your way, walk along that (either way)
 - If it no longer blocks the way then **goto 1**
- Features:
 - Runs fast, scales well
 - Looks OK (i.e. the non-player characters looks like they try to find a way)
 - Uses constant memory
 - With concave obstacles the NPCs can get stuck

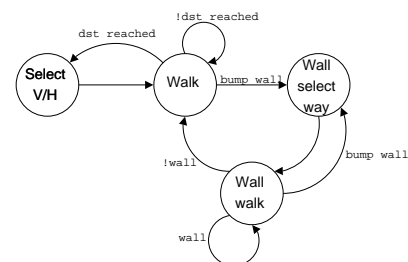
Crash-and-turn II



- How does it behave?
 - We look at the special case with Manhattan movement
- No need to be ashamed for this stupid algorithm: In command and conquer, it was possible to trap units inside U-shaped areas

Crash-and-turn implementation

- We can define a simple state machine for crash-and-turn
- The algorithm selects a direction to minimize
- If the NPC bumps against a wall, it starts walking along the wall



C-T implementation, II

```
void crashAndTurnFsm() {
  switch(this.ctFsm.state) {
  case SELECT_V_H:
    /* Select vertical or horizontal */
    this.ctFsm.selectWay = !this.ctFsm.selectWay;
    /* Set dir depending on selectWay (up/down, left/right) */
    break;
  case WALK:
    /* Walk towards the selected direction */
    if (this.move(this.ctFsm.dir) == false)
      this.ctFsm.state = WALL_SELECT_WAY; /* Bump! */
    else if (this.x == this.dstX && this.y == this.dstY)
      this.ctFsm.state = SELECT_V_H; /* Walk OK - have we reached the dst? */
    break;
  case WALL_SELECT_WAY:
    /* Select the direction along the wall */
    if (this.ctFsm.dir == UP || this.ctFsm.dir == DOWN) {
      this.ctFsm.h_way_right = !this.ctFsm.h_way_right;
      this.ctFsm.wall_dir = this.ctFsm.h_way_right ? RIGHT : LEFT;
    }
    else if (p_ai->ct_data.dir == LEFT || p_ai->ct_data.dir == RIGHT)
      ...
    this.ctFsm.state = WALL_WALK;
    break;
  case WALL_WALK:
    /* Walk along the wall */
    if (!this.is_wall(this.ctFsm.dir))
      this.ctFsm.state = WALK;
    else if (this.move(this.ctFsm.wall_dir) == false)
      this.ctFsm.state = WALL_SELECT_WAY; /* Bump! */
    break;
  }
}
```

Simon Kågström (BTH)

Algorithms for games

29th June 2006

36 / 96

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

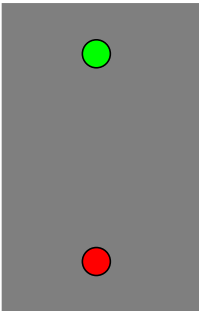
Simon Kågström (BTH)

Algorithms for games

29th June 2006

38 / 96

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)?
- 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 would do

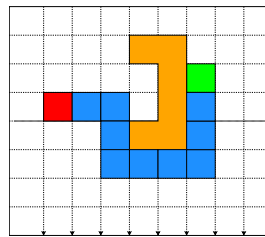
Simon Kågström (BTH)

Algorithms for games

29th June 2006

39 / 96

A*, terminology



- A* finds a shortest path between red 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 than on dirt roads

Simon Kågström (BTH)

Algorithms for games

29th June 2006

40 / 96

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

$$f(\text{node}) = g(\text{node}) + h(\text{node})$$

- Where $g(\text{node})$ is the accumulated cost to get to the node and
- $h(\text{node})$ 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

Simon Kågström (BTH)

Algorithms for games

29th June 2006

41 / 96

Heuristics

- There are different ways of approximating $h(\text{node})$:

- 1 Manhattan distance, $d_{\text{manhattan}} = D(\Delta x + \Delta y)$

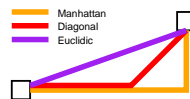
- D is the minimal cost to move between two nodes
- Manhattan distance is optimistic, so we can guarantee shortest paths

- 2 Diagonal distance, $d_{\text{diagonal}} = D(\max(\Delta x, \Delta y))$

- 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, $d_{\text{euclid}} = D\sqrt{\Delta x^2 + \Delta y^2}$

