CE318: High-level Games Development

Lecture 8: Steering and Navigation

Diego Perez

dperez@essex.ac.uk
Office 3A.527

2016/17

Outline

Outline

Steering Behaviours

Steering behaviours aim to help autonomous characters move in a realistic manner, by using simple forces that are combined to produce life-like, improvisational navigation around the characters' environment.

They are not based on complex strategies involving path planning or global calculations, but instead use local information, such as neighbours' forces. This makes them simple to understand and implement, but still able to produce very complex movement patterns.

Bibliography:

- Craig W. Reynolds "Steering Behaviours for Autonomous Characters":

  `www.cs.uu.nl/docs/vakken/mcrs/papers/8.pdf`
  `http://www.red3d.com/cwr/steer/gdc99/`

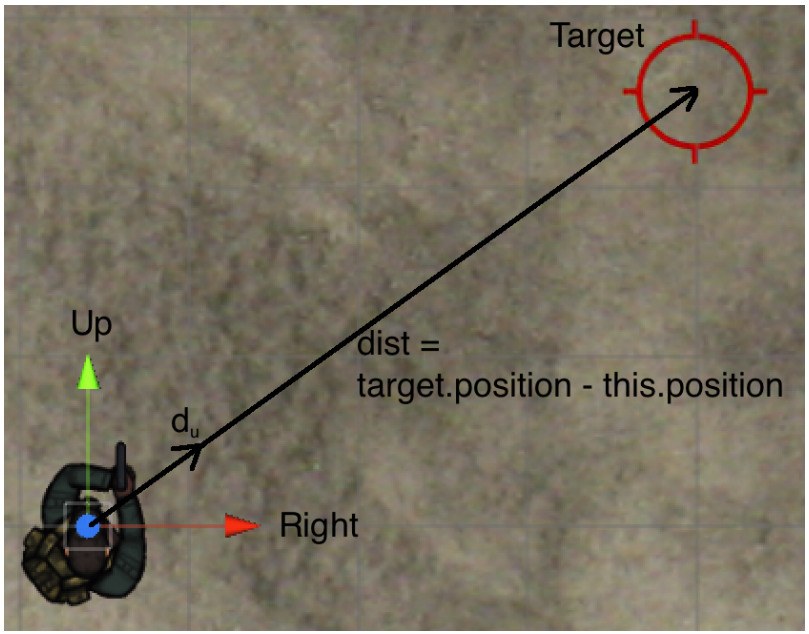
- Understanding Steering Behaviours:

  `http://gamedevelopment.tutsplus.com/series/understanding-steering-behaviors--gamedev-12732`

Orientation

Scenario: the character in the image wants to change its orientation to face the target.

- In 2D, Up $\vec{U} = (0.0, 1.0)$ and Right $\vec{R} = (1.0, 0.0)$ form the local space.
- `this.position` is the position of the character $\vec{c} = (c_x, c_y)$.
- `target.position` is the position of the target $\vec{t} = (t_x, t_y)$.



- `dist` is the vector from the character to the target $\vec{d} = (t_x - c_x, t_y - c_y)$.
  - $|\vec{d}|$ is the distance to the target.
  - $\vec{d_u} = (d_x, d_y) = (\frac{t_x - c_x}{|\vec{d}|}, \frac{t_y - c_y}{|\vec{d}|})$ is the unit vector towards the target.

- The new orientation (in radians) is the *arc tangent* of $(-d_x, d_y)$.
- Or, from the dot product, the *arc cos* of $(\vec{U} \cdot \vec{d_u})$.

Movement and Orientation

Movement can be obtained by adding a velocity $\vec{v} = (v_x, v_y, v_z)$ to the current position $\vec{c} = (c_x, c_y, c_z)$:

$$\vec{c_{t+1}} = (c_{t_x}, c_{t_y}, c_{t_z}) + (v_x, v_y, v_z)$$

Usually, this change is scaled by a factor of time. Assuming Δt is the time between two consecutive calls to an update function, the position can be defined as:

$$\vec{c_{t+1}} = (c_{t_x}, c_{t_y}, c_{t_z}) + ((v_x, v_y, v_z) \times \Delta t)$$

In code, we can update the position and orientation of an agent by doing:

```
1 void Update() {
2    Vector3 target = world.GetTarget (); //Target to move to/away, etc.
3    Vector3 velocity = behaviour.Move(target); //obtain desired vel.
4    position = position + velocity * Time.deltaTime; //update position.
5 }
```

Types of Steering Behaviours

Some Steering Behaviours:

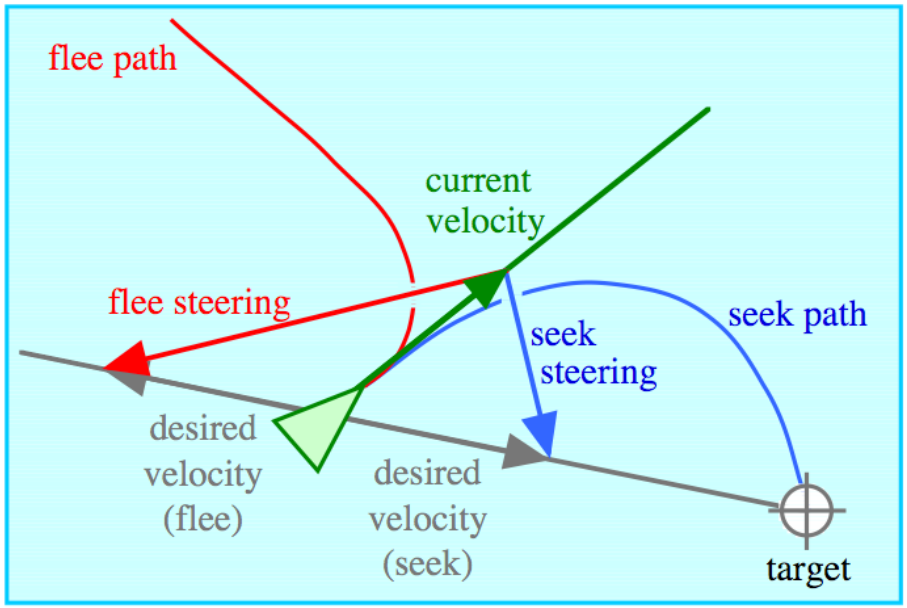
- **Seek**: steers the character towards a specified position in global space.
- **Flee**: the inverse of *Seek*.
- **Arrival**: as *Seek*, but decelerating when approaching the target.
- **Wander**: an "interesting" random steering.
- **Pursuit**: like *Seek* but the target is a moving object.
- **Evade**: analogous to *Pursuit*, but *Flee* is used to steer away from the predicted future position of the target.
- ...

Other concepts:

- Flocking.
- Obstacle/Wall Prediction and Avoidance.

Seek (1/2)

Seek (or pursuit of a static target) acts to steer the character towards a specified position in global space. This behavior adjusts the character so that its velocity is radially aligned towards the target. Note that this is different from an attractive force (such as gravity) which would produce an orbital path *around* the target point.



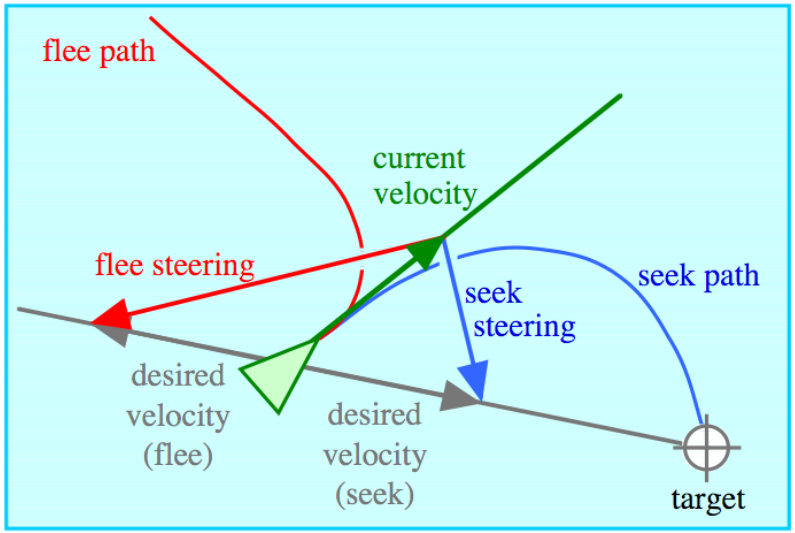
Seek (2/2)

```
1  void seek() {
2     //Desired velocity is the vector to the target.
3     desiredVelocity = (target.position - rigidbody.position);
4
5     //... normalized and with the correct speed
6     desiredVelocity.Normalize ();
7     desiredVelocity *= speed;
8
9     //The steering component to steer towards the target:
10    steering = desiredVelocity - currentVelocity;
11    desiredSpeed = speed;
12 }
```

