Block 2: Game Data Structures and Patterns

Description:
This block of spikes is designed to familiarise you with the data structures and patterns that underlie modern game engines and other real-time systems.

Spikes:

*Spike 5: Game State Management
*Spike 6: Game Data Structures
*Spike 7: Graphs
*Spike 8: Command Pattern
*Spike 9: Composite Pattern
*Spike 10: Component Pattern
*Spike 11: Messaging
*Spike 12: Announcements and Blackboards
*Spike 13: Unit Testing

Spike 5: Game State Management

CORE SPIKE

Context: Game state management is a common feature of games.

Knowledge/Skill Gap: The developer is not aware of implementation methods for flexible game state (“stages” of a game) management.

Goals/Deliverables:
[PAPER DESIGN] + [CODE] + [SPIKE REPORT]
You need to create the “Zorkish Adventure” game, as described in the specification document available on the unit website.
You will need to deliver the following items:

1. A simple paper-based plan for your code design. (REQUIRED)
2. Create a simple console program that implements the “Zorkish: Phase I” game (although it has no gameplay yet) using a flexible (extensible) game state management method of some kind. The OO State Pattern is the strong suggestion. The implementation must demonstrate the following game stages (states):
    a. “Main Menu” (which allows the user to select other stages…)
    b. “About” (remember to include your own details here)
    c. “Help” (summary of commands – simple hard-coded text is fine)
    d. “Select Adventure” (use a hard-coded list and the title of your test game)
    e. “Gameplay” (test game which only accepts “quit” and “hiscore” commands)
    f. “New High Score” (allows user to enter their name, but doesn’t work yet)
    g. “View Hall Of Fame” (view list of name/score. Simple hard-coded text is fine.)

Recommendations:

- Read the complete Zorkish game specification document.
- If not familiar (or you need a reminder) research/read about the state pattern used to represent each “stage” of the game.
- On paper create a design for your implementation. A strong suggestion is a UML class diagram representing a state pattern you would need specific to the Zorkish game.
- Use an “agile” (test often) approach as you implement your design. For example, implement a single state and test, then two states and test changing between the two.
- Leave complex issues or issues that might distract from this spike until last.
- Stay focused on the main points of this spike – state management! Do NOT implement a complex gameplay parser, the “Hall of Fame” file IO, scoring features etc. Read later spikes to see why! Focus on the minimum to get this spike done!

Spike 6: Basic Game Data Structures

CORE SPIKE

Context:
Game developers will often encounter a variety of different types of data and access/usage scenarios, even in a single project. It is essential, therefore, that developers be aware of the different data types available to them, their advantages and disadvantages, and suitability for different purposes.

Knowledge/Skill Gap:
The developer is not familiar with common data types, their various strengths and weaknesses, and their applicability in common game scenarios.

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

1. Research and evaluate four different data structures that could be used to create the player inventory for the Zorkish game. At a minimum you must show your awareness of advantages and disadvantages for this application. Document your evaluation criteria and results in a short report.
2. Using your decision (as documented in your short report), create a working inventory system demonstration program. Your work must demonstrate (bug free) inventory access, addition and removal.

Note: The short report is separate from the spike report. Ask your tutor about it you aren't sure what to put in each. Your work won't be accepted unless you have all deliverables.

Recommendations:

- A nice overview of different data structures is available in the C++ STL is here: http://en.cppreference.com/w/cpp/container. You may use other libraries, but the STL is recommended.
- Keep the short report SHORT - very focused and concise. No padding or fluff!
- If you do not yet have a specific Player Character class/object for Zorkish, now might be a great time to create one.
- In this case the demonstration program does not technically need to be part of your other Zorkish code features that you've created so far.
- Include a map

Spike 7: Graphs

CORE SPIKE

Context:
Selecting appropriate data structures and representations for game information are critical performance and development issues for game programmers. Graphs are a general data structure with many applications. Developers should be able to take advantage of graphs data structures in their game implementations.

Knowledge/Skill Gap:
The developer is not familiar with the use of graphs data structures for representation of game world composed of locations and connections.