- If you allow movement along arbitrary angles



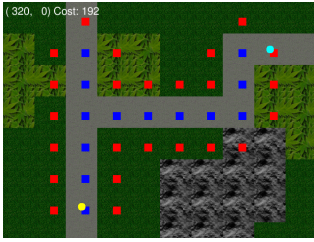
Simon Kågström (BTH)

Algorithms for games

29th June 2006

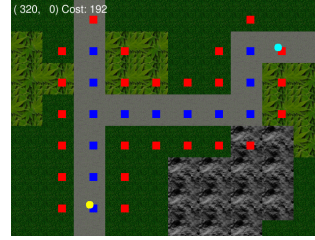
42 / 96

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



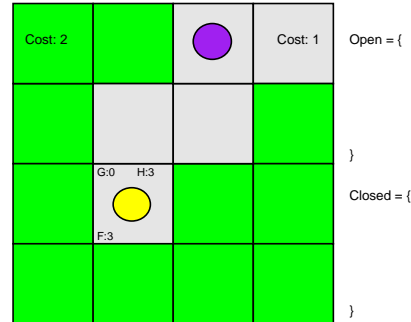
- Basic algorithm (repeat until goal is found):
 - 1 Get the node with lowest *f* from *Open*, put in *Closed*
 - 2 Expand the neighbors of that node, calculate *f* and insert into *Open*
- The red nodes are in the *Open* set, the blue in the *Closed*

Implementation sketch

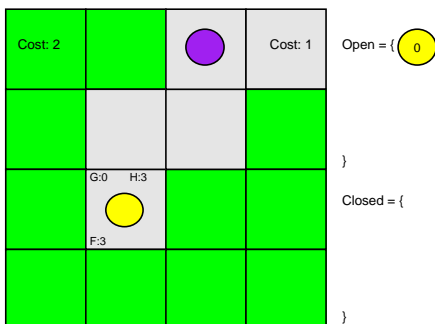
```

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}
    
```

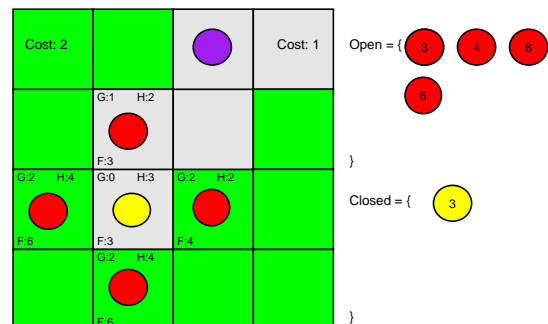
A*, execution



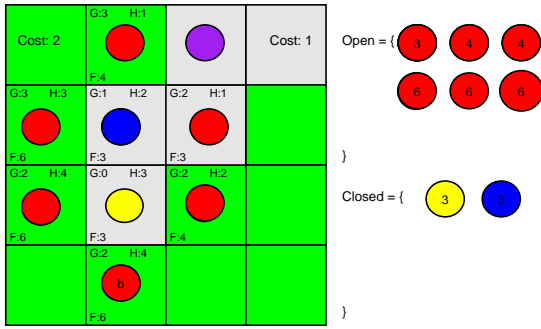
A*, execution



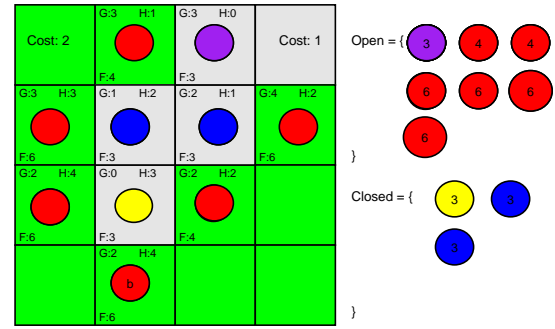
A*, execution



A*, execution



A*, execution



A*, some implementation notes

Some tips for the implementation:

- ① You can store the g -value $g[node]$ for a node in the node, i.e., `node.g`
 - (Same with f)
 - The Finder class uses this (see the instructions for lab 3)
- ② HEURISTIC can be implemented as Manhattan distance between the nodes (i.e., distance between the tiles)
- ③ The predecessor π is a reference to the previous node
- ④ The *Open* set should provide fast extraction of the lowest- f node and quick insertion
 - The Vector class is not the best, but will do initially

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
- ...

Questions?