In `FixedUpdate()`, we update the object's velocity accordingly:

```
1  void Start () {
2     rigidbody.velocity = initialVelocity;
3     currentVelocity = rigidbody.velocity;
4  }
5
6  void FixedUpdate () {
7
8     seek();
9
10    //Add steering component to the current velocity and desired speed
11    currentVelocity = currentVelocity + steering/rigidbody.mass;
12    currentVelocity.Normalize ();
13    currentVelocity *= desiredSpeed;
14
15    //Set the velocity to the rigidbody's velocity.
16    rigidbody.velocity = currentVelocity;
17 }
```

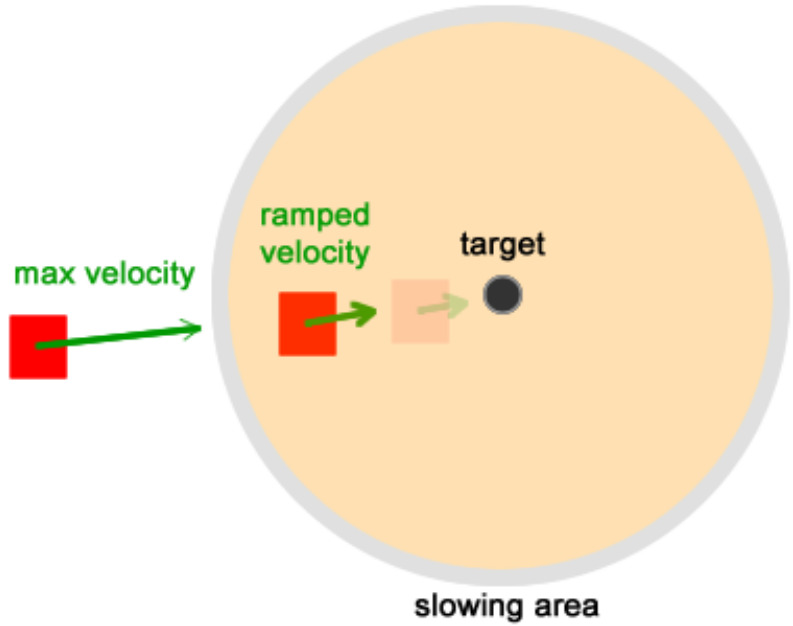
Flee




```
1  void flee()
2  {
3      //Desired velocity is the vector to the target.
4      desiredVelocity = (rigidbody.position - target.position);
5
6      //... normalized and with the correct speed
7      desiredVelocity.Normalize ();
8      desiredVelocity *= speed;
9
10     //The steering component to steer towards the target:
11     steering = desiredVelocity - currentVelocity;
12     desiredSpeed = speed;
13 }
```

Arrival (1/2)

The **Arrival** behaviour is identical to Seek, but instead of moving through the target at full speed, the character will slow down when reaching the target.



- Velocity reduced linearly to 0 when inside the slowing area.
- Achieved by adding a new steering component (*arrival*).
- Adding this component will (eventually) turn character's velocity to 0.

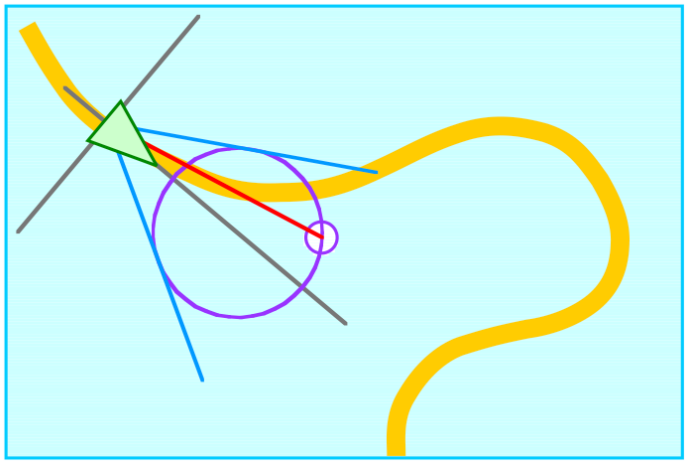
Arrival (2/2)

```
1  void arrival()
2  {
3     //Vector and distance to the target.
4     Vector3 targetOffset = (target.position - rigidbody.position);
5
6     //distance can be modified with an offset to stop
7     //  before reaching the target.
8     float distance = targetOffset.magnitude - arrivalStopDist;
9
10    //Speed will be normal speed if beyond slowing down distance.
11    // ...or less if inside the radius.
12    float rampedSpeed = speed * (distance / arrivalSlowingDist);
13    float reducedSpeed = Mathf.Min(speed, rampedSpeed);
14
15    //calculate desired velocity with the appropriate speed
16    desiredVelocity = targetOffset;
17    desiredVelocity.Normalize ();
18    desiredVelocity *= (reducedSpeed);
19
20    //and then the steering component of the velocity.
21    steering = desiredVelocity - currentVelocity;
22    desiredSpeed = reducedSpeed;
23 }
```

Wander (1/2)

The **Wander** behaviour is a type of random steering.

- Easy implementation (but uninteresting): a different random steering component each frame.
- Better: keep steering component, changing it randomly each frame.



- Two (Imaginary) circles (or spheres) located in front of the agent.
- Steering component is a vector to a point in the larger circle.
- This point is moved slightly and randomly every "N" frames.
- The circles radius determines the maximum wandering "strength".
- The magnitude of the random displacement (smaller circle) determines the wander "rate".

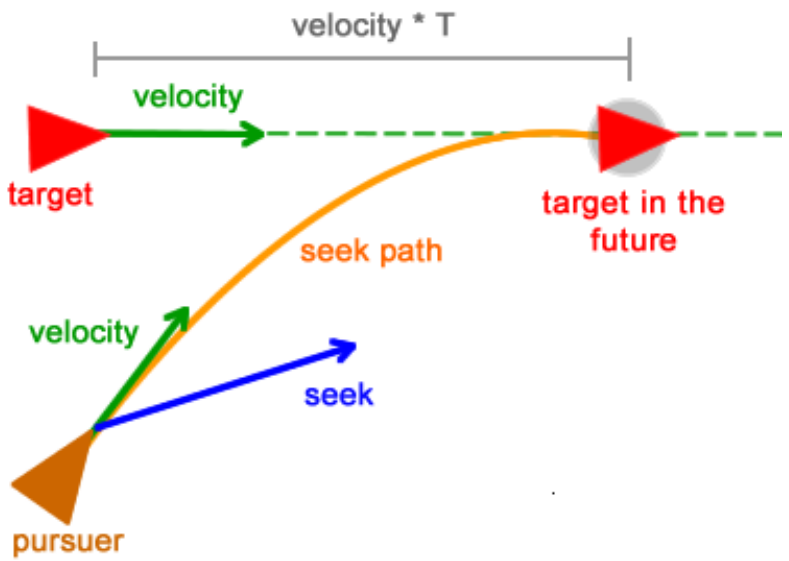
Wander (2/2)

```
1  void wander()
2  {
3     //Change the wander component every 'wanderTime' seconds
4     acumTime += Time.deltaTime;
5     if (acumTime > wanderTime) {
6        float r = Random.Range (-90.0f, 90.0f);
7        wanderSeeering = Quaternion.Euler (0, 0, r) * wanderSeeering;
8        acumTime = 0.0f;
9     }
10
11    //The new desired component is calculated by
12    // adding the wander component to the current velocity.
13    Vector3 wanderComponent = (wanderSeeering * circleRadius);
14    desiredVelocity = currentVelocity + wanderComponent;
15    desiredVelocity.Normalize ();
16    desiredVelocity *= speed;
17
18    //and then the steering component of the velocity.
19    steering = desiredVelocity - currentVelocity;
20    desiredSpeed = speed;
21 }
```

Pursuit (1/2)

The **Pursuit** behaviour is equivalent to the *Seek* behaviour, but the target is moving.

- Easy implementation (but inaccurate): leave seek as it is.
- Better: predict where the target is going to be.



