CE318: Games Console Programming
Lecture 7: Advanced Game AI & Gameplay

Philipp Rohlfshagen

prohlf@essex.ac.uk
Office 2.529

Outline

Lecture Overview

In today's lecture, we will explore the same themes as in lecture 6 but looking at more advanced aspects. We will use this to re-write, improve and optimise the game code we developed in the last lecture.

1. First we will look at importing the level using both custom importers and processors (this includes type readers and writers). This gives us more flexibility and allows us to use 2D arrays for the map from the onset.

2. We then finish off path finding (A*) and look at behaviour trees to generate some interesting opponent behaviours. You will be asked to use A* in the labs.

3. Finally, we will add some interactivity to the game: shooting. This includes the AI to aim and shoot at the gamer as well as a brief discussion of hierarchical collision detection.

Outline

Importing Custom File Types

The XNA content pipelines makes it easy and efficient to deal with a variety of assets. It supports numerous popular file formats. However, there is often the need to import custom files, such as level descriptions, which can be quite complex.

For this, we require:

- Content Importer – associated with the file extension we create
- Content Processor – parses the data and builds the asset
- Type Writer – writes XNA *.xnb* files
- Type Reader – reads *.xnb* files
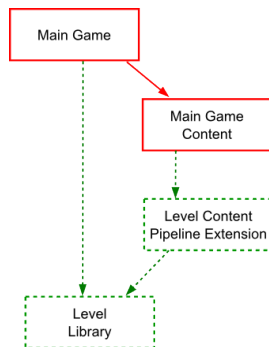
Great tutorial at: `rbwhitaker.wikidot.com/content-pipeline-extension-1`

Using a Content Processor

We create a new file type, *.map*, which has the following format:

```
rows
columns
wwwwww
w----w
w----w
wwwwww
```

We will use the same symbols as before. Feel free to change this in your code and extend it as you see fit.

One idea: use numbers for the walls to have walls of variable height and a ship that can move smoothly up and down as well.

Since we will use a processor, our content importer can be extremely simple:

```
1 using TImport = System.String;
```

```
1 [ContentImporter(".map", DisplayName = "Map Importer",
      DefaultProcessor = "MapProcessor")]
2 public class MapImporter : ContentImporter<TImport>
3 {
4   public override TImport Import(string filename,
        ContentImporterContext context)
5   {
6     return System.IO.File.ReadAllText(filename);
7   }
8 }
```

Note the declarations of TImport, TInput, TOutput, TWrite, TRead

Content Processor

The content processor is responsible for creating the data type `Map`:

```
1 using TInput = System.String;
2 using TOutput = MapGameLibrary.Map;
```

```
1 [ContentProcessor(DisplayName = "Map Processor")]
2 public class MapProcessor : ContentProcessor<TInput, TOutput>
3 {
4     public override TOutput Process(TInput input,
            ContentProcessorContext context)
5     {
6         string[] lines = input.Split(new char[] { '\n' });
7         int rows = Convert.ToInt32(lines[0]);
8         int columns = Convert.ToInt32(lines[1]);
9         char[,] mapData = new char[rows, columns];
10
11        for (int row = 0; row < rows; row++)
12        {
13            StringReader sr = new StringReader(lines[row + 2]);
14            char[] symbols = new char[columns];
15            sr.Read(symbols, 0, columns);
16
17            for (int column = 0; column < columns; column++)
18            {
19                mapData[row, column] = symbols[column];
20            }
21        }
22        return new MapGameLibrary.Map(mapData);
23    }
24 }
```

Using a Type Writer

Once the object has been processed, we need to write it an *.xnb* file. This is done using a contet type writer.

```
1 using TWrite = MapGameLibrary.Map;
```

```
 1 [ContentTypeWriter]
 2 public class MapTypeWriter : ContentTypeWriter<TWrite>
 3 {
 4   protected override void Write(ContentWriter output, TWrite map)
 5   {
 6     output.Write(map.Rows);
 7     output.Write(map.Columns);
 8
 9     for (int row = 0; row < map.Rows; row++)
10     {
11       for (int column = 0; column < map.Columns; column++)
12       {
13         output.Write(map.Get(row, column));
14       }
15     }
16   }
17
18   public override string GetRuntimeReader(TargetPlatform
          targetPlatform)
19   {
20     return "MapGameLibrary.MapReader, MapGameLibrary";
21   }
22 }
```

