Task 09 - Spike Summary Report

Spike: Task_09
Title: Game Data Management
Author: Thomas Horsley, 103071494

Goals & Deliverables

Aim: Develop an understanding around various data collection methods in relation to games programming. Specifically when to use them, how to implement them and best practices surrounding the data structures in question. From here, choose the most applicable data structure and implement an inventory system for Zorkish.

Deliverables:

- Functioning Code
- Spike Report
- Short Report on Data Structures
- Git Commit History

Technology, Tools and Resources

Tech and Tools

The project was scripted in C++ 20 using Visual Studio Community 2022.

UML's and charts are made with *www.Lucidchart.com*

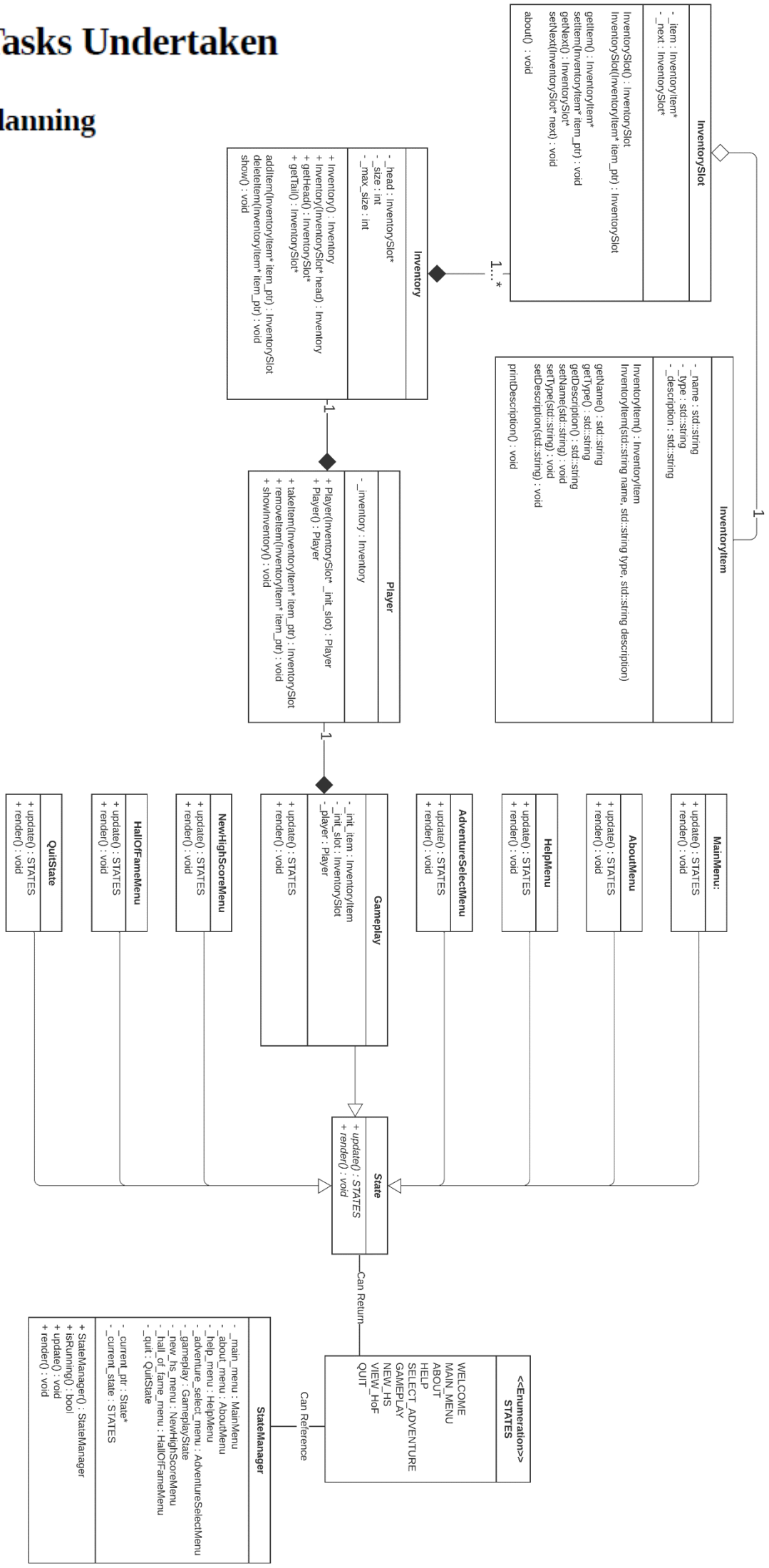
Source control is handled using Git.

Resources

- Echo360 Lecture "*Data Structures*"
- Vectors vs Arrays, What's the