CE318: Games Console Programming

Lecture 6: Level Generation, Collision Detection & Game AI

Philipp Rohlfshagen

prohlf@essex.ac.uk
Office 2.529

Outline

Outline

Progress Test and Labs

1. First we will briefly go through the 10 most difficult questions of the progress test and discuss their answers.

2. Last lab, you were asked to implement an animated arm using models and hierarichal transformations. You may have discovered that:
   - SketchUp stores the location of the vertices, not their relative translations
   - Blender stores absolute translations and uses a different coordinate systems
   - This may depend on DCC or exporter (3D Rad is better)

   These factors have complicated the exercise considerably. You are thus encouraged to have a look at the artciulated arm using primitives, which is much simpler.

   Remember, the general idea is as follows:
   - Rotate the object around the appropriate pivot point
   - For this you may need to translate all parts appropriately
   - Accumulate all rotations and translations of the children to draw the parent

Outline

Creating Levels

We are now half-way though the course. The second half of the course will put everything we have covered so far into practice, building a game of increasing complexity.

In particular, in the next two lectures, we will develop a simple 3D game with the following attributes:

- A 3D maze that consists of a floor, walls and some other items
- The level is loaded from a text file and can thus be edited by hand
- A ship that can explore the 3D maze, collect items and shoot
- An enemy that uses simple AI in an attempt to kill you
- Different camera perspectives: first person and third person
- Different types of collision detection

We will add these attributes one by one and then start to visually improve the game (lecture 8). You can use this as the basis of your assignment.

Loading Levels

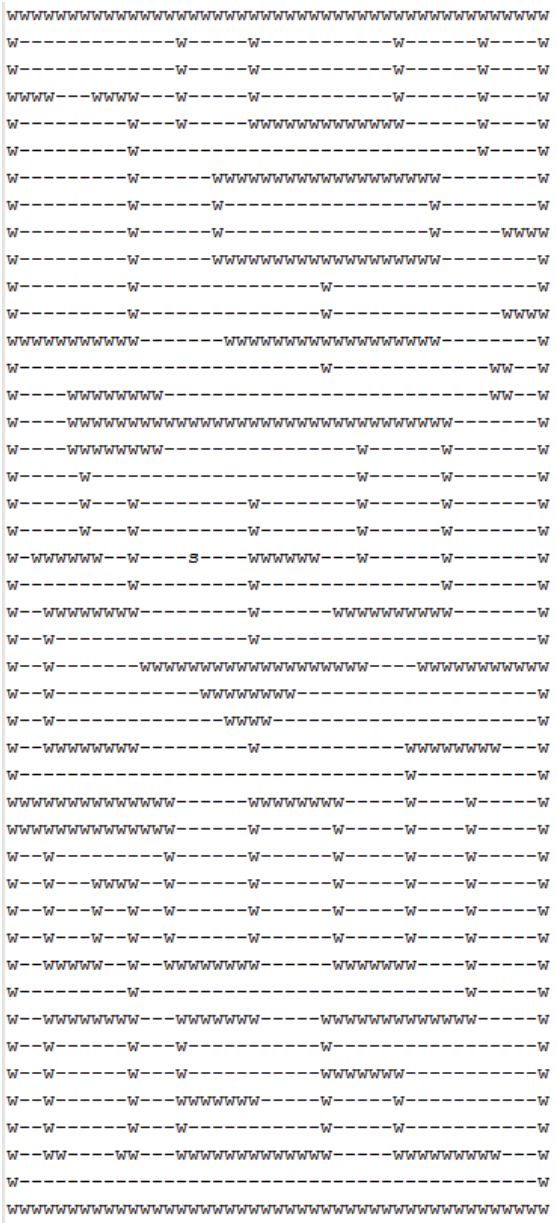
The idea is to have levels of certain size to be loaded from a plain text file. We use the following symbols to specify elements in the level:

- '-': an empty field
- 'w': a wall (we will use models for this)
- 't': half a wall, moved up
- 'b': half a wall, moved down
- 's': the ship (i.e., the player)
- 'e': the enemy (i.e., the opponent)
- 'p': pick-ups (could be anything)

We could simply load the text file from the game, by-passing the content pipeline. Instead, we will use a custom content importer to load and generate the level. For this, we need to create two projects:

- A Content Pipeline Extension
- A Game Library for shared code

Loading Levels



To create the level and custom content importer:

- Start a new project
- Create new Content Pipeline Extension Library
- Level importer goes here
- Add reference to content project
- Create a new Game Library
- Add reference to content and code projects
- Code for level goes here
- Use same namespace or use imports

Level Importer

Creates an instance of `Level` and then parses the text file and adds the symbols to the level's internal representation of the map.

```
1  [ContentImporter(".lev", DisplayName = "MyLevelImporter")]
2  public class MyContentImporter : ContentImporter<Level>
3  {
4    public override Level Import(string filename, ContentImporterContext
         context)
5    {
6      Level level = new Level();
7      int index=0;
8
9      StreamReader reader = new StreamReader(File.OpenRead(filename));
10
11     for (int i = 0; i < level.size; i++)
12     {
13       string current = reader.ReadLine();
14       StringReader sr = new StringReader(current);
15
16       char[] symbols = new char[level.size];
17       sr.Read(symbols, 0, symbols.Length);
18
19       for (int j = 0; j < symbols.Length; j++)
20         level.map[index++] = symbols[j];
21     }
22
23     return level;
24   }
25 }
```

Change from ContentProcessor template

Level

At first, `Level` can be basic but eventually it will be responsible for all objects in the world. We will be adding additional methods as required. Important to take an OO approach to maximise code readibility, re-use and ease of maintanence.

```
1  public class Level
2  {
3    public const int size = 45;
4    public char[] map;
5
6    public Level()
7    {
8      map = new char[size * size];
9    }
10 }
```

The size is fixed although this could / should be part of the text file. Once the characters have been passed to the level, we can use that information to create the 3D maze.

Note: we are using a 1D arrays. In languages like Java, this is more efficient than 2D arrays. C#, however, uses rectangular arrays. Nevertheless, 2D arrays require some more work with the content importer due to serialisation issues.

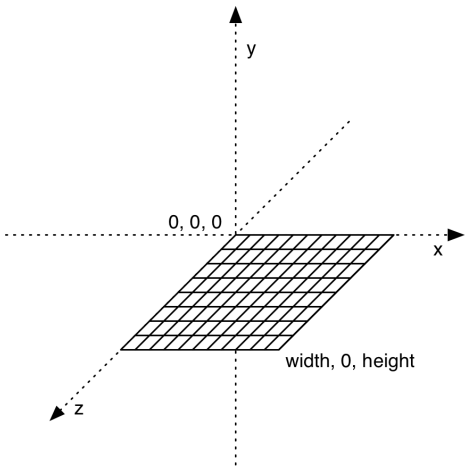
Level

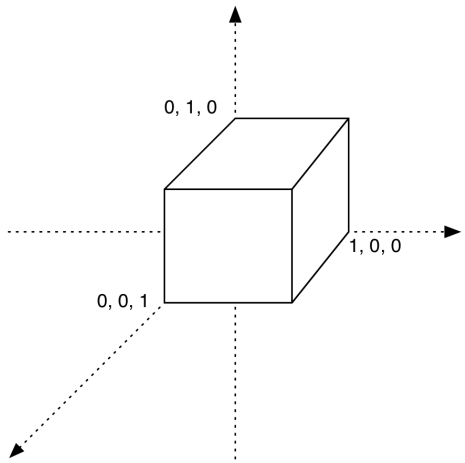
First we must decide on how to layout the level in 3D space and what units to use. We choose as origin (0, 0, 0) and expand along the positive z and x axes.