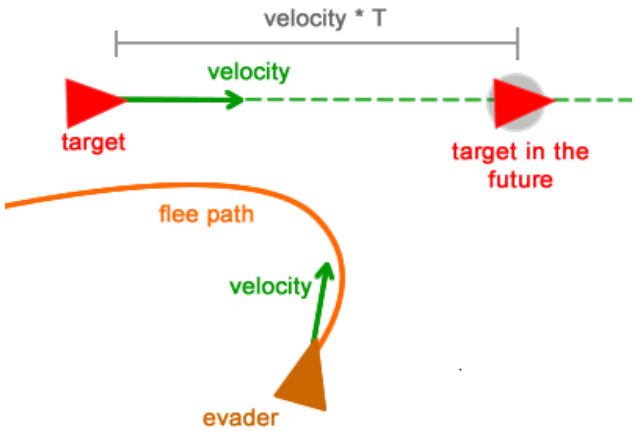
- Linear velocity-based predictor: assumes target will not turn.
- This is fine: evaluated at every frame (error is small).
- *Seek* to a predicted position T frames ahead.
- T should decrease as target is closer.

Pursuit (2/2)

```
1  void pursuit()
2  {
3      //For variable T depending on distance.
4      Vector3 toTarget = target.position - rigidbody.position;
5      float distance = toTarget.magnitude;
6      T = Mathf.Min(T, distance / speed);
7
8      Vector3 advance = target.GetComponent<Rigidbody2D>().velocity * T;
9      Vector3 predictedTarget = target.position + advance;
10
11     //calculate desired velocity using the prediction.
12     desiredVelocity = (predictedTarget - rigidbody.position);
13     desiredVelocity.Normalize ();
14     desiredVelocity *= speed;
15
16     //and then the steering component of the velocity.
17     steering = desiredVelocity - currentVelocity;
18     desiredSpeed = speed;
19 }
```

Evade

The **Evade** behaviour is equivalent to the *Flee* behaviour, but the target to flee from is moving (or to *Pursuit* but fleeing instead of seeking). No need to decrease T with distance!




```
1  void pursuit(){
2     //No need to decrease T with distance.
3     Vector3 advance = target.rigidbody.velocity * T;
4     Vector3 predictedTarget = target.position + advance;
5
6     //calculate desired velocity using the prediction.
7     desiredVelocity = (rigidbody.position - predictedTarget);
8     desiredVelocity.Normalize ();
9     desiredVelocity *= speed;
10
11    //and then the steering component of the velocity.
12    steering = desiredVelocity - currentVelocity;
13    desiredSpeed = speed;
14 }
```

Outline

Pathfinding

Amongst the most fundamental game Artificial Intelligence (AI) is path-finding, an ability central to most interactions of NPCs and gamers. We will consider simple Path-finding AI which, combined with some scripted behaviours, may lead to a sufficiently strong opponent.

First, we need to construct a graph that we can search:

- Nodes represent points in 2D or 3D space
- Nodes have neighbours (adjacent nodes)
- Neighbouring nodes may have different distances
- Nodes (i.e., their locations) can be constructed manually or automatically

The set of nodes that make up the graph are known as the pathfinding network.

We need a way to construct the graph.

- Discretise the continuous space using grids
- Simplest: regular grids (rectangular or hexagonal)
- Rectangular grid: 4- or 8-way connectivity

Pathfinding

In rectangular grids with 8-way connectivity, not all tile centers are equidistant. If we assume unit squares, then

- left, right, up, down: $1/2+1/2 = 1$
- up-left, up-right, down-left, down-right: $\sqrt{2}/2 + \sqrt{2}/2 = \sqrt{2}$

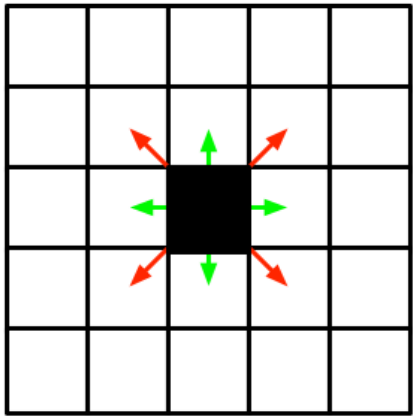
We also get "stair-case" appearance which may be improved by using higher resolutions (speed vs accuracy / visual appeal / gameplay experience).

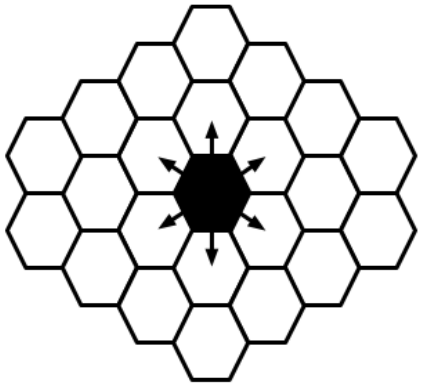
Hexagonal grids are bit more difficult to implement but may lead to more natural movement. Such regular grids can be generated automatically, speeding up development times. However, they are also costly as all "uninteresting" areas are covered equally.

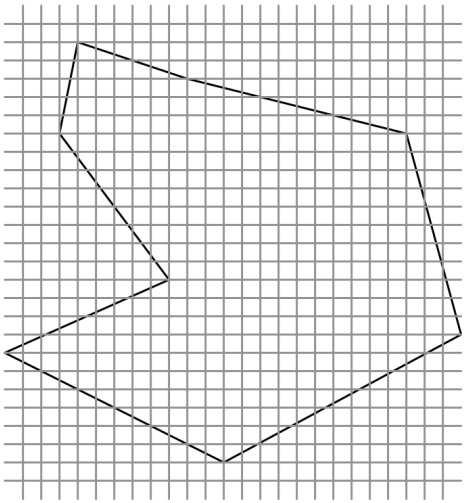
We can use arbitrary grids instead:

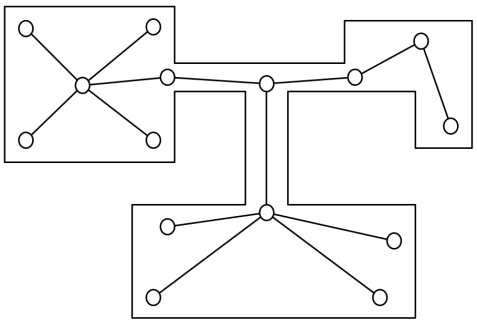
- Place nodes in specific locations (e.g., centre of door / hallway)
- Only need to stores points of interest (less data)
- Can incorporate tactics (places to hide etc.)

Pathfinding









Pathfinding

Once a grid is in place, we need to compute how to get from A to B using a path planning (or path finding) algorithm.

- Need origin (point A) and target (point B)
- We are usually interested in the shortest / quickest legal path

If level is static (and pathfinding network is small), can use a lookup table of precomputed shortest distances:

From A to E: lookup A-E, find node C, lookup C-E, find node K ...

Table can be generated using **Dijkstra's algorithm**.

Q Are distances always symmetric?

Assume 256 / 2048 nodes, how much memory is required?

- 256, need 8 bits and 65536 (2^{16}) entries, 65538 bytes = 64 kb
- 2048, need 11 bits, 5.5MB (more since we have no 11bit data structure)

Look-up tables are very fast but large, require level to be static and takes time to compute (or load). Can use a combination of table and search algorithm.

Dijkstra

Dijkstra's algoritm is a classical pathfinding algorithm that is very efficient (polynomial runtime). It finds the shortest path from a node to any other node in the network. It is a variation of breadth first search.

Dijkstra's algorithms makes use of a **priority queue**: a list that holds items with an associated priority. The list if always ordered such that the highest priority items are on top. The algorithm's efficiency depends crucially on the implementation of the priority queue used.

Unfortunately, there is none built-in priority queue in C#. It is simple to create your own (naive) implementation and not too difficult to create an efficient priority queue by yourself. We assume we have a class for this purpose, `PQ`.

Dijkstra

Both Dijkstra (and A*) search a graph $G = (N, E)$. We thus need to represent our world as a weighted unidirected graph. For this purpose we create the following classes:

- E: an edge from one node to another, with a given cost.
- N: a node with attributes required by the pathfinding algorithms.