Post Mortem - Boulder Dash



- Another old favorite of mine is **Boulder Dash**
- 1994-1995 I made a version of it for MS-DOS
 - Worked fine, but with ugly code
- Last time this course was held, I made an updated version for **Mophon**
- http://spel.bth.se/index.php/Ska:Boulder_Dash

Simon Kågström (BTH)

Algorithms for games

29th June 2006

51 / 96

Simon Kågström (BTH)

Algorithms for games

29th June 2006

55 / 96

Boulder Dash - design

- I wanted to keep the gameplay from my first version (shown below)
 - More puzzle-style and less of the originals arcade-style gameplay
 - Walls that can be passed one-way only
 - Bombs that can be placed
 - Blocks that can be pushed in all directions, etc., etc.



Simon Kågström (BTH)

Algorithms for games

29th June 2006

56 / 96

How it turned out



- The game stays fairly close to my original implementation
- The new version is scrolling
- A few features (key types, the lamp) not present in the original game were added
- Much cleaner implementation
- Three levels were constructed, easy to add more

Simon Kågström (BTH)

Algorithms for games

29th June 2006

57 / 96

5 things that went good

Description

- Performance is fine on a slow phone (more a few slides ahead)
- Many features
- Extensible design, easy to add levels, nice graphics!
- The implementation of lamp visibility became short and efficient (below)

Code

```
/* (to the right)      (up)
 *   .   3             4 5 6
 * . 7 8 2             2 3 .
 * . 4 5 6 1           1 . .
 * X 1 2 3 0           X . .
 *   1 2 3
 *
 * 1,2,3               1,2,4
 * 4,5,6               1,3,5
 * 4,5,7,8             1,3,6
 */
```

Simon Kågström (BTH)

Algorithms for games

29th June 2006

59 / 96

5 things that went bad



- The player moves in steps of 8 (the tile size)
- The sprites are only 8x8 pixels large, which is too small to show details
 - I started looking at these two problems too late and it became difficult to fix these
- The levels supplied are moderately fun to play. I could use a level designer.
- The implementation is fairly tied to Mophon (more than needed), which makes porting more difficult

Simon Kågström (BTH)

Algorithms for games

29th June 2006

60 / 96

- Optimization: *A triangle was an improvement to the square wheel. It eliminated one bump* (BC comics)
- Java is faster than C++! (<http://www.kano.net/javabench/>)
- No, C++ is faster than Java! (http://www.freewebs.com/godaves/javabench_revisited/)
- Who cares? Writing in assembly is faster anyway! (<http://www.azillionmonkeys.com/qed/optimize.html>)
 - Please take the UNIX programming course (DVC011) to find out more
- Here, you don't have a choice

- (From Sun's CLDC HotSpot whitepaper, updated)

CPU type	ARM
CPU freq.	30-400MHz
RAM	MB-range
ROM / flash	8 MB - GB-range
RAM for Java stack	Around 1 MB
- The hardware and JVMs differ between different phones
 - E.g., FPU support can make a large difference!
 - Might get substantially different results when running on actual hardware
- http://www.club-java.com/TastePhone/J2ME/MIDP_mobile.jsp

- Different types of optimization
 - 1 Maintainability
 - 2 Portability
 - 3 Size
 - 4 Speed
- General tips:
 - Reduce memory usage
 - Avoid using objects
 - Recycle objects when using them
 - Perform cleanup manually instead of relying on the GC
 - Minimize network data
 - Remove unused graphics

- "Maintainability is the least important optimization aspect"
 - Don't believe him!
 - Write simple and well-thought out code, it will save you time
- "Avoid objects whenever possible"
 - Don't believe him!
 - Structure your code well instead, use objects if that is appropriate, primitive types if that is appropriate
- Java optimization tricks
 - Loop expansion, common subexpressions, unnecessary evaluations
 - You can safely skip these
 - Sometimes taken care of by the compiler (more so in the future), sometimes make no difference anyway

Description

- In a single-threaded game, the game loop is always the base of the performance issues
- Here, either the update() or the draw() functions
- *But don't rely on code inspection*