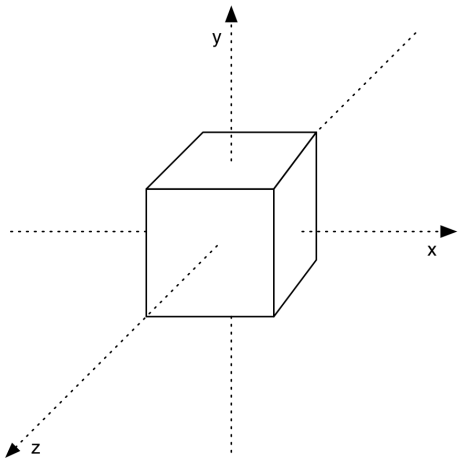
- Our world will be based on a grid, each cell of size 1×1.
- We will use SketchUp to create the models
- Each wall is of size $1m \times 1m \times 1m$

With these conventions, we can use the x-y coordinates directly to place the walls in the world. To convert from a 1D array to a 2D array, we can use:

```
1 Matrix.CreateTranslation(new Vector3((i % size), 0, (i / size)));
```







Level

The floor is also a model. We create a thin $1m \times 1m$ box and then scale it along x and z depending on the size of the map. For now we will only consider square levels such that size = width - height.

We only load the wall model once and then use multiple world transforms to place the same geometry at different locations (note: XNA always reuses the same model anyway).

Wer can use some helper tools to aide the process of placing everything correctly. The code includes

- A coordinate system to indicate the origin
- A grid to indicate where the cells are

Cameras

We will make use of 2 camera types:

- A simple (almost first-person) chase camera
- A third person chase camera

We make use of a camera class with the following constructor:

```
1 public Camera(GraphicsDevice device, Vector3 position, Vector3 target,
      Vector3 up, float angleOfView, float nearPlane, float farPlane)
```

which has an `Update()` method:

```
1 public void Update(Vector3 position, Vector3 target, Vector3 up)
```

Assume a `Matrix shipMatrix` that holds the ship's translation and rotation.

We then define:

```
1 camera1.Update(shipMatrix.Translation + (1f * -look) + (0.5f *
      shipMatrix.Up), shipMatrix.Translation + shipMatrix.Forward,
      shipMatrix.Up);
```

and

```
1 camera2.Update(shipMatrix.Translation + new Vector3(5, 8, 5),
      shipMatrix.Translation, shipMatrix.Up);
```

Outline

Collision detection

Collision detection is one of the most essential features of any video game. We will look at three different approaches to collision detection. Next lecture, we will focus on the issue of efficiency.

1. Using the map used to generate the world
2. Bounding spheres
3. Bounding boxes

The simplest approach to collision detection in our case is to use the 2D map itself: the agent has coordinates in 3D space, two of which translate directly to the map (i.e., x and z). Since we know where the boxes are, we can check whether the coordinates intercept any of the objects that make up the world.

Q How do we get the coordinates from the ship's world matrix?
Q What are the biggest drawbacks of this approach?

Bounding Spheres & Boxes

When designing a game, it is often useful to see where specific things are in the world, such as meshes, coordinates, etc. Microsoft provides a nice class called `DebugShapeRenderer` that allows one to easily draw bounding spheres, bounding boxes and bounding frustims onto the screen.

To use this class, call

```
1 DebugShapeRenderer.Initialize(GraphicsDevice);
```

then make use of

- `DebugShapeRenderer.AddBoundingSphere()`
- `DebugShapeRenderer.AddBoundingBox()`
- `DebugShapeRenderer.AddLine()`
- `DebugShapeRenderer.AddTriangle()`

Can add objects for single frame or for a duration. We will see examples of this shortly.

Bounding Spheres

We already mentioned `BoundingSphere` in lecture 4. Remember:

- Each mesh of the model has its own bounding sphere
- To get the model's bounding sphere, merge all spheres together
- Need to translate and scale sphere by the mesh's transform

Once every object has its correct bounding sphere, we can use

- `BoundingSphere.Contains()`
- `BoundingSphere.Intersects()`

to check for collisions.

```
1  BoundingSphere transfrmdShip = shipSphere.Transform(shipTranslation);
2
3  for (int i = 0; i < wallWorlds.Count(); i++)
4  {
5    BoundingSphere transfrmdWorld = wallSphere.Transform(wallWorlds[i]);
6
7    if (transfrmdWorld.Intersects(transfrmdShip))
8      return true;
9  }
10
11 return false;
```

Q What might be problematic with the use of bounding spheres?