The edge is defined as follows:

```
1  public class E
2  {
3    public N node; //destination of this edge
4    public double cost;
5
6    public E(N node, double cost)
7    {
8      this.node = node;
9      this.cost = cost;
10   }
11 }
```

Dijkstra

We define the node with respect to the pathfinding algorithms introduced next.

```
1  public class N
2  {
3    public N parent;
4    public double g, h;
5    public bool visited = false;
6    public List<E> adj;
7    public int x, y;
8
9    public N(int x, int y)
10   {
11     adj = new List<E>();
12     this.x = x;
13     this.y = y;
14   }
15
16   public bool isEqual(N another)
17   {
18     return x == another.x && y == another.y;
19   }
20
21   public bool IsBetter(N another)
22   {
23     if ((g + h) < (another.g + another.h))
24       return true;
25     else
26       return false;
27   }
28 }
```

Dijkstra

```
1  public static List<N> Dijsktra(N start, N target)
2  {
3    PQ pq = new PQ();
4    start.visited = true;
5    start.g = 0;  //we only use g in Dijkstra
6    pq.Add(start);
7
8    while (!pq.IsEmpty()) {
9      N currentNode = pq.Get(); //removes the node from PQ.
10
11     if (currentNode.isEqual(target))
12       break;
13
14     foreach (E next in currentNode.adj) {
15       double currentDistance = next.cost;
16
17       if (!next.node.visited) {
18         next.node.visited = true;
19         next.node.g = currentDistance + currentNode.g;
20         next.node.parent = currentNode;
21         pq.Add(next.node);
22       }
23       else if (currentDistance + currentNode.g < next.node.g) {
24           next.node.g = currentDistance + currentNode.g;
25           next.node.parent = currentNode;
26         }
27       }
28     }
29   }
30   return ExtractPath(target);
31 }
```

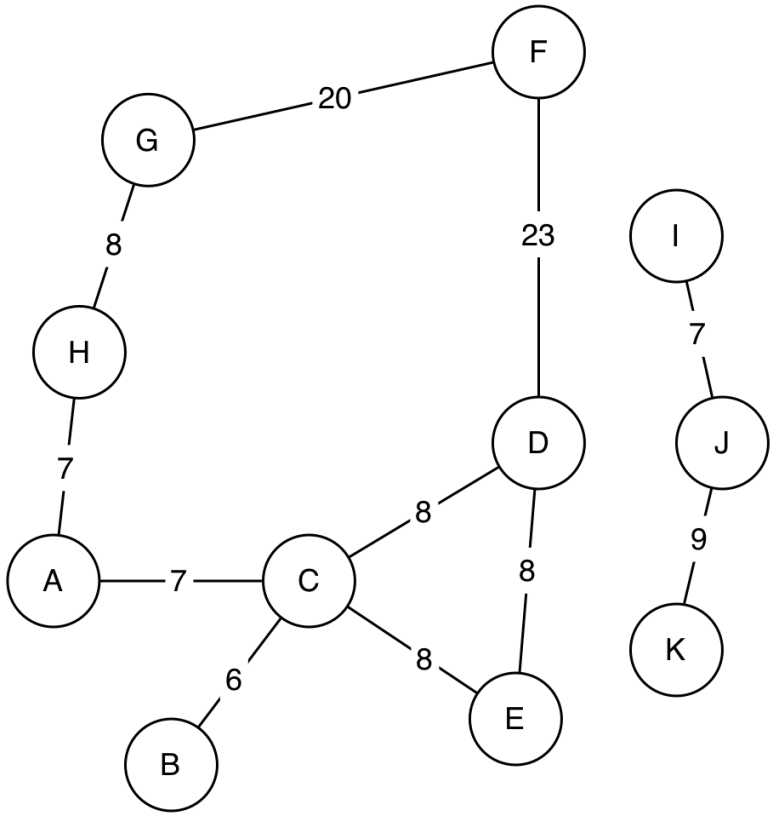
Dijkstra

The route can be extracted easily by following the chain of parents from the target to the start. We then reverse the route to return the path from start to finish.

```
1  private static List<N> ExtractPath(N target)
2  {
3    List<N> route = new List<N>();
4    N current = target;
5    route.Add(current);
6
7    while (current.parent != null)
8    {
9      route.Add(current.parent);
10     current = current.parent;
11   }
12
13   route.Reverse();
14   return route;
15 }
```

Dijkstra - Example (1/2)

Exercise: using the graph shown below, compute the shortest distance from G to D using Dijkstra's algorithm.



Graph and solution from Ahlquist and Novak, ch. 6

Dijkstra - Example (2/2)

A B C D E F G H I J K, queue:
A B C D E F G0 H I J K, queue: G0
A B C D E Fg20 G0 H I J K, queue: Fg20
A B C D E Fg20 G0 Hg8 I J K, queue: Hg8, Fg20
Ah15 B C D E Fg20 G0 Hg8 I J K, queue: Ah15 Fg20
Ah15 B Ca22 D E Fg20 G0 Hg8 I J K, queue: Fg20 Ca22
Ah15 B Ca22 Df43 E Fg20 G0 Hg8 I J K, queue: Ca22 Df43
Ah15 Bc28 Ca22 Df43 E Fg20 G0 Hg8 I J K, queue: Bc28 Df43
Ah15 Bc28 Ca22 Dc30 E Fg20 G0 Hg8 I J K, queue: Bc28 Dc30
Ah15 Bc28 Ca22 Df30 Ec30 Fg20 G0 Hg8 I J K, queue: Bc28 Dc30 Ec30
Ah15 Bc28 Ca22 Df30 Ec30 Fg20 G0 Hg8 I J K, queue: Dc30 Ec30

Nomenclature:

- X: Not visited yet from origin.
- Xy15: The quickest way to get to X (from the origin) is through Y, and its associated cost is 15.
- The priority queue sorts by ascending cost.

Dijkstra is complete and always finds the shortest path. Also, it is efficient. However, we can often do better. **Q** How could we improve Dijkstra?

A*

Dijkstra uses the distance travelled so far and utilises a priority queue to consider nodes in the graph with the smallest distance from the start found so far.

We would expect to perform better if we could also guess how close a node is to the target (for point-to-point distances).

A* uses a **heuristic** to estimate the utility of a node with respect to the target. This allows the algorithm to explore the graph towards the more promising regions. For this to work, we require the following:

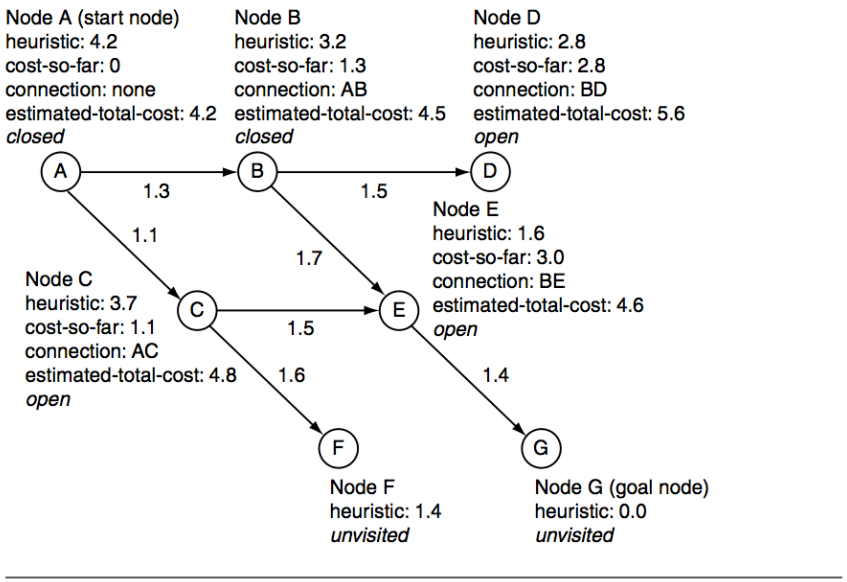
- The heuristic must be **admissible**.
- An admissible heuristic never over-estimates the distance from A to B
- The closer the heuristic to the actual distance, the better A* performs.

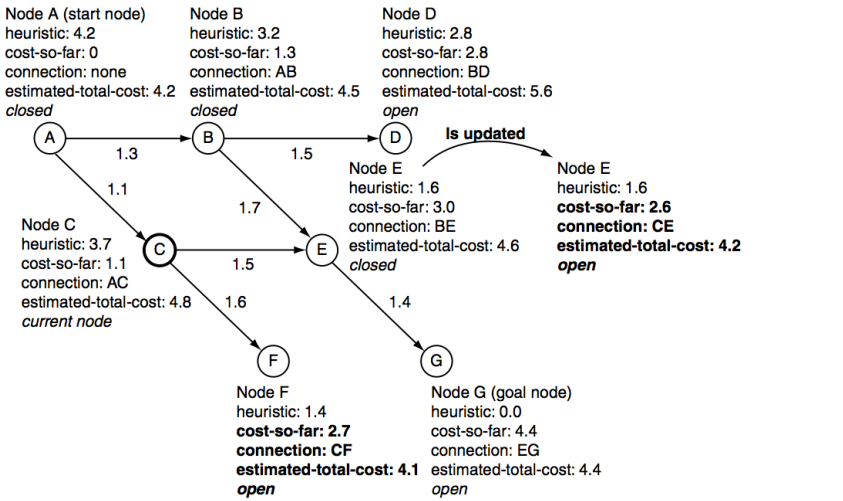
A* I

```
1  public enum Heuristics { STRAIGHT, MANHATTAN };
2
3
4  public static List<N> AStar(N start, N target, Heuristics heuristic)
5  {
6    PQ open = new PQ();
7    List<N> closed = new List<N>();
8    start.g = 0;
9    start.h = GetHValue(heuristic, start, target);
10   open.Add(start);
11
12   while (!open.IsEmpty()) {
13     currentNode = open.Get();
14     closed.Add(currentNode);
15
16     if (currentNode.isEqual(target)) break;
17
18     foreach (E next in currentNode.adj) {
19       double currentDistance = next.cost;
20
21       if (!open.Contains(next.node) && !closed.Contains(next.node)) {
22         next.node.g = currentDistance + currentNode.g;
23         next.node.h = GetHValue(heuristic, next.node, target);
24         next.node.parent = currentNode;
25         open.Add(next.node);
26       }
27       else if (currentDistance + currentNode.g < next.node.g) {
28         next.node.g = currentDistance + currentNode.g;
29         next.node.parent = currentNode;
30
```

