Topic 3
Reactive animat, search, obstacle avoidance & problem solving

• Game’s “Physical” Environment
• Representing Space in the Game World
• Navigation in the Game World
• Obstacle Avoidance
• A Reactive Control System
• Rule-based decision-making

Reading: Champandard Chapters 5, 6, 8
The FEAR documentation
Website links on graph search
Game’s “Physical” Environment

• Distinguish between two aspects of a game environment:

  **Structure** - topography of the environment as it constrains movement (physics and layout of walls, paths, obstacles etc.
  **Detail** – graphical appearance of the game world and placement of objects which don’t impede movement

• What about players, NPCs and monsters? Really need to consider moving things as a third category, especially when interactions go beyond simply destroying everything you see

• Humans see the world mostly visually through detail, while AI sees the world as simplified data structures and must interpret these as well as it can

• We can try to make an AI interpret the graphical world directly, as if it was seeing through eyes, but such *machine vision* has proven to be very difficult to program and expensive to compute (at least at a human level of skill)

• It is an important concept of nouvelle game AI that an animat should only have local, not global, knowledge of the game (like a human)

• Having complete, perfect knowledge of the world is not a good model for AI or games
Machine Vision

- Getting a machine to see is a traditional sub-discipline of AI
- A typical system might involve a camera returning a digitised image, interpretive software and some kind of output arrangement
- Eg handwritten letter recogniser

- More sophisticated output information requires more complex processing. Eg. scene analysis for robot navigation
Representing Space in the Game World

• It’s important how space is represented

• 2D vs 3D – how the location of objects is encoded in the structure, not how the detail makes the world appear

• Discrete vs continuous – meaning whether objects are placed in a grid or matrix with a finite number of locations or not (eg chess vs marbles)

• Representation of time – also discrete (turn-taking) or continuous (stream of consciousness)

• Conversions – discrete vs continuous is a relative matter, since a fine enough unit size (grid or time-steps) may be considered continuous in practice

• In fact all representations in computers must ultimately be discrete, but may approximate continuous or not
At present, game engines provide a *locomotion layer* which abstracts basic movement actions away from the strategic control of direction.

Physics simulation is required to handle gravity, throwing, fire, water etc.

In future, more low level motion might be handled by the AI.

**Control**
- (forward/backward, turns)
- signals from user or parameters via API from the decision-making AI

**Collision detection**
- Physics in the game signals a collision halting (forward) motion

**Integration**
- signals from environment alter behaviour as appropriate (eg falls)

**Simulation loop**
- (walking, running)
- physics handles displacement
- animation handles limb cycling

*Image: Chris Bayliss*
Representing Space in the Game World

- In classical AI navigation experiment, travel paths in the world model might be done at design-time as a graph - a complete, compact representation of this space
- Finding an optimal path from a current location to a target was then a matter of search on the graph
- This is well-studied and good search algorithms such as A* are available
Search - Basics

- *Uninformed* search algorithms simply follow a pattern to examine all nodes until one containing the goal is found.
- Depth-first search - start at a root and explore as far as possible along each branch before backtracking until goal is found.
- Breadth-first search - start at a root and explore all neighbouring nodes, then for each neighbour, explore their unexplored neighbours, and so on until goal is found.

On this graph, starting at A, choosing left nodes before right and remembering previously visited nodes:

- DFS visits nodes in this order: A, B, D, F, E, C, G
- BFS visits nodes in this order: A, B, C, E, D, F, G
Search – Using Domain Information

- *Informed* search algorithms use some heuristic to choose intelligently which node to search next
- Best-first search – modifies breadth-first method to order all current paths by a heuristic which estimates how close the end of the path is to a goal. Paths that are closest to a goal are extended first.
- *A* search - is a best-first method that associates a cost and an estimate with the path leading to each node
  - Cost is zero for the first node; for all others, cost is the cumulative sum associated with all its ancestor nodes plus the cost of the operation which reached the node (e.g. linear distance)
  - Estimate measures how far a given node is thought to be from a goal state (e.g. intensity on a sensory gradient)
  - Cost and estimate are summed to score each path for eligibility for exploration
- For more detail, view the ‘Graph Search Methods’ link on the website
Navigation in the Game World

• There are problems with this kind of model however:
  - Depends on global information at design-time, so
  - Question of realism arises – not comparable with the limited viewpoint of real biological creatures
  - A lot of information may overwhelm decision-making processes
  - Information does update dynamically via sensors, so cannot track changes (e.g., moving creatures in the world)

• Nouvelle game AI animats are (virtually) embodied, which implies that
  - they have a (simulated) limited perceptual system which updates the AI continuously
  - they need more plausible navigation algorithms which can work on limited information and in real time

• For now we are interested in reactive solutions
• Fortunately, such solutions have been studied for the design of robots
Modelling an Animat’s State in Space

- For many (but not all) AI models, need a description of the animat’s position and orientation in space, as well as how it will move.