Using a Type Reader

Finally, we need a type reader (as specified by the writer):

```csharp
1  using TRead = MapGameLibrary.Map;
```

```csharp
1  public class MapReader : ContentTypeReader<TRead>
2  {
3    protected override TRead Read(ContentReader input, TRead
         existingInstance)
4    {
5      int rows = input.ReadInt32();
6      int columns = input.ReadInt32();
7      char[,] mapData = new char[rows, columns];
8
9      for (int row = 0; row < rows; row++)
10     {
11       for (int column = 0; column < columns; column++)
12       {
13         mapData[row, column] = input.ReadChar();
14       }
15     }
16
17     return new Map(mapData);
18   }
19 }
```

This seems like a lot of work and it is - for something as simple as our *.map*. However, levels in more complex games require a much more detailed description and hence more processing.

Some Notes

It is important to get all relations correct:

- The individual projects must reference each other properly
- The importer references the file type, the processor the importer etc.
- Each file needs to specify the data type with `using`
- Importer, processor and writer go into the pipeline extension
- The reader goes into the game library

We will use this approach to load the behaviour trees from xml files later on.

When you add a model that has a texture already attached, you usually need to load the texture into the ContentPipeline as well. However, to prevent the texture from being loaded twice, you then need to exclude it from the project.

Some of you had problems with the DLL files in the lab. The solution is to save the object on the PC's local drive. The DLL corresponds to the game library which contains code without an entry point. You can't execute the libraries code but different project can access it (code common to multiple projects).

Outline

A*

Last lecture we covered basic path finding using Dijkstra's algorithm. 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 **admissable**.
- An addmissable heuristic never over-estimates the distance from A to B
- The closer the heuristic to the actual distance, the better A* performs.

Q Why is it important not to overestimate the cost/distance to the target?

A* I

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

A* II

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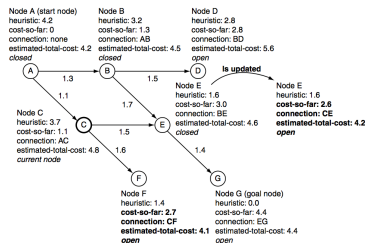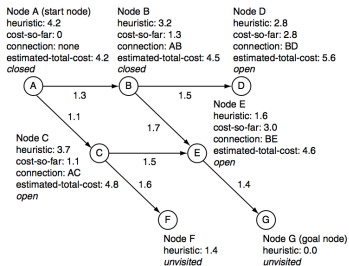
Heuristics

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 admissable heuristic as close as possible to the real distance.

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

Rectangular grid with 4-way connectivity:

- Straight-line / Euclidean distance
- Manhattan distance

Rectangular grid with 8-way connectivity:

- Straight-line distance / Euclidean
- Diagonal/Chebychev distance

Q What would be an inadmissable heuristic in these cases?

In some cases we can also use a pre-computed exact heuristic.

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 StraightLineDistance(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.

Q What happens if we leave out the square root of the Euclidean distance?

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.

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

Decision Making

Path following is but one of the many attributes required by NPCs to engage the gamer. Another important aspect is **decision making**. This may be achieved in numerous different ways:

- Decision trees
- Finite state machines / hierarchical finite state machines
- Behaviour trees / behaviour-oriented design
- Neural networks
- Fuzzy logic
- Rule-based systems

In many cases, decision making is **scripted**. Also, numerous tools exist to support decision making architectures, such as *influence maps*.

Today we will look at **behaviour trees** (BTs).

Behaviour Trees

For this topic, we will use the notation and examples from Millington and Funge (ch. 5). You are encouraged to read this chapter prior to next week's lab.

Behaviour trees have become very popular in recent years (starting 2004) as a more flexible, scalable and intuitive way to encode complex NPC behaviours. One of the first popular games to use BTs was Halo 2.

