Core Spikes

Introduction

This document is an overview of the Spikes available in this subject.

What is a Spike?

A Spike is a small project designed to close, through research & implementation, a knowledge or skill gap. The emphasis here is on the small – you should only do what you need to close (and demonstrate that you have closed) the skill gap.

Each Spike consists of a Spike Report and a deliverable. The deliverable is usually a code piece, but can be a research report (or, occasionally, both). If the deliverable is a research report, the research report is a separate document to the Spike Report – you must submit both reports.

The Spike Report is a report that details the work you have done to complete the spike. A template for this report is available on blackboard.

All of the spikes below are compulsory. Completing them is required to get a Pass. Doing these spikes to a high degree of quality and doing additional spikes from the Non-Cores Spike list (see Blackboard) is required to get a Credit. Distinction and High Distinction require a Credit level of work plus a project and a research project, respectively.

These spikes broken up into Streams to indicate Spikes that have common themes. All Streams and Spikes in this document must be completed to get a passing grade,

Stream 1: How to do C++ work & Game Loops

Description:
This stream of spikes is designed to familiarise you with the programing language and IDE you will be using for the rest of the semester.

Spikes:

*Spike 1: Simple Game Loop
*Spike 2: IDE Experience & Comparison
*Spike 3: Debugger Use
*Spike 4: Non-Blocking Game Loops

Spike 1: Simple Game Loop
CORE SPIKE

Context: Games are commonly driven by some form of game loop.

Knowledge/Skill Gap: The developer is not familiar with basic game loops.

Goals/Deliverables:
You need to create the "GridWorld" game, as described in a simple specification document available **below**. The game will demonstrate the use of a simple game loop, separation of update/render code, and game data (both for the world map and players current location).

You will need to deliver the following items:
1. A simple paper-based plan for your code design. (Yes, a simple functional design is fine – just as long as you can demonstrate that you planned first before coding!)
2. Create a simple console program that implements the "GridWorld" game using a simple game loop. The game must demonstrate the separation of:
    a. processing of input (text commands from the player),
    b. updating of a game model (where the player is and their options),
    c. display (output current location and options) of the game to the user.
3. Spike Outcome Report.
   Note: The Spike Outcome Report is always a required "deliverable". It will not be repeated in future spike requirements as it is assumed.

Recommendations:
- Read the "GridWorld" game details and on paper (as required in the deliverables) do a quick sketch/design of how you will organise your code. (No formal standard specified… just use something that would work to help you explain your design to another programmer)
- Look at the Spike Outcome Report template – note what you need to record for later.
- Use an IDE for C/C++ development that you already know. (We'll look at some others later so don't get distracted from the main point of this spike!)
- Begin small, test often. Use simple printouts to check values (or the debugger if you already know that and are comfortable). Don't try fancy debugging yet if you don't know it – that's a later spike if it's something you need to work out. Stay on target!

Tips:
- Consider a DEBUG macro condition if extra code is needed for testing purposes.
- You do not need to show the "map" or the current player location – but it is useful.
- If you find some useful resources that helped you to get your code working, then remember to note them down and include them in your spike outcome report. Google, books, blogs, classmates, etc. all go in the spike report.
- Have a plan for when you present your outcomes to the tutor. (They will probably say something like "Right – show me"… and it's up to you to show-off what you've done.)
- Many of these tips also apply to spikes in general and are not exclusive to this spike.

DO NOT
- Make things any more complicated than you need – just get it working.
- Create complex data-structures. A simple 2D array is fine (and expected).

- Add all the fancy features you can think of! Save ideas for later and perhaps develop them as a portfolio item. (You should still document/note them down though.)
- Load the map data from a file. Simply hard code the map data.

Spike ILOs

Design:	LOW	(1)
Implementation:	MEDIUM	(2)
Maintenance:	–	(0)
Performance:	–	(0)

Spike 2: IDE Experience & Comparison
CORE SPIKE

Context: There are many IDE tools available and developers need to be able to make informed decisions in order to be productive with the tools they select.

Knowledge/Skill Gap: The developer is not familiar with different IDE development tools.

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

1. Using the simple command line program from Spike 1 create a working project using three of the following IDEs:
    a. Microsoft Visual Studio 2010/2012/2013 (any version)
    b. Code::Blocks
    c. Eclipse+CDT
    d. XCode
    e. Netbeans (for C/C++ development)
2. Create a short report to document your findings. (Note: the short report is NOT the spike report - you should have two reports for this spike.) Include the following details:
    a. For each IDE, a point-form list of the steps required to create a new project in the IDE, add new files, compile and run a command line program.
    b. For each IDE, a point-form list of the steps required to create a break-point location in the program, and run the program in using the IDE's integrated debugging system so that program execution stops at the nominated point.
    c. For each IDE, note how to inspect the value of variables during debugging.
    d. A comparison matrix (table) to represent the qualities of each IDE (you must decide what criteria you think is important) and your rating (value).
    e. Clearly state which IDE you will use in future and support your decision.