Goals/Deliverables:
[FORMAT DESIGN] + [CODE] + [SPIKE REPORT]
Extend the Zorkish program you created in Spike 19 to include the loading of an Adventure text file and the representation of location and connection information as a graph data structure. Refer to the game specification document on the unit website for details.
Your implementation at this stage does not (yet) need a complex command processor, but will need to support "go" commands in order to show the graph structure.
Create a Zorkish game that demonstrates the following:

1. Specify on paper (REQUIRED) a simple text-file format that represents a Zorkish game "Adventure" details. Specifically, it will need to include (at this stage) world locations, location details (name, description etc.) and connections to other locations. You can include other details in your design, but we only need locations and connections for this spike.
2. Be able to load an "Adventure" file from the "Select Adventure" game state.
3. Represent the locations and connections as a graph in your program. Implement a basic set of "go" command with directions that map onto your graph. (You must implement more than North, East, etc. This is NOT a grid world – any direction is possible!)

Recommendations:

- Don't worry about implementing objects or other entities in the world for this spike. Just focus on locations and connections and some basic move commands
- Don't worry about a command processing system (that's another spike). This spike is about graph representation of game locations.
- Read the game specifications again. Clearly identify the minimum you need for this spike (and the focus on a graph to represent the game locations).
- Make a list of the type of details that need to be stored at each location.
- Convert your list into a node design (class?) and a graph design. Identify what graph based functions you will need to move a player around the world.
- To move the player how will you alter the graph? Does the graph contain a reference to the player? Does the Player contain a reference to the graph? What potential advantages/disadvantages are offered by each approach?
- Create a very simple text file. Write code to load the file (maybe just print the details back to screen to start with) and then create a graph using the loaded details.
- Implement the basic (minimum) "go" commands you need to test your graph data
- Test.

Spike 8: Command Pattern

CORE

Context:
A text-based adventure game can create an immersive experience where a player feels like the game "understands" them. One part of this is a robust way to process user input (text commands) and turn them into game world actions. The skills used to create good parsers and management of commands are also very useful in many other games and software projects.

Knowledge/Skill Gap: The developer needs to know how to create, for a text-based adventure game, a text command parser that is robust and accepts typical human variations, and an effective way to design, manage and extend commands.

Goals/Deliverables:
[CODE] + [SPIKE REPORT]
Building on the work of earlier Zorkish Spikes and create a robust text command parser and command manager for the game. (Refer to the game specification document on the subject website for details). The command processor should accept alias commands and optional or variable number of words in commands.
Create part of the Zorkish: Phase 2 game that demonstrates the following:

1. The loading of adventure files (text) that includes (partial) specification of game entities
2. A robust command processor (supporting aliases and optional words)

Recommendations:

- Use an OO command pattern.
- Read the game specifications again.
- Research the "command pattern". (You'll probably want Command objects and a CommandManager that can "run()" each command object when needed.)
- Research "dictionaries" – collections that can access contents using string keys. (STL)
- Put designs and plans on paper. Think as much as possible before you code! (If you do this, be sure to include your paper design with your outcome report.)
- Create a new adventure file that contains a minimal game world description and some game entities.
- Update the adventure loading code so that your game world (graph) supports entities.
- Implement a simple command (but using the command pattern/manager) and test.
- Implement other commands – and test… extend… until done.

Spike 9: Composite Pattern

CORE SPIKE

Context: A text-based adventure game can create an immersive experience where entities of the game can be compositions of other entities. To do this, player commands need to act on “entities” of the game, some of which are composed of other entities.

Knowledge/Skill Gap: The developer needs to know how to create and modify, for a text based adventure game, game entities that are composed of other game entities.

Goals/Deliverables:
[CODE] + [SPIKE REPORT]
Building on the work of earlier Zorkish Spikes, extend the game world to support game entities composed of other game entities. Update the command parser and command manager for the game to support actions that modify the composition and location of game entities.
Create part of the Zorkish game that demonstrates the following:

1. Adventure (world) files that include the specification of game entities, their properties, and any nested entities (composition) they may contain.
2. Players are able to observe and modify entities (what they contain, and their location) ie. “look in”, “take _ [from] _”, “put _ in _”, “open _ [with] _”

Recommendations:

- A dictionary collection making reference to objects using strings as keys, and an OO command pattern. The game location graph can be extended to support entities that are collections of entities – this is the essence of the OO composite pattern!
- Read the game specifications again.
- Research dictionaries – collections that can access contents using string keys. (STL)
- Put designs and plans on paper. Think as much as possible before you code! (If you do this, be sure to include your paper design with your outcome report.)
- Create a new adventure file that contains a minimal game world description and some entities that also contain other entities that you can use later for testing.
- Update the adventure loading code so that your game world (graph) supports the entities and the composite pattern
- Extend the player commands (the command pattern/manager) to enable modification of game entities composition and test
- Implement other commands – and test… extend… until done.
- Test. Check for memory leaks… (Seriously!)