Some of the advantages of BTs are as follows:

- Can incorporate numerous concerns such as path finding and planning
- Modular and scalable
- Easy to develop, even for non-technical developers
- Can use GUIs for easy creation and manipulation of BTs

Tasks can be composed into sub-trees for more complex actions and multiple tasks can define specific behaviours.

A Task

A task is essentially an activity that, given some CPU time, returns a value.

- Simplest case: returns success or failure (boolean)
- Often desirable to return a more complex outcome (enumeration, range)

Task should be broken down into smaller complete (independent) actions.

A basic BT consists of the following 3 elements:

- **Conditions**: test some property of the game
- **Actions**: alter the state of the game; they usually succeed
- **Composites**: collections of child tasks (conditions, actions, composites)

Conditions and actions sit at the leaf nodes of the tree. They are preceded by composites. Actions can be anything, including playing animations etc.

Composite tasks (always consider child behaviours in sequence):

- **Selector**: returns success as soon as one child behaviour succeeds
- **Sequence**: returns success only if all child behaviours succeeds

Two simple examples of selectors and sequences:
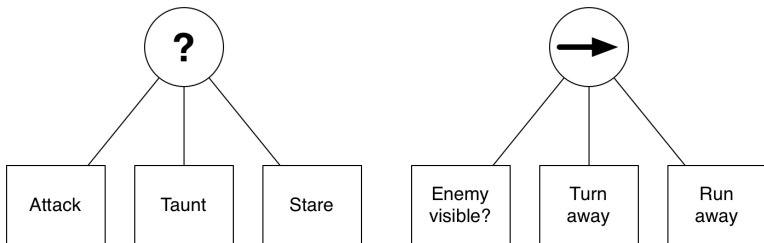
Q What do Selector and Sequence correspond to in term of logical operands?

Order of Actions

We can use the order of actions to imply priority:

- Child actions are always executed in the order they were defined
- Often a fixed order is necessary (some actions are prerequisites of others)
- Sometimes, however, this leads to predictable (and boring) behaviour

Imagine you need to obtains items A, B and C to carry out an action. If the items are independent of one another, one obtains a more diverse behaviour if these items are always collected in a different order.

We want to have **partial ordering**: some strict order mixed with some random order. We thus use two new operators, random Selector and random Sequence.

Note: you can see from this that the design of behaviours does not only have to consider functionality (i.e., getting the job done) but also gameplay experience.
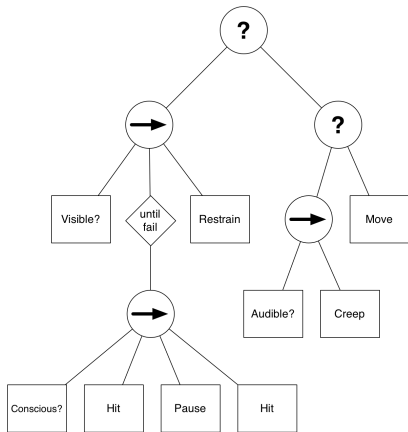
Partial Order Tree
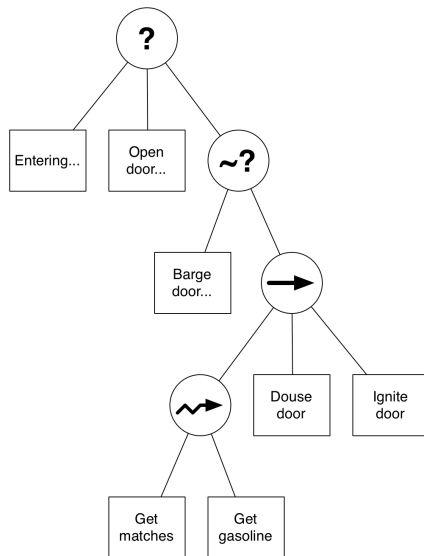
Decorators

Decorators add extended functionality to an existing task. A decorator has a single child whose behaviour it modifies. A popular usage is **filters**: if and how a task may run.

For instance, we can repeatedly execute a task until some goal has been achieved.

Decorators can also be used to invert the return value of an action.