Recommendations:

- Start with the IDE that you know first (which you probably used for Spike 1). If you don't know any, we suggest Visual Studio or whatever your friends can help you with.
- Assume you are writing steps to help a colleague to create a project in the IDE.
- As a review, show your steps with some other students. Make sure your notes make enough sense! Update any suggestions or omissions so that your notes are valuable.
- Be sure to note the version numbers of software you use in your spike outcome report.
- Move on to the next IDE and repeat the steps, documenting as you go. (Try to complete all the deliverables for each IDE before moving on to the next.)
- Remember that spikes should be about the doing the minimum required to close the gap in knowledge or skill (or technology). However if you do not write enough so that someone else would understand, you have not complete to spike work correctly.

Note:
If you wish compare a different IDE that is not listed you MUST get permission from your tutor first. If you do not, your outcome report will NOT be accepted.

Spike ILOs

Design:	–	(0)
Implementation:	LOW	(1)
Maintenance:	LOW	(1)
Performance:	–	(0)

Spike 3: Debugger Use

CORE SPIKE

Context: Effective use of the debugger is essential for isolating and repairing code errors within a non-trivial project of source code.

Knowledge/Skill Gap: The developer is not familiar with the use of debugging tools.

Goals/Deliverables: [FIXED CODE] + [SPIKE REPORT]
You need to demonstrate that you have done the following:

1. Downloaded, compiled and run the "spike3" program provided on blackboard in the IDE of your choice. The program contains a number of "bugs", including a deliberate memory leak. (Memory is allocated, but not de-allocated.) You must discover and fix errors. (You might not find them all!)
2. Use IDE debugging tools to identify memory leaks and other issues. You must document in your spike report the IDE steps you used to do identify bugs. (You are welcome to use screen shots to supplement the written steps you have used. See the planning notes for suggested steps.)
3. Save the fixed code and included source code comments to document what you changed.

Recommendations:

1. Open the program in your IDE of choice provided it supports debugging and ideally "inspection". The subsequent steps may vary depending on your IDE, but should be similar.
    a. Add a break point at a point in the source code at a point likely to be the cause of the leak. (If you're unsure, place one close to the start of the application and step through all of it. This does mean more stepping, but at least you are less likely to skip over the broken code!)
    b. Compile and run the program in debug mode (or similar in your IDE).
    c. When the program breaks, step through the program, examining the source code and variables for potential causes of the leak. Note them down.
    d. Repair any leaks found. Add comments about your victory!

Note:

- You might not understand all the code provided, but that's okay! You should still be able to step through the program execution to find some bugs, if not all.
- The goal of this spike is not really about fixing the bugs/leak(s), but rather to get you familiar with the process of using the debugger, and to demonstrate its usefulness. Your spike report should not dwell on the nature of the bugs (or your corrections), but rather show that you can effectively use the debugger to solve problems (the spike "gap").

Spike ILOs

Design:	–	(0)
Implementation:	LOW	(1)
Maintenance:	LOW	(1)
Performance:	–	(0)

Spike 4: Non-Blocking Game Loops

CORE SPIKE

Context: The non-blocking game loop is a more sophisticated implementation of the game loop concept. It is the most common form of game loop used by modern games.

Knowledge/Skill Gap: The developer is unfamiliar with the non-blocking game loop.

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

1. Create a console program that implements the “Gridworld” game using a non-blocking game loop. The loop must execute continuously, only processing input when it occurs, and only providing output when necessary. The Gridworld game should be implemented with a timer.

Recommendations:

- The input stream in C++ makes it easy to determine whether input has occurred and act on it if it has - you'll have to go beyond simply using the >> operator, though.

Stream 2: Simple Game Implementation

Description:
This stream of spikes is designed to give you a practical familiarity with a game engine or game framework of your choice. This stream is implementation focused, with each spike representing the implementation of a different core game engine feature.

Spikes:

*Spike 5: Game Framework Familiarisation
*Spike 6: Initialisation & Feature Detection
*Spike 7: Input Handling
*Spike 8: 2D Drawing
*Spike 9: Sound

Spike 5: Game Framework Familiarisation

CORE SPIKE

Context: The next series of spikes will use a framework of your choosing - this is your chance to get to know it.

Knowledge/Skill Gap: The developer needs to familiarise themselves with the advantages and disadvantages of their chosen framework.

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

In preparation for the use of your framework

1. investigate and present a short report that clearly documents the overall structure of your chosen framework,
2. list a number of previous/current applications that use your chosen framework
3. formally list (comparison table?) the advantages and disadvantages of your chosen framework against a list of criteria that you define OR provide a comparison against another framework for a number of criteria you define.