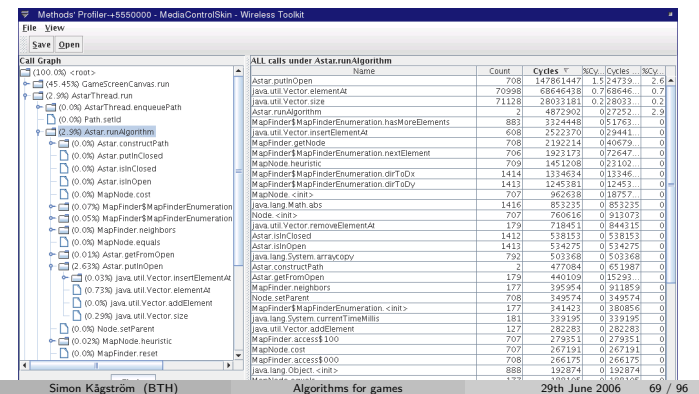
Example

```
public void run() {
    Graphics g = getGraphics();

    while (true) {
        long before = System.currentTimeMillis();

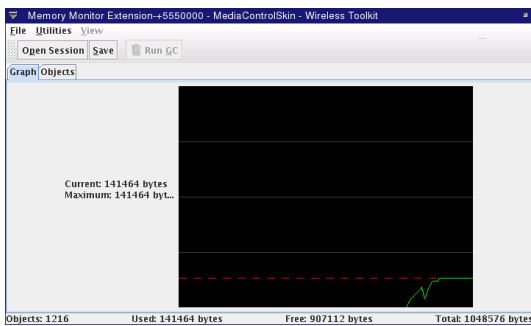
        this.update();
        draw(g);
        try {
            long sleepTime = 33 - (System.currentTimeMillis() - before);
            Thread.sleep(sleepTime);
        } catch (InterruptedException e) {}
    }
}
```

- A profiler shows where the hotspots in the program are
- Use the profiler *before* starting to optimize



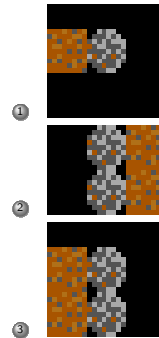
Memory consumption

- Object size is determined by the data in the attributes
- Reducing precision (e.g., double to float) reduces memory consumption
- Static constants



Simon Kågström (BTH) Algorithms for games 29th June 2006 70 / 96

Example, Boulder Dash



- We will illustrate the importance of algorithms and data structures from a **Boulder Dash** implementation
- (See <http://www.bd-fans.com/>)
- We will look at boulder/diamond falling. A boulder fall when:
 - There is an empty space under it,
 - There is a boulder *under it* and the space *west and south west* is empty
 - There is a boulder *under it* and the space *east and south east* is empty

Simon Kågström (BTH) Algorithms for games 29th June 2006 72 / 96

Boulder dash, first try

Description

- The easy(?) way of implementing boulder falling
- The tilemap is used to store boulders directly
- However: the complexity is $O(\text{size}_{\text{level}})$
- I.e. even with few boulders this will be slow - we can do better!
- Why am I looping backwards in the y-direction?

Example

```
for (y = LEVEL_H; y >= 0; y--) {
  for (x = 0; x < LEVEL_W; x++) {
    if (map.getTileType(x,y) == BOULDER &&
        (map.getTileType(x,y+1) == EMPTY || /* case 1 */
         map.getTileType(x,y+1) == BOULDER && /* case 2, 3 */)) {
      Boulder boulder = map.getTile(x,y);
      boulder.fall();
    }
  }
}
```

Simon Kågström (BTH) Algorithms for games 29th June 2006 74 / 96

Second try, boulders in a vector

Description

- Keeping the boulders in a vector lets us traverse the vector for checking case 1
- However, cases 2 and 3 require a traversal over the boulders again, making the complexity $O(\text{size}_{\text{vec}}^2)$
- So this will be slow when we increase the number of boulders

Example

```
/* Case 1 */
for (i = 0; i < n_boulders; i++) {
  if (map.getTileType(boulder[i].x, boulder[i].y + 1) == EMPTY) /* case 1 */
    boulder[i].fall();
  for (j = 0; j < n_boulders; j++) {
    /* case 2,3 */
  }
}
```

Simon Kågström (BTH) Algorithms for games 29th June 2006 76 / 96

Third try, combination of the two

Description

- Using a boulder vector *and* a matrix for the playfield, we can get $O(\text{size}_{\text{vec}})$ on average!
- The matrix consists of references to boulders
- Memory can be reduced by holding indices to the boulder vector
- Conclusion: with clever use of data structures, the performance can be noticeably increased

Example

```
for (i = 0; i < n_boulders; i++) {
  /* Case 1 */
  if (map.getTileType(boulder[i].x, boulder[i].y+1) == EMPTY ||
      (map.getTileType(boulder[i].x, boulder[i].y+1) == BOULDER &&
       /* Case 2 and 3 */))
    boulder[i].fall();
}
```