Resource Management using Decorators

Individual parts of a BT often need to access the same resource (e.g., limited number of pathfinding instances, sounds). We can:

- Hard-code a resource query into the behaviour
- Create a condition task to perform the test using a Sequence
- Use a Decorator to guard the resource

Decorators can be used in a general approach such that the designer does not need to be aware of the low level details of the game's resource management.

We can use *semaphores*: a mechanism to ensure a limited resource is not over-subscribed. A decorator is subsequently associated with a semaphore which in turn is associated with the resource.

Can use a semaphore factor with unique name identifies – these can be used by both programmers and behaviour designers.

Concurrency and Timing

We need to take the time it takes to execute an action into account. If we execute each BT in its own thread, we can:

- Wait for some time for actions to finish (put the thread to sleeping)
- Timeout current tasks and return outcome (e.g., failure)
- Don't execute an action for some time after its last execution

Most importantly, taking care of concurrency allows for **parallel** tasks.

Sequence, Selector and Parallel form the backbone of most BTs.

Parallel is similar to sequence but runs all tasks simultaneously. If one task fails, all other threads are asked to terminate (**note**: resources must be released etc. so one has to implement this properly). Can also implements this as a Selector.

Uses of parallel:

- Execute multiple actions simultaneously such as falling and shouting
- Control a group of characters: combine individual and group behaviours
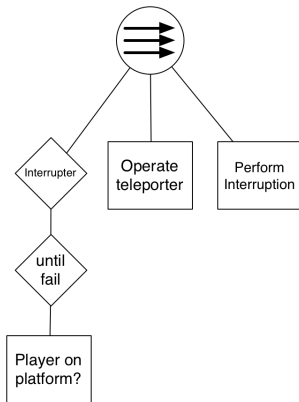- We can also use parallel for continuous condition checking
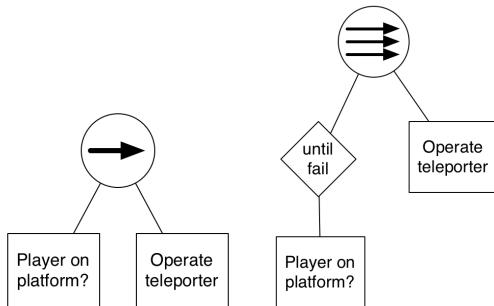
Q What is the difference between these trees?
Q What is wrong with the parallel tree?

Often we need intra-task behaviour: the ability of behaviours to affect one another (horizontal interaction). We do this using a decorator and a new task that communicates with the decorator.

Data

In addition to intra-behaviour communication, we often need to refer to (shared) data to carry out the actions of a behaviour. For instance, the action "open door" needs to refer to a particular door (and room).

The best approach is to decouple the data from the tree (rather that, say, use arguments to the actions). The central external data structure is called a **blackboard**. It can store any kind of data and responds to any type of request. This allows to still write independent tasks but which may communicate with one another.

It is common for sub-trees to have their own blackboards.

Reuse of Behaviours

It is apparent that BTs, if designed properly, can be quite modular and hence it makes sense to re-use whole or partial trees. Actions and conditions can be parameterised and data can be exchanged via the blackboard. This allows one to re-use trees if agents require identical behaviours. This can be done in numerous different ways:

- Use a cloning operation
- Create a behaviour tree library
- Use a dictionary and reference names to retrieve trees/sub-trees

Note on implementation: we can use the design tool to create the tree. We then need to create the actions and conditions. Each will be a class that carries out the required logic. Choose the names appropriately to identify the right objects during the creation of the tree.