A* II

```
31            if (open.Contains(next.node))
32              open.Remove(next.node);
33
34            if (closed.Contains(next.node))
35              closed.Remove(next.node);
36
37            open.Add(next.node);
38          }
39        }
40    }
41    return ExtractPath(target);
42  }
```

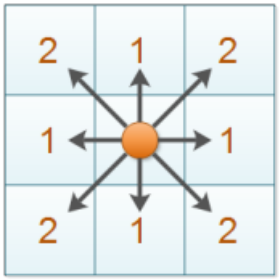




Heuristics

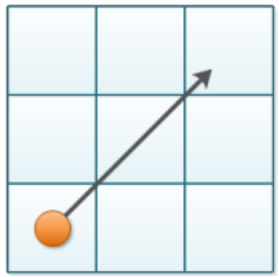
Some types of heuristic:

Manhattan Distance



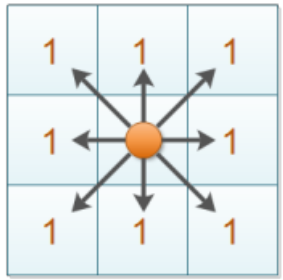
$$|x_1 - x_2| + |y_1 - y_2|$$

Euclidean Distance



$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Chebyshev Distance



$$\max(|x_1 - x_2|, |y_1 - y_2|)$$

(`lyfat.wordpress.com/2012/05/22/euclidean-vs-chebyshev-vs-manhattan-distance/`)

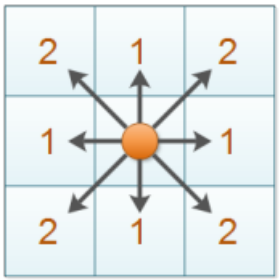
To use A*, we need a heuristic. In the simplest case, we can say $h = 0$. A* then behaves identical to Dijsktra. In order to improve the algorithm, we should choose an admissible heuristic as close as possible to the real distance.

- Rectangular grid, 4-way connectivity: which are admissible?
- Rectangular grid, 8-way connectivity: which are admissible?

Heuristics

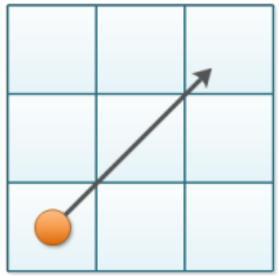
Some types of heuristic:

Manhattan Distance



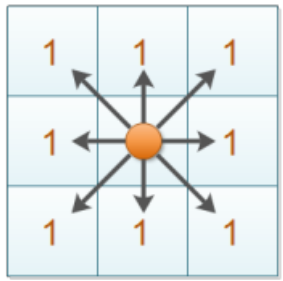
$$|x_1 - x_2| + |y_1 - y_2|$$

Euclidean Distance



$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Chebyshev Distance



$$\max(|x_1 - x_2|, |y_1 - y_2|)$$

(`lyfat.wordpress.com/2012/05/22/euclidean-vs-chebyshev-vs-manhattan-distance/`)

Admissible heuristics:

- Rectangular grid, 4-way: **Euclidean, Manhattan or Chebyshev.**
- Rectangular grid, 8-way: **Euclidean or Chebyshev (Diagonal).**
- In some cases we can also use a **pre-computed exact** heuristic.

Q What other attribute is important for the heuristic to be useful?

A*

Our helper functions / heuristics are defined as follows:

```
1  private static double GetHValue(Heuristics heuristic, N from, N to)
2  {
3    switch (heuristic)
4    {
5      case Heuristics.STRAIGHT: return StraightLineDistance(from, to);
6      case Heuristics.MANHATTAN: return ManhattanDistance(from, to);
7    }
8
9    return -1;
10 }
```

```
1  private static double EuclideanDistance(N from, N to)
2  {
3    return Math.Sqrt(Math.Pow((from.row - to.row), 2) + Math.Pow((from.
         column - to.column), 2));
4  }
```

```
1  private static double ManhattanDistance(N from, N to)
2  {
3    return Math.Abs(from.x - to.x) + Math.Abs(from.y - to.y);
4  }
```

Here you can add additional heuristics including those for other grids etc.

Some Facts about A*

Some facts from `theory.stanford.edu/~amitp/GameProgramming/Heuristics.html`:

- If $h(n)$ is 0, only $g(n)$ matters, and A* equals Dijkstra's algorithm.
- If $h(n)$ is never more than the actual distance, then A* is guaranteed to find a shortest path. The lower $h(n)$, the more node A* expands.
- If $h(n)$ is exactly equal to the actual distances, then A* will only follow the best path and never expand anything else.
- If $h(n)$ is sometimes greater than the cost of moving from n to the goal, then A* is not guaranteed to find a shortest path.
- If $h(n)$ is very high relative to $g(n)$, then only $h(n)$ plays a role, and A* turns into Best-First-Search.

`http://en.wikipedia.org/wiki/A*_search_algorithm`

Path Following

Once an optimal path from A to B has been found, the agent must follow it. There are several complications.

- Other agents or moving objects might occupy nodes in the graph
- The level might be fully dynamic
- Fog of War

Numerous solutions:

- Re-compute (parts of) the path
- Use large nodes that can be occupied by multiple agents
- Divert to closest available node
- Step through or side-step
- Plan all paths centrally, block nodes before they are occupied
- Minimise crossings of agents (use shortest distance)

Also need to bear in mind visual perception:

- Need to turn into the right direction before moving
- Movement should look natural and proportional to distance

Outline

Navigation Meshes

A **Navigation Mesh** (or **NavMesh**) is a data structure which is used to model the traversable areas of a virtual map.

- It is a collection of two-dimensional convex polygons (usually triangles) that define which areas of a virtual map are traversable by agents.
- Each polygon acts as a single node that links to other adjacent nodes.
- The polygons are **convex**: any point from inside the polygon can be reached in a straight line from any other point inside the same polygon.

Advantages:

- Creates shorter and more natural paths than traditional grid and waypoint-based pathfinding systems.
- Allows pathfinding algorithms to ignore static and dynamic obstacles.
- A wide variety of pathfinding algorithms can be modified and optimized for using navigation meshes.

Disadvantages:

- Not as efficient as grid and waypoint-based pathfinding systems.
- Procedural generation of a NavMesh can yield sub-optimal solutions.
- Their manual creation can be time-consuming and yield flawed solutions.

NavMesh vs. Grid/Waypoint-based Pathfinding (1/4)

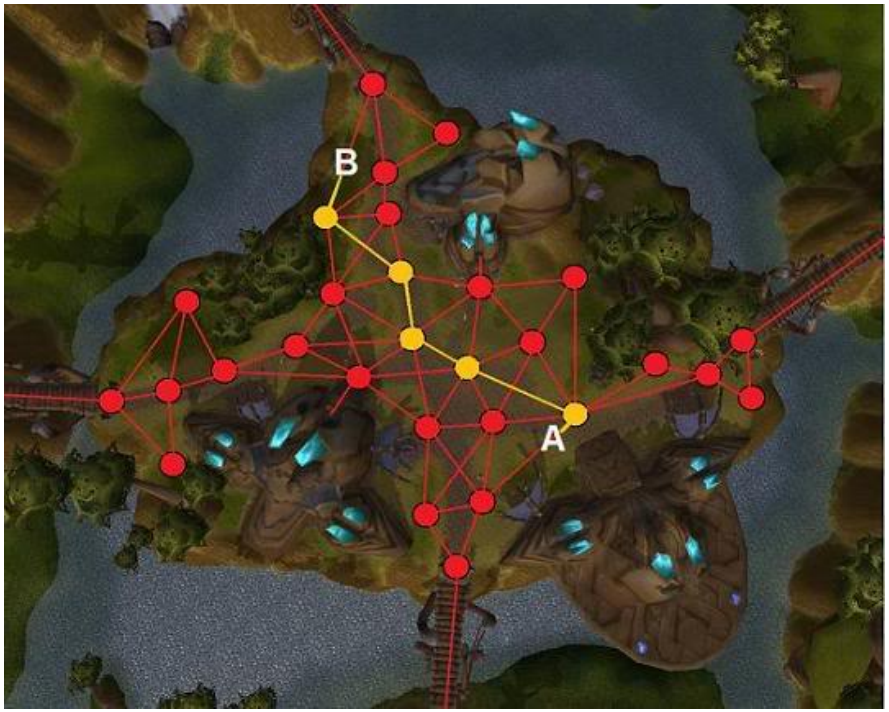
4 more advantages:

1) Waypoint-based graphs tend to require many more nodes, especially in large areas:

Source: `http://www.ai-blog.net/archives/000152.html`

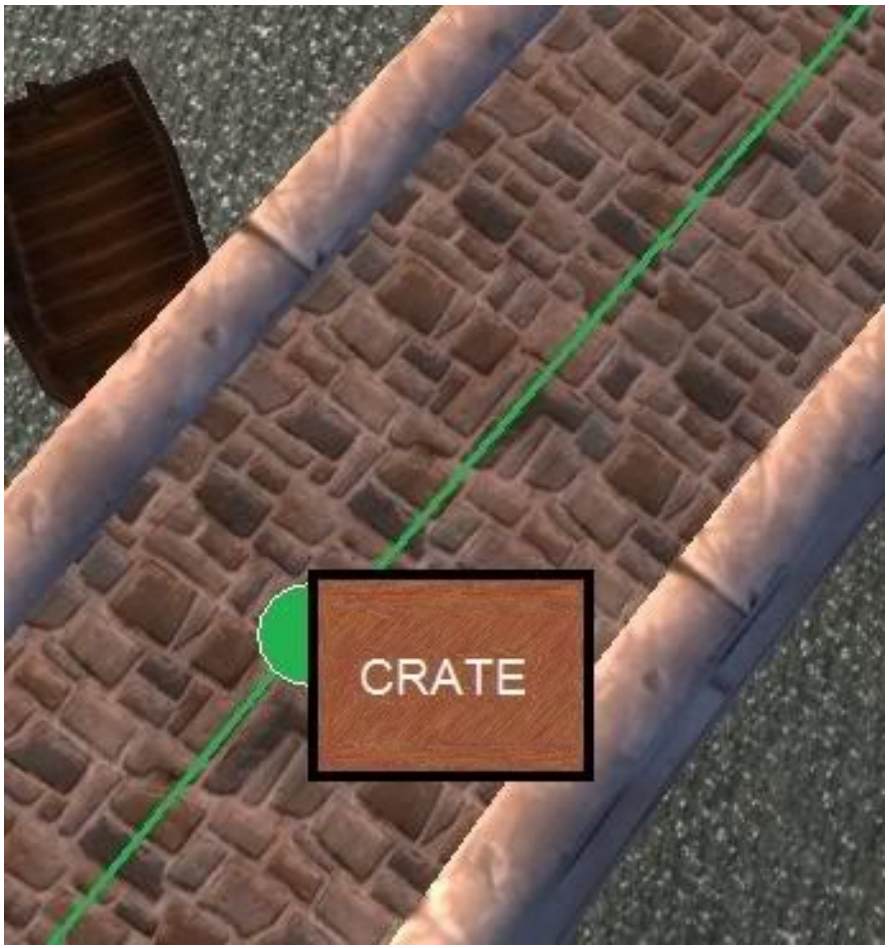
NavMesh vs. Grid/Waypoint-based Pathfinding (2/4)

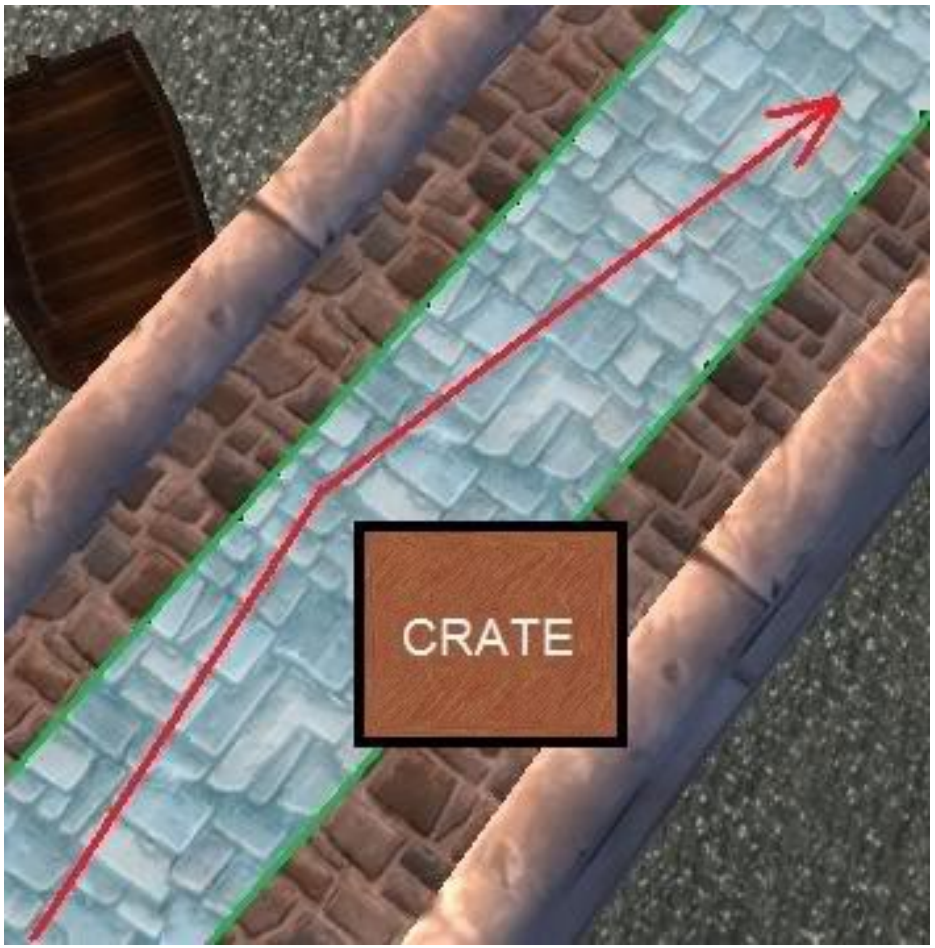
2) A non-natural zigzag movement: characters must adjust to the determined path.



NavMesh vs. Grid/Waypoint-based Pathfinding (3/4)

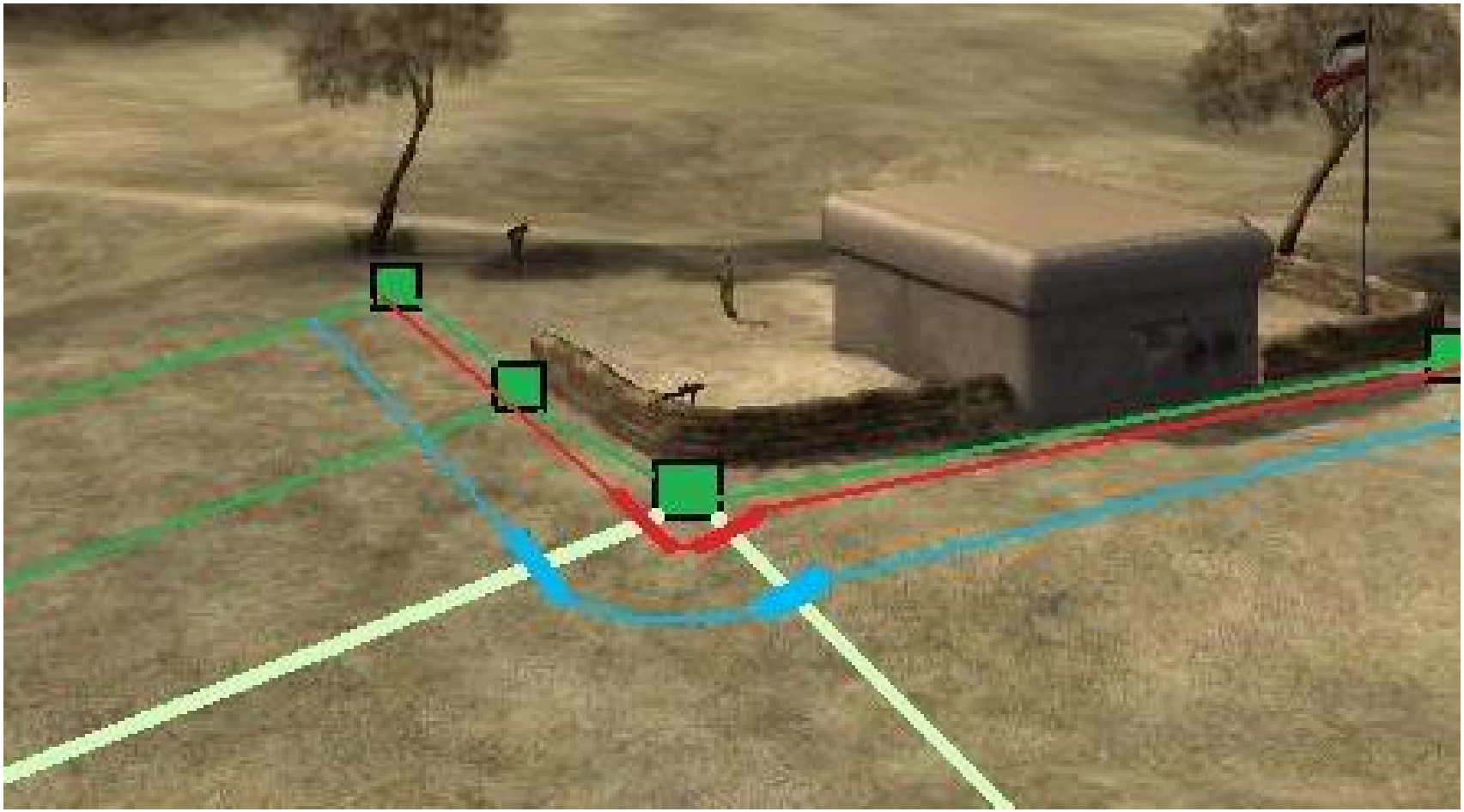
3) Robust dynamic obstacle avoidance:





NavMesh vs. Grid/Waypoint-based Pathfinding (4/4)

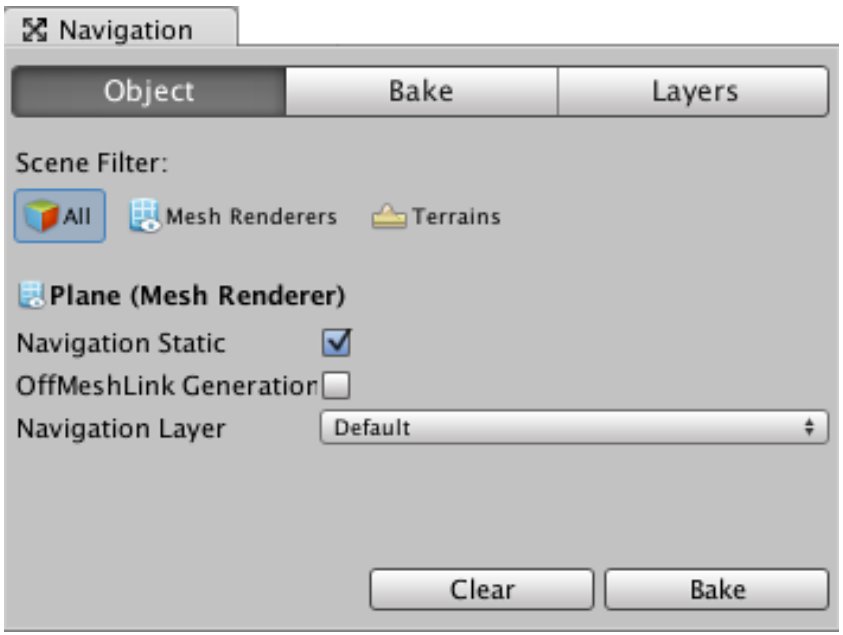
4) Smooth paths for different character sizes:



NavMesh in Unity - Objects

In Unity, NavMesh generation is handled from the Navigation window (*Window* $\rightarrow$ *Navigation*). The process of creating a NavMesh is called (again) **baking**.

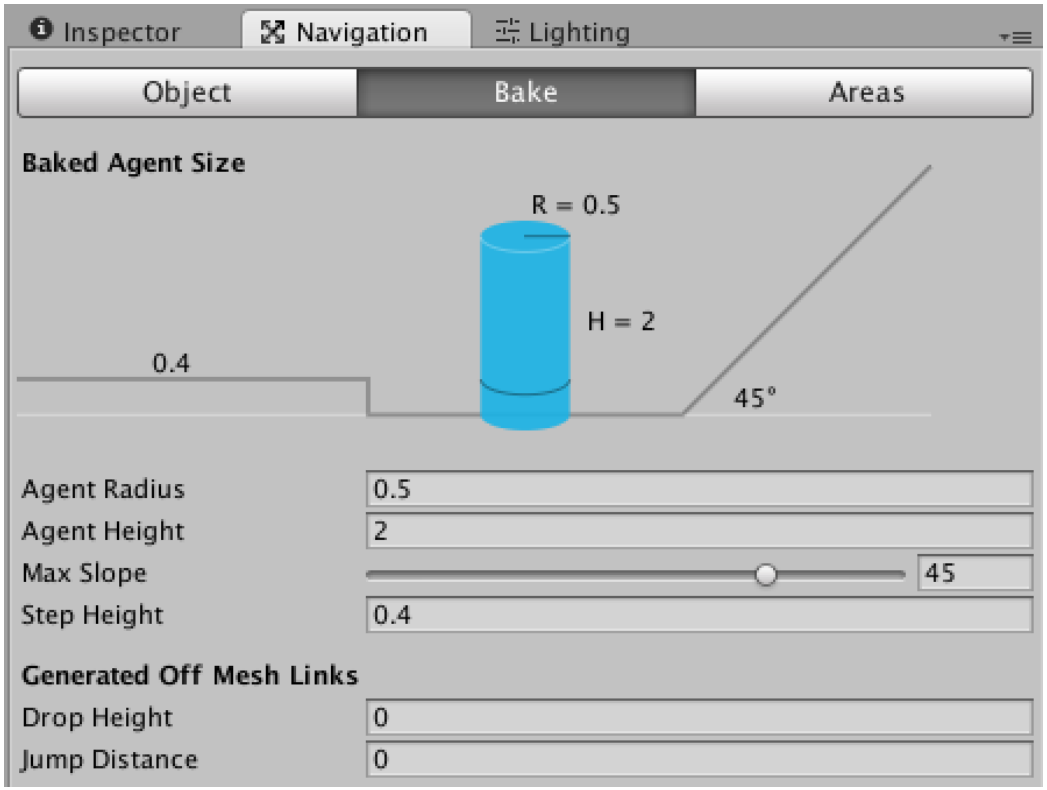
In order to bake a NavMesh, it is necessary to tell Unity which objects are *navigation static*. This objects won't move and can be used to calculate the navigable areas. Examples of these objects are floors, walls and static obstacles.



- Navigation static: uses the objects marked as *static*.
- OffMesh Link Generation: if Offmesh links will be created.
- Navigation Layer: Custom or Built-in layer for the mesh:
  - Default (Walkable).
  - Not Walkable.
  - Jump: for generated off-mesh links.

NavMesh in Unity - Baking

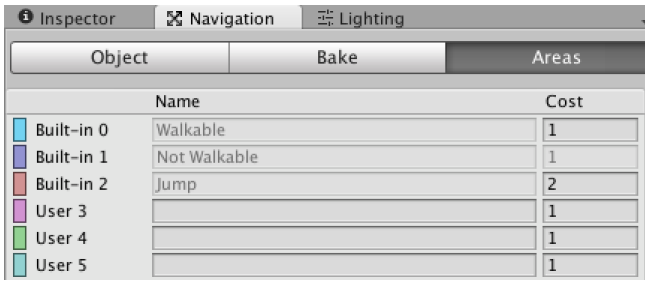
Baking of the NavMesh performed in the *Bake* pane, with objects selected.



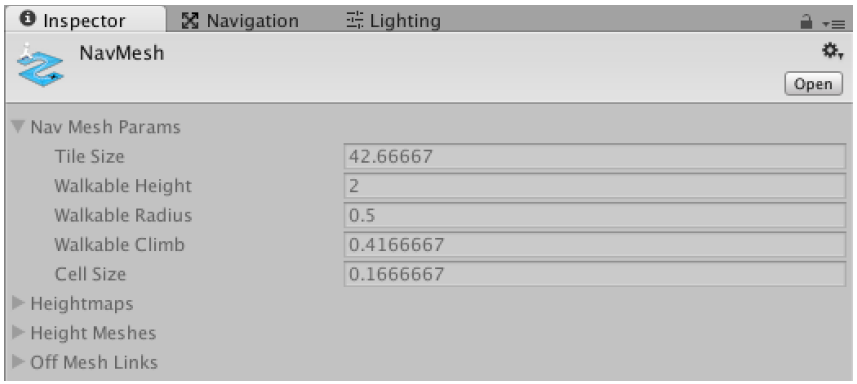
- *Radius:* how close a navigating character can get to a wall. Therefore if defines the narrowest path a character can traverse.
- *Height:* height of the ceiling above the navmesh surface.
- *Max Slope:* threshold of steepness where a ramp becomes a wall
- *Step Height:* maximum height of step that is ignored by the character.