Simon Kågström (BTH) Algorithms for games 29th June 2006 78 / 96

Additional thoughts



- "If a tree falls in the forest and there is no one there to see it, does it make a sound?"
- We could update only the boulders that are close enough to the player
- Maybe the boulders could be updated every second iteration of the game loop (slower falling perhaps)
- (Applicable also in other cases)

Simon Kågström (BTH) Algorithms for games 29th June 2006 79 / 96

Java compiler/JVM

- Java differs from C/C++ in that the compiler only do a small part of the optimization - the JVM is responsible for most of the code optimizations

Examples

```
public int test_code_motion() {
    int out = 0;
    for (int i = 0; i < 10; i++)
    {
        int tjoho = 5 * a;
        out += tjoho;
    }
    return out;
}

Code:
0: iconst_0
1: istore_1
2: iconst_0
3: istore_2
4: iload_2
5: bipush 10
7: if_icmpge 27
10: iconst_5
11: aload_0
12: getfield #4; //Field a:I
15: imul
16: istore_3
17: iload_1
18: iload_3
19: iadd
20: istore_1
21: inc 2, 1
23: goto 4
27: iload_1
28: ireturn

<Test::test_code_motion():
1e0: push %ebp
1e1: mov %esp,%ebp
1e3: mov 0x8(%ebp),%eax
1e6: mov 0x4(%eax),%eax
1e9: lea (%eax,%eax,4),%eax
1ec: lea (%eax,%eax,8),%edx
1ef: add %edx,%eax
1f1: pop %ebp
1f2: ret
```

Simon Kågström (BTH)

Algorithms for games

29th June 2006 81 / 96

Java compiler/JVM, II

- How much the JVM can do varies depending on the implementation
- The best will be able to do optimizations like GCC did on the last slide
- On mobile phones, they are likely not that advanced
 - Constant propagation will be there
 - Copy propagation
 - (possible) common subexpression elimination
 - (possible) code motion

Don't spend your efforts optimizing things the compiler is better at!

Simon Kågström (BTH)

Algorithms for games

29th June 2006 83 / 96

Misc performance issues

- On some architectures floating point arithmetic can be very expensive
- switch statements come in two variants: as vector indexes if the values are adjacent or as lookup tables if the values are disparate
- Local variables are faster than member variables
- Object-orientation comes with a cost (table lookup for derived class members etc)
 - Calling private methods is slightly cheaper than public
 - Class-static methods are even cheaper
- Declare constants as final
- Threads and thread switching includes overhead, especially synchronized methods

But do not spend time on low-level optimization needlessly

Simon Kågström (BTH)

Algorithms for games

29th June 2006 84 / 96

Object Pooling

- One optimization (described in the book) is to maintain a pool of objects that are reused
 - Moving an object off-screen when it is no longer used and
 - Reinitializing the object again when it is time to move it back again
- This reduces memory and potentially the creation and deletion time
- *But:* it breaks the common initialize-in-constructor pattern
- **Conclusion:** Use if you have scarce memory or in performance-critical code (A* could be one example)

Simon Kågström (BTH)

Algorithms for games

29th June 2006 85 / 96

Threading strategies

Description

- In my A* implementation, I use multithreading to hide the latency
- The game runs in one thread and a *worker thread* computes paths in the background
- The game loop checks for finished paths and handles them when they are ready

Code

```
public class GameScreenCanvas {
    ...
    public void run() {
        long before = System.currentTimeMillis();
        this.update();
        this.draw();

        long sleepTime = System.currentTimeMillis() - before;
        Thread.sleep(sleepTime);
    }

    private void update() {
        Path p = this.astarThread.dequeuePath();
        if (p != null)
            this.handlePath(p); /* Or something */
        player.handleInput();
        ...
    }
}
```

Simon Kågström (BTH)

Algorithms for games

29th June 2006 87 / 96

Threading strategies, II

Description

- The A* thread just waits for items, handles them and then enqueues the result
- The thread blocks if no work is available

Code

```
public class AstarThread ... {
    public void run()
    {
        while (true) {
            /* Get a request (block if no work available) */
            Request r = this.dequeueRequest();
            Path p = astar.runAlgorithm(...);

            p.setId(r.id);
            this.enqueuePath(p);
        }
    }
}
```

Simon Kågström (BTH)

Algorithms for games

29th June 2006 89 / 96