Bounding Boxes

A sphere is very useful to approximate the shape of complex objects (e.g., spaceship with wings etc.). However, in some cases, a bounding box is a much better fit (in the case of our walls, we can get a perfect fit). Unfortunately, XNA does not provide a readily available bounding box. There are two solutions to this:

- Create a bounding box from a bounding sphere using

  `BoundingBox.CreateFromSphere()`
- Create a bounding box from 2 points and try to make it fit the object

  `BoundingBox box = new BoundingBox(Vector3 min, Vector3 max);`

We will make use the latter. Since we know the size and location of each wall segment, we can simply create a bounding box as follows:

```
1 Vector3 min = new Vector3(0f);
2 Vector3 max = new Vector3(1f);
3 from = Vector3.Transform(min, objectWorld);
4 to = Vector3.Transform(max, objectWorld);
5 BoundingBox box = new BoundingBox(min, max);
```

Outline

Game AI vs Computer Science AI

All of our non-player characters (NPCs) require some mechanism to act in the world. The primary goal of this behaviour is to **engage** the player.

There are fundamental differences between game AI and computer science AI (see Ahlquist and Novak, ch. 1):

- Computer science AI is about substance: **act intelligently**
  - Minimax, $\alpha\beta$, Monte Carlo tree search, neural networks,
  - Bayesian (belief) networks, temporal difference learning, behaviour trees
- Game AI is about appearances: **appear intelligent**
  - Scripting, triggers, animations, path-finding using classical algorithms
  - Rule-based systems, expert systems, finite state machines

The simplicity of game AI had some important consquences:

> *"The increasing number of multi-player online games (among others) is an indication that humans seek more intelligent opponents and richer inter-activity."* *Yannakakis and Hallam (2005)*

Game AI vs Computer Science AI

Why does game AI not take full advantage of the advances in CS AI? There are many reasons (although recently, more game developers pay more attention to these issues):

- More costly to develop, debug and maintain

However, there are other reasons too. First, game AI can cheat:

- Know where the player is at all times (e.g., ignore Fog of War)
- Move faster, build units faster, increase resources automatically
- Bias random events

Such shortcuts are often visible to the player and may have a negative impact on gameplay.

Another issue is the complexity of video games. CS AI is often applied to much simpler games (e.g., board games). In contrast, video games pose many difficult challenges:

- Time constraints, simultaneous moves, large number of players
- Complex rules (huge branching factors), open-endedness

Pathfinding

Amongst the most fundamental game 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:

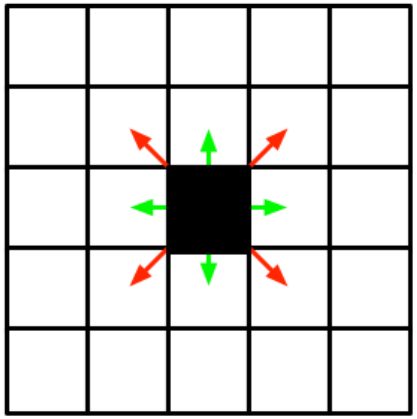
- Nodes represent points in 2D or 3D space
- Nodes have neighbours (adjacent nodes)
- Neighbouring nodes may have different distances
- Nodes (i.e., their locations) can be constructed manually or automatically

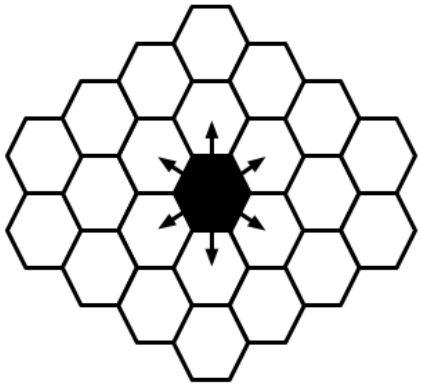
The set of nodes that make up the graph are known as the pathfinding network.

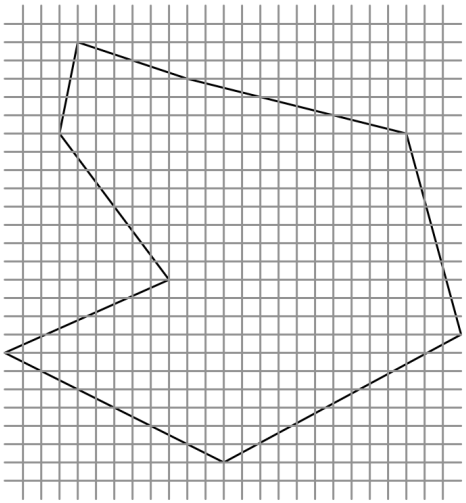
We need a way to construct the graph.

