Task 21 - Spike: Message Systems

CORE SPIKE

Context: Objects in games often need to communicate with a wide range of other objects. In order avoid the maintenance nightmare that is coupling every game object to every other game, a more general-purpose messaging system can used to help game objects communicate. Two general message system architectures are “blackboard” systems or “dispatcher” systems.

A blackboard approach is where game objects “post” (leave) and also “check” (look) for messages with some centrally-available message location (the blackboard). A dispatcher system is more like a “post office”, which can encapsulate and manage the collection and distribution of messages to known game entities.

Knowledge/Skill Gap: Developer needs to be able to send messages between the components in the Zorkish game context, using a flexible and robust messaging system.

Goals/Deliverables:
[CODE] + [SPIKE REPORT]

You need to develop a messaging system that would be suitable to facilitate communication between game objects (entities) in your Zorkish implementation. Develop and test a system that is suitable for this role.

Note: You do NOT need to include this into a full “Zorkish” implementation – focus (spike) on creating a suitable design and a simple (Zorkish-free) demonstration of it working, with just what is needed to show it.

You need to produce:

1. Design details for the message system (overall architecture, i.e. blackboard or dispatcher), expressed as class/module/sequence diagrams (or equivalent). Include a clear description of your message details. (See notes below). Include the design details in your spike report.

2. A working demonstration that shows how your message system can
    a. Send/Leave a simple message from one game entity to another (A to B) to change a state.
    b. Send/Leave a message with extra data from one game entity to another (A to B with data)
    c. Send/Leave a message response for the (A to B, B to A response)

Note: Do NOT implement broadcasting (A to Many), filtering, or schedule/delay behaviour in this spike. If you wish to do this, look at the next spike extension that is for these options.

Recommendations:
You’ll want to put some thought into this before you start coding - design an **architecture** to handle messages, and a **specification** for your message details.

You will need to consider:
- How messages are sent, received and acted upon
- How messages are addressed
- What content is included in a message
- How objects “register” (connected to post office?) to receive messages
- Whether a message contains information about who sent it (is it needed?)

For implementation, keep it simple. Simple types and strings are fine for message “details”.

On the next page are a number of game-context message examples to help with the concepts involved.

Message Examples:

Consider these examples, written as simple sentences, to help describe the message types involved. The data in curly braces is just a simple format example - you don't need to follow it at all or use the same key: value pairs!

Remember that in these examples the messages sent and shared in an open and clear (truthful) way. Although from a human point of view it they might seem strange (announcing that you are about to attack the enemy and how every time!), they make sense for game objects. Think of it more like a script of what will happen in a play.

A to B to change state (simple message, no extra data)

- The player (A) wants to open a box (B) that has a state of “closed” to “open”. (No extra data)
  { from: A, to: B, message: “open” }

- When a light switch (A) is changed by the player to “on” (internal state), the light sends a message to the ceiling light globe (B) to change its state to “glow”. The light globe (B) gets the message, but because it is blown, it doesn’t change its appearance.
  { from: player, to: A (light switch), message: “turn on” }
  { from: A (light switch), to: B (light globe), message: “glow!” } // globe ignore it

- A player uses a key (A) is on a locked door (B), so the key sends a message to the door to “unlock”.
  { from: player, to: A, message: “unlock took B” } // door identified in the message string? Ok …
  { from: A, to: B, message: “unlock” } // should this work? Key match? Hmm…

A to B with data. (Simple message as “kind”/type, but with - say - an extra data part as … string or pointer)

- A character (A) wants to unlock some luggage (B). It has a combination lock so it needs the combination as well (as data) to work. So, A sends a message to B to “unlock” with the extra data of “12345”. As this is the correct combination and the lock opens! It also opens planetary air shields.
  { from: A, to: B, kind: “unlock”, data: “1,2,3,4,5” }

- Elf character (A) is doing a mid-height sword attack to an orc (B). So
  { from: A, to: B, kind: “attack”, data: ”sword at mid-height” }

A to B, then B to A with response

- In a world-construction game, a player (A) is at a making table (B) and wants to make a sword. It requires a collection of resources, 1 stick of wood and 2 iron. The player A sends a message to B with the resources as pointers in a list (data). When the construction is finished, the making table B sends a message to A with the sword as a pointer (data)
  { from: A, to: B, message: ”make”, data: [0x0001, 0x0002, 0x0003] }
  … // later when it’s done …
  { from: B, to: A, message: ”make result”, data: [0x0004] } // fake addresses are given as examples.

- In a soccer game, the AI player A (also known by int id=3) has the ball and wants to kick the ball to the best positioned team mate. It asks the map (B, known by id=12) “who is best to kick to?” (using an enum). The map figures this out when it can (it’s rather busy) then sends a message back to player (id=3) with a list of potential support players, each with int id numbers.
  { from: 3, to: 12, message: BEST_KICK_LIST, data: [] }
  { from: 12, to: 3, message: BEST_KICK_LIST_RESULT, data: [4,1,3,9] }

A flexible system is nice – try not to create a very ridged system if you can avoid it, but also just get it working! ☺