Recommendations:

- You are expected to include a reference list/bibliography
- You are expected to do more than a simple first look at Wikipedia or the website for your framework.
- After your initial web-search research, select a few good quality sources that you could easily reference
- Collect general information needed to fulfil the report requirements.
- When evaluating new technologies it is good practice to select and define clear criteria.
- Methodically apply your comparison criteria. Try to go beyond simple yes/no criteria!
- Give full credit to any sources that you use
- **Clear your chosen framework with your tutor before you start**

Spike 6: Initialisation & Feature Detection

CORE SPIKE

Context: Game frameworks provide an easy way for developers to avoid writing and testing lowlevel code for games, however libraries and resources need to be initialised and used the right way before the fun can begin.

Knowledge/Skill Gap: The developer needs to know how to create a program using their chosen framework, show a window of a specified size and to detect and initialise a range of subsystems ready for use.

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

Create a simple application that creates a drawable window of size 600 wide by 800 high, and will close in response to an exit message (provided by the framework when the user presses Close [or Alt+F4, or Ctrl+Q, etc.]). You should also initialise at minimum the sound and input sub-systems as a reference task for later work.

Recommendations:

- Find a basic tutorial on your framework, read it and then recreate it.
- You will need to first install your framework (including making sure that your IDE of choice knows where the lib and header files are, and that your running application has a copy of the required dll's so that it can load them at run time)

Spike 7: Input Handling

CORE SPIKE

Context:
Developers need to know how to use a library that supports input for game use, and configure the library code to provide both event-based keyboard input and key-state models in order to support different input interaction models needed for typical gameplay.

Knowledge/Skill Gap: The developer needs to know how to handle event-based keyboard input (key-type up/down and simpler key events) as well as a buffered state view of the entire keyboard key set.

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

Create a simple console application that uses your framework to capture keyboard events and display informative messages as text. Your application must demonstrate two modes of operation concurrently:

1. When a specific alpha key (a-z) is pressed "down" or is released "up",
2. A keyboard state model that keeps track of all the current key states for number keys (1,2,.. 9,0).

Recommendations:

- Do NOT use images or fonts for this spike – use simple console text output only!
- Read some tutorials on keyboard handling in your framework. Make sure you have examples for the type of things you need for this spike.
- Given the need for "polling" and presenting the keyboard state model, a simple slow gamestyle loop could be used that polls at a set time interval and displays the current key states. For example, something simple like [---X-X----] to indicate number keys "4" and "6" are currently being held down is fine.

Spike 8: 2D Drawing

CORE SPIKE

Context:
The facility to load 2D images from file and display them to screen is a critical part of many software applications, especially 2D and 3D games. Using a library to support this functionality requires the developer to understand a libraries API and 2D graphics terminology.

Knowledge/Skill Gap: The developer needs to know how to load and display multiple images, including sub-regions of one image onto another.

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

Create a graphical 2D application capable of display images. Your application must:

1. Display a single image as the background image for your application, which can be toggled "on" or "off" using the "0" (zero) key
2. Load one other image that contains three identifiable sub regions within it. <example> 3. Define three rectangles that specify the sub-region ("part") for each cells image.
   a. Display each cell ("tile") image to a unique random locations using a toggle "on" or "off" in response to the 1,2 and 3 number keys.

Recommendations:

- Find and read documentation and tutorials related to simple (not complex or extended) image loading and display in your framework. Note – keep this as simple as possible.
- Create two of your own simple images (but do not waste time on this) saved as simple format. (.bmp is enough - there's no need to launch into more complex formats)
- Make sure you are aware of the bit-depth of your images and the screen. Always "optimize" your images to the current screen bit-depth to avoid performance penalties.
- Strongly suggest that you display messages to the console that describe what is happening and help debug your program -- such as "loading image1.bmp", "tile 3 display ON at location (10, 40)" and so on.
- Do not over-engineer this; if you are implementing classes or using a number of libraries you have almost certainly gone too far! Just because you can does not mean you should…

Spike 9: Sound

CORE SPIKE

Context: Playing sounds on demand for a game, based on game events, and playing background music, are key components to creating entertaining and immersive game environments.

Knowledge/Skill Gap: The developer needs to know how to use your framework to load, play and control game sound and music.

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

Create a simple application that demonstrates the following features.

1. Keys 1, 2, and 3 will each play a unique sample sound as soon as each key is pressed even if that sound is already playing.
2. Play or pause (not stop) background music in response to key-down press “0” (zero) being used as a toggle.