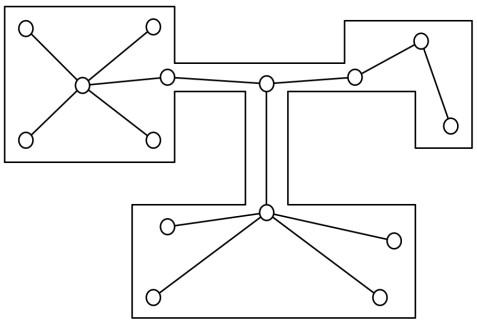
- Discretise the continuous space using grids
- Simplest: regular grids (rectangular or hexagonal)
- Rectangular grid: 4- or 8-way connectivity

Pathfinding









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

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 **Dijstra'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 $G = (V, E)$. 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 vertex to another, with a given cost
- N: a node (vertex) with attributes required by the pathfinding algorithms

The edge is defined as follows:

```
1  public class E
2  {
3    public N node;
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();
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

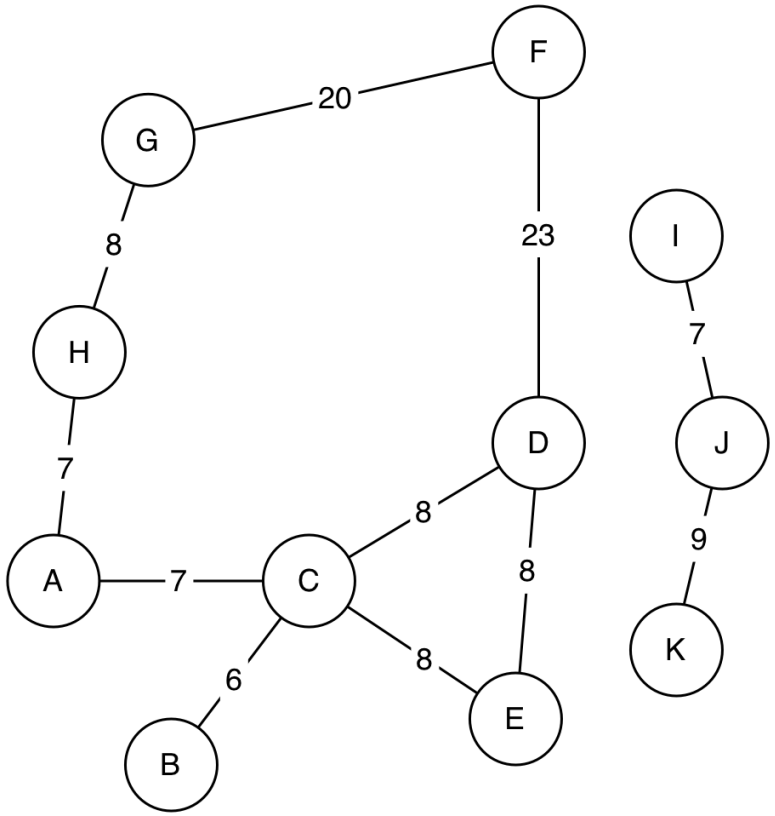
The route can be extracted easily by following the chain of parents from the target to the start. We then reverse the route to return the path from start to finish.

```
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 }
```

Q How can we use Dijstra to build a complete look-up table?

Dijkstra

Class 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

Solution

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 Df30
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

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?

Outline

Summary & Lab Preview

In today's lecture we put together some of the concepts covered in the first half of the course to design and build a customisable 3D world. The world contains wall, pick-ups, an agent (the player) and an enemy that uses simple pathfinding AI to locate the player. Next lecture, we will add interactions (shooting) and later, we will add more visuals to make the game more appealing.

We covered

- Content importers
- Level design
- Level construction
- Created a 3D maze
- Collision detection
- Game AI
- Pathfinding algorithms
- Path following

In the lab you will be will be asked to

- Create some interesting mazes using text files
- Load the mazes and create the level
- Create a searchable graph from the map
- Implement some basic opponent AI behaviour