NavMesh in Unity - Areas *

Different surfaces might have different associated costs. For instance, moving through snow or mud could convey a higher cost. In some situations, most of the ground will be "normal" but there might be some areas that are advantageous to cross. Each NavMesh section can be set up with a cost value through the use of Navmesh Areas.



The baked NavMesh object contains displays information about the mesh in the inspector:



NavMesh in Unity - The NavMesh Agent (1/4)

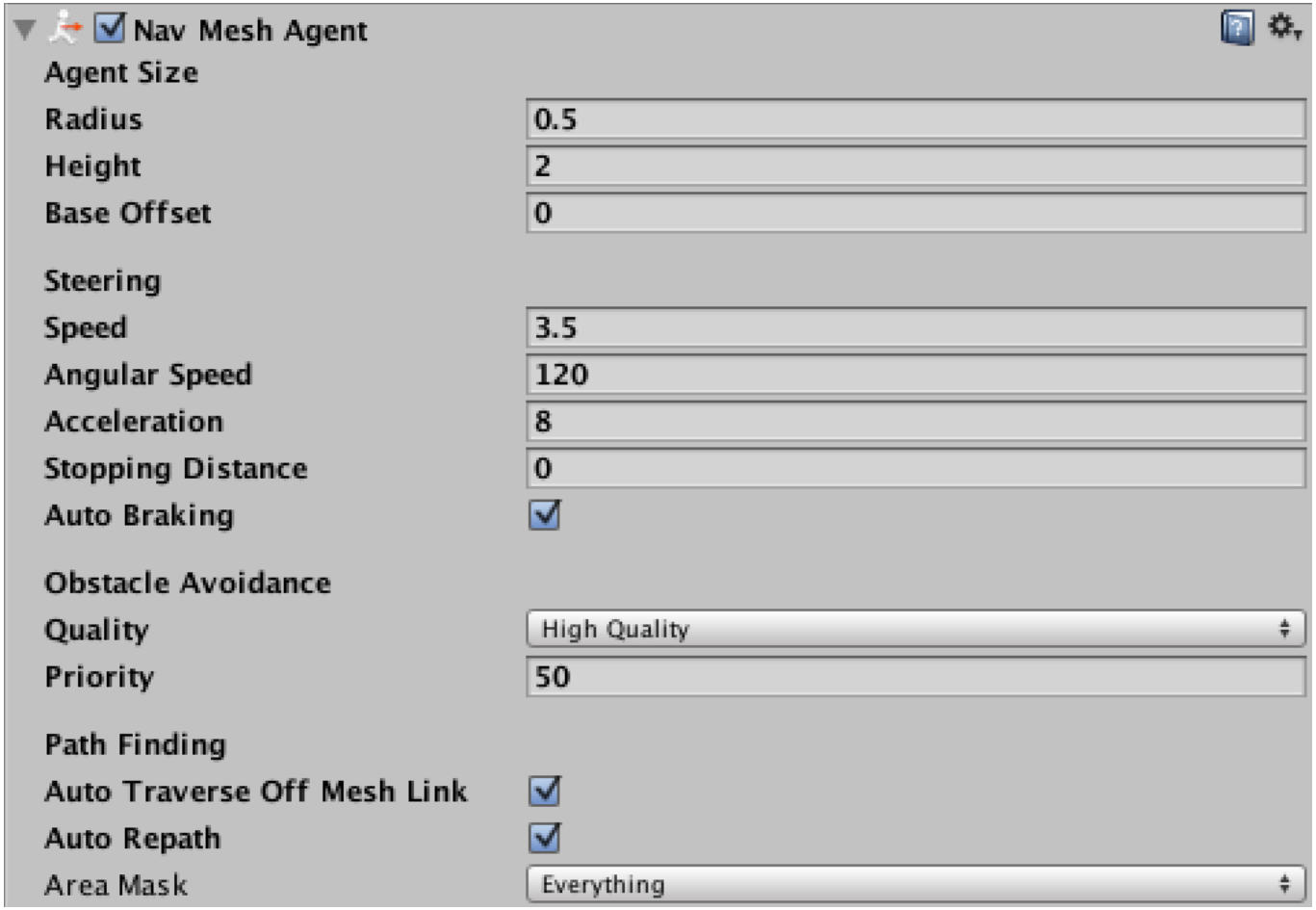
A NavMesh agent is a **component** represented by a cylinder with a *Radius* and *Height*. The agent will aim to keep a distance of *Radius* or greater between its centre and surrounding obstacles and cannot pass under a ceiling lower than the *Height* property.

The agent is able to query paths to the NavMesh, which are represented as a sequence of *waypoints* connected by straight lines that denote the shortest path. However, rather than follow this path perfectly, the agent will use more realistic motion that incorporates acceleration and gradual turning, specified by the *Speed*, *Acceleration* and *Angular Speed* properties).

As it moves around, an agent will avoid fixed obstacles in the scene as well as other agents. The *Obstacle Avoidance Type* gives a hint to the navigation system about how accurate the avoidance should be. The *Avoidance Priority* determines how the agent will behave when it encounters another agent - the agent with the larger priority number will avoid while the other will just follow its normal path.

NavMesh in Unity - The NavMesh Agent (2/4)

A **NavMesh Agent component** can be added to a character to allow it to navigate from place to place automatically over a Navigation Mesh. The Nav Mesh Agent component handles both the pathfinding and the movement control of a character.



NavMesh in Unity - The NavMesh Agent (3/4)

Radius: Radius around the agent within which obstacles should not pass.

Height: The height clearance the agent needs overhead.

Base offset: Height difference between the anchor point of the GameObject and the centre point of the agents cylinder.

Speed: Maximum movement speed (in world units per second).

Angular Speed: Maximum speed of rotation (degrees per second).

Acceleration: Maximum acceleration (in world units per second squared).

Stopping Distance: The agent will stop when this close to the goal location.

Auto Braking: Should the agent slow down as it approaches the target?

Obstacle Avoidance Quality: Approximate quality level for obs. avoidance.

Obstacle Avoidance Priority: Agents of lower priority will be ignored by this agent when performing avoidance.

Auto Traverse Off Mesh Link: Should the agent traverse Off Mesh links?

Auto Repath: Calculate a new path if the current one becomes invalid?

Area Mask: NavMesh areas that are walkable by this agent.

NavMesh in Unity - The NavMesh Agent (4/4) *

The NavMesh agent can be controlled directly from scripting. Retrieving the component is simple:

```
1 NavMeshAgent agent = GetComponent<NavMeshAgent> ();
```

And setting a path for the agent is specified calling *SetDestination* with a *Vector3* position:

```
1 agent.SetDestination (target.position);
```

Other useful functions of the NavMeshAgent class:

- *CalculatePath:* Calculate a path to a point and store the resulting path.
- *SetPath:* Assign a new path to this agent.
- *Raycast:* Trace a straight path towards a target postion in the NavMesh without moving the agent.
- *Stop, Resume:* Stops/Resumes the movement along the current path.
- *GetLayerCost*: Gets the cost for crossing ground of a particular type.

Finally, a path is described by the class *NavMeshPath*:

- *Vector3[] corners*: Waypoints of the path.
- *NavMeshPathStatus status*: *Complete* (path reaches destination), *Partial* (path cannot reach destination) or *Invalid*.

NavMesh in Unity - The NavMesh Obstacle

A **NavMesh Obstacle** is a movable object that must be avoided by a NavMesh agent (examples: doors that can be open, movable walls, etc). In order to mark a game object as a NavMesh obstacle, you need to add a **Nav Mesh Obstacle** component to the object.

The parameters *Radius* and *Height* specify the dimensions of the cylinder that blocks the agent's path.

There are two more parameters (*Move Threshold* and *Carve*) that are used to let the agent calculate a different path if the obstacle is in the way to the destination. If these are not used, the default behaviour is that the agent will try to go through the path with the obstacle anyway.

Outline

Lab Preview

We'll do more Assignment Part I demonstrations.

Next lecture: Game AI: Decision Making.