**Absolute Coordinate System**
- (0,0) World Origin
- Animat (3,3)
- Object (6,1)

**Relative Coordinate System**
- (0,0) Egocentric Origin
- Animat (3,2)
- Object (3,2)

- Continuous moves and turns
  - any angle
  - any distance

- Discrete moves and turns
  - 90 deg angles
  - unit distance
Animat brains in FEAR

- In FEAR architectures, modules and interfaces are defined *declaratively* and at a very high level in XML
- This is supposed to make building and debugging easy
- Yet for a standard animat in the Quake 2 demonstrator, the most interesting specification is that of the animat’s brain, which is a C++ function defined in the file Brain.cpp called Think and that is *procedural*
- The following very basic navigating brain depends on calls to the physics, vision and motion modules

```cpp
void Think()
{
    bool col = physics->Collision(); // check for touching contact

    float front_obstacle = vision->Tracewalk (0.0f); // locate any problems ahead

    // decide what actions to call
    motion->Move(Forward);
    if (col || front_obstacle < 2.0f)
        motion->Turn(orientation); // where orientation is some angle
}
```
Sensing and Acting in the World

• Physics module
  // returns true if animat has bumped something, false otherwise
  physics->Collision(); // Polled
  physics->OnCollision(); // Asynchronous callback detecting a collision

• Vision Module
  // returns the distance to nearest object at bearing angle
  vision->TraceWalk(angle, steps); // simulating walking on that bearing
  vision->TraceLine(angle); // as if throwing projectile on that bearing

• Motion module
  // continuous - angle is in deg. from current orientation
  motion->Turn(angle);

  // continuous - move in direction, a literal eg. Motion::Forward
  motion->Move(direction);
Obstacle Avoidance – Basic functionality

• Finding one’s way around obstacles is fundamental to navigation
• Well suited to implementation by a reactive control system
• Begins from general requirements:

1. When no obstacle is sensed, move ahead normally (straight or wandering randomly)
2. If wall detected on one side, veer away slowly
3. If obstacle detected ahead, turn to one side to avoid it
4. When stuck in a corner, a more radical turn is needed
A Reactive Control System

• Essentially, a reactive system is simply a set of standard behaviours – sensori-motor functions, which engage in a particular situation

• These should operate in parallel, competing for control of the animat’s body => need for arbitration in case of a tie

• In FEAR we have Synchronous Calls. Based on the client-server model. Requests by a client are made and do not return until the server has computed a result. Usually based on simple function calls. Commonly used for the delegation of tasks

• An if-else-if-else structure could organise calls to sensori-motor functions

• Or maybe use Asynchronous Events. Based on something called the observer design pattern: have a set of behaviours ready to go and a set of events that will trigger them (event-based processing)

• These can interrupt another routine when something comes up unexpectedly
Rule-Based Decision Making

- Design progresses from a simple reactive code to Rule Based Systems
- In a *forward-chaining* RBS, sensory input triggers some rules, which in turn trigger other rules, which eventually trigger rules generating output
- This is usually most appropriate for animats, which must work from perception to action (cf. backward chaining)
- Arbitration – how are competing motor outputs combined to control the limbs?
  - **Independent sum** – different outputs connected to different effectors so that they cannot clash
  - **Combination** – some formula (eg weighted sum) combines two or more outputs to blend them into output commands
  - **Suppression** – certain components get priority over others, and can replace their output with their own (eg. subsumption architecture)
  - **Sequential** – output of different behaviours getting control at alternate times
Subsumption architecture works on real robots

- Motor load
- Bump switches
- IR proximity
- Photosensors

Sensory triggers

- Random Act
- Escape
- Avoid
- Follow
- Cruise

Motor command