Spike 10: Component Pattern

CORE SPIKE

Context:
Game programming often makes heavy use of inheritance, which can be appropriate for their roles as simulations (however stylised) of the real world. In many instances though, this can lead to unnecessarily deep class ‘trees’ and many abstract objects intended to represent specific properties an object deeper in the tree may have. The component pattern de - emphasises inheritance as the source of object attributes by creating objects out of components, each one contributing some attribute or function to the complete object.

Knowledge/Skill Gap: The developer needs to know how to create and modify, for a text - based adventure game, game entities that are the sum of their parts, receiving attributes from components rather than inheritance.

Goals/Deliverables:
[CODE] + [SPIKE REPORT]
Building on the work of earlier Zorkish Spikes, extend the game world to support game entities composed of components that contribute properties and actions.
Create part of a game that demonstrates the following:

1. Game objects that receive attributes (damage, health, flammability, etc.) from component objects rather than inheritance.
2. Game objects that receive actions (can be picked up, can be attacked, etc.) from component objects rather than inheritance

Recommendations:

- This is a good place to start learning about the component pattern: http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/
- The visitor pattern is has some similarities to the component pattern, and tutorials on implementing the visitor pattern can provide inspiration for implementations of the component pattern.

Spike 11: Messaging

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 game object it could feasibly need to communicate with, a messaging system is used to help game objects communicate.

Knowledge/Skill Gap: The developer needs to implement a messaging system to allow the components in the Zorkish game to communicate with each other in an expandable fashion.

Goals/Deliverables:
[SPECIFICATION] + [CODE] + [SPIKE REPORT]
You need to develop a messaging system to facilitate communication between game objects in your Zorkish series of Spikes.
You will be using this messaging system in all future Zorkish spikes, so you'll want to put some thought into this before you start coding - develop a specification for your messaging system that described how it will operate. You will need to consider:

- How messages are sent
- How messages are received and acted upon.
- How messages are addressed
- What content is included in a message
- How objects register to receive messages
- Whether a message contains information about who sent it

Recommendations

- You'll need to give some thought to what form your messages will take. Some options are:
    - Create a Message class that contains all the message information
    - Create a Messaging class tree with different Message subclasses containing different information
        - The subclasses in such a tree can be created based on any of: message types, message components, recipient types, sender types, or combinations of these - if you go down this path you'll need to put thought into how each Message subtype will be used.
    - Have a message be a pointer to a chunk of data
        - dangerous for a great many reasons, but has some advantages - if you choose this method, you should demonstrate awareness of the advantages and disadvantages in your specification and spike report
    - A string
        - Sometime simple is best - a string can often contain all the necessary information for a simple message, and is guaranteed to be human readable when things break
    - Other message data formats
        - There are a lot of other message data formats, each with their own advantages and disadvantages, such as XML, JSON, CSV, binary data, etc.

Spike 12: Announcements & Blackboards

CORE SPIKE

Context:
A messaging system allows for immediate communication with other objects in a robust and expandable fashion, but a one-to-one message system doesn't suit all game scenarios. Some messages need to be sent to a number of subscribed destinations simultaneously (announcements). Other messages can only be handled by their intended recipient at some point in the future, and need to be held until they're accessed (blackboard).

Knowledge/Skill Gap: The developer needs to be able to send messages that can be addressed to a number of subscribed destinations and messages that won't be received immediately

Goals/Deliverables:
[SPECIFICATION] + [CODE] + [SPIKE REPORT]
You need to expand your messaging system from Spike 16 to include

1. Announcements, a single message sent to a number of subscribers
2. Blackboards, messages that are accessed by the recipient instead of delivered

Recommendations

- You'll need to give some thought to how your addressing scheme will handle multiple recipients, and to what will differentiate an announcement-style message from a normal one:
    - will they be different objects?
    - will they use a different system?
    - will they be transparently handled by your messaging system without the sender and receiver knowing the difference?
- You'll also need to consider how your Blackboard will handle messages:
    - will it make repeated attempts to send a message?
    - will objects repeatedly check the blackboard for messages?
    - what will your Blackboard do when a destination is unavailable or invalid?
    - what will your Blackboard do after a message has been received? will it keep or discard it? how long will it be kept?
- It's OK for the answer to the above questions to be "not implemented" or "the system breaks" - you just have to demonstrate you've given some thought to these and similar problems