Behaviour Oriented Design

Behaviour trees may be seen as one instance of Behaviour Oriented Design (BOD) which is also used frequently in robotics and models of animal behaviours. Hereis an example of POSH (simulating a group of monkeys):
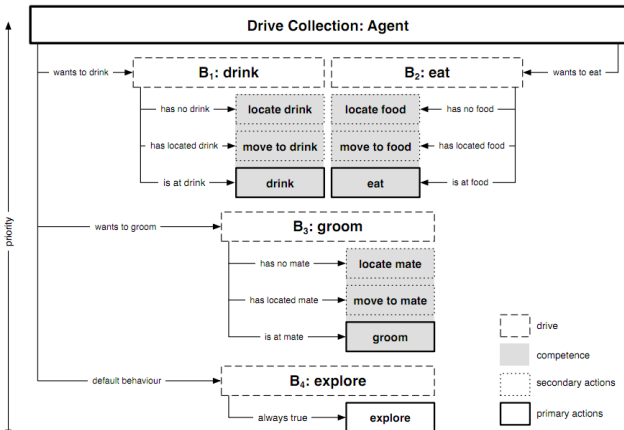
Outline

Using the 3D Maze in a Game

To make use of the 3D world we have developed so far, we still require two features to create a game:

- Interactions between enemy and player
- A goal / mission the player must reach / complete

First, we extend our game to interact with the opponent. We do this by allowing both sides to shoot at one another.

For this we require:

- Projectiles and possibly a crosshair
- Collision detection for all the projectiles
- A concept of health / injury / damage
    - Different degrees of damage depending on impact of projectile
- Ammunition (unlimited, or supplied via pick-ups)
- An enemy that is able to locate, aim and shoot at us

Gamer Shooting

We can use a simple model for the projectiles which are added to the game whenever a specific key has been pressed.

Q What information is required by the projectile to update it properly?

We then use bounding spheres for each projectile and check consistently for collisions. If one occurred, we update the damage status of the ship or enemy (if that caused the collision) and/or remove the projectile from the list of active projectiles.

It is more difficult to make the enemy shoot at the gamer since we need to provide all required behaviours, including pathfinding, aiming and shooting. We can also introduce different levels of difficulty here:

- Speed of the enemy
- Degree of armour
- Rate of fire

Enemy Shooting

We essentially require the following from the enemy:

- Locate and approach the gamer
- Once in line of sight, aim at player
- Shoot at player
- Evade projectile ejected from player

We already have the pathfinding part. We now need to find the direction the gamer is at and whether he/she is in line of sight.

Q How do you obtain the angle between the enemy and the ship?

Once we now the direction, we can use `Ray` to check for line of sight. `Ray` is a straight line with some direction originating from some position that can be used to check for intersections. Checks return

- `null` if no collisions occurred
- The distance to the item collided with

Enemy Shooting

```
 1 public bool IsObjectVisible(BoundingSphere objectSphere, Vector3
       objectPosition)
 2 {
 3   Vector3 direction = objectPosition - translation.Translation;
 4   direction.Normalize();
 5
 6   Ray ray = new Ray(translation.Translation, direction);
 7
 8   float? distShip = ray.Intersects(objectSphere);
 9   bool lineOfSight = false;
10
11   if (distShip != null)
12   {
13     lineOfSight = true;
14
15     for (int i = 0; i < level.walls.Count; i++)
16     {
17       float? distWall = ray.Intersects(level.walls[i].GetBoundingBox()
             );
18
19       if (distWall < distShip)
20       {
21         lineOfSight = false;
22         break;
23       }
24     }
25   }
26
27   return lineOfSight;
28 }
```

Collision Detection

The collision detection methods we used so far are very inefficient: we always check all objects to see if they collided with any other object. Instead, we should use structures like binary space partition trees to narrow down the number of items that need to be compared.

Likewise, we use spheres or bounding boxes for the collision detection. In many cases (e.g., walls), this is fine. If, however, the structure of the object is more complex, we might need a more fine-grained approach, particularly when it comes to getting hit by projectiles.

If a collision has occurred (using the merged sphere), then check all mesh-spheres and accept hit only if one of these spheres has been intersected.

Outline

Summary

In this lecture we covered advanced techniques for content import, advanced AI techniques and improved our game by adding interactivity in the form of shooting.

We covered

- Content Importer / processor
- Content Reader / writer
- Behaviour trees

- Hierarchical collision detection
- Regional collision detection
- Shooting

In the lab you will implement some more advanced AI behaviour. The game will be a simple capture-the-flag scenario. There will be two goal-oriented behaviours:

- Capture the flag
- Kill the enemy