Recommendations:

- Find and read tutorials for playing a sound when an event occurs, and for playing and pausing the playback of music. (Note – you need to PAUSE the music, not just stop and start it again.)
- Create or download some sounds and a music file suitable for your intended work. (Ensure you have the right licenses for any sound or music)
- Your keyboard input spike will give you suitable code starting point for response to key events. Keep it simple.
- You may need to add debug code to your work to ensure systems are initialising and loading as needed.
- You will need to research an appropriate format for your sound file.

Stream 3: Game Programming Patterns

Description:
This stream of spikes is designed to familiarise you with common Game Programming Patterns.

Spikes:

*Spike 10: Messaging
*Spike 11: Announcements & Blackboards
*Spike 12: Game State Management
*Spike 13: Composite Pattern
*Spike 14: Component Pattern
*Spike 15: Command Pattern

Spike 10: 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 11: 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

- Announcements, a single message sent to a number of subscribers
- 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

Spike 12: 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 13: 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 14: 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 15: 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.

Stream 4: Game Data Structures

Description:
This stream of spikes is designed to familiarise you with data structures that are useful for storing and manipulating game data.

Spikes:

*Spike 16: Basic Game Data Structures
*Spike 17: Graphs
*Spike 18: Hash Maps

Spike 16: 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 three or more 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.

Spike 17: 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 18: Hash maps

CORE SPIKE

Context:
Games often make use of datum that are intrinsically related. Key-Value pair data types allow developers to associated pairs of datum with each other, where one datum is used as a key to retrieve another (the value).

Knowledge/Skill Gap:
The developer is not familiar hash maps as the most common implementation of the key-value-pair data structure in C++.

Goals/Deliverables:
[CODE] + [SPIKE REPORT] + [SHORT REPORT]
Extend your report and code from Spike 19 to include the STL Hash Map.

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!

Stream 4: Game Engine Optimisation

Description:
This stream of spikes is designed to familiarise you with the process of optimising a game application and the tools and techniques that are useful for that purpose.

Spikes:

*Spike 19: Measuring Performance
*Spike 20: Optimising Simple Collision Detection

Spike 19: Measuring Performance

CORE SPIKE

Context:
Before any optimisations can be undertaken, the performance characteristics of the application must be understood. Good optimisation processes requires solid measurement techniques to collect data that can then be accurately compared and analysed, and used to inform code changes.

Knowledge/Skill Gap:
Developers need to be able to collect and analyse software performance data.

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

Use the code provided on the unit website for this spike. The code uses SDL, and creates a number of boxes with random size, position and velocity. The code is designed to run a number of different "tests", each using a standard game loop that updates (moves) each box, checks for collisions, and draws all the rectangles to the screen. Each test uses a different collision detection approach, and is timed.

Measure the performance of the various collision-checking methods provided. You can use the timing code provided or any other profiling tool you choose to collect meaningful data, and then create a report that presents a comparison of the different collision detection approaches. Your results must be collected without the overhead of rendering time, and for a sufficient duration to create stable results. Your report must include:

1. A description of the each collision approach (and the differences between them). 2.
The method you used to collect your data and your reasons for choosing it

3. The raw data you collect to support your results.
4. Analysis and discussion of the results.
5. A summary table of the results

Recommendations:
- Download, compile and run the code provided on the unit website. You will need to create a new project that has the appropriate references/paths set to your SDL libraries.
- Read through the code carefully and understand the structure. The code uses function pointers to make running different tests easy. You may need to read up on this first.
- Identify the different collision (crash_test) functions that are available in the code.
- There is an explicit delay in the game loop of the default code. You should remove this so that the code runs as fast as possible.
- When collecting collision data, you also want to remove the overhead of rendering. (There is a simple Boolean flag to do this.)

- Make sure you run the tests for a more than just a few seconds, and repeat the tests in order to get a good sample size for your results.
- Run the tests, collect the data, analysis and present in your report. A pretty chart is highly recommended. Don’t forget to include the summary table.
- The timing code provided is the simplest way to get results, but using a more sophisticated profiling tool is recommended, both to improve your results and to improve your familiarity with professional tools

Spike 20: Optimising Simple Collision Detection

CORE SPIKE

Context:
Once measurements have been completed, informed optimisations can be made.

Knowledge/Skill Gap:
Once measurements have been made, the developer can identify and execute improvements to code.

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

Using the results of your measurements from Spike 30, modify the provided code to use the most collision detection method found to be the most optimal. When this is done, add your own modifications to further improve performance. Using your performance measurement method from Spike 30, demonstrate that your modification result in performance improvements (however minor they might be) over the most optimal path found in Spike 30. Include the results of your performance testing and the rationale for your modifications in your Spike Report.

Recommendations:

- You do not have to achieve enormous performance improvements (although there certainly are substantial gains to be made), you just have to achieve a measureable improvement.
- If the improvements you make do not result in consistently better results, consider why this might be and note the reason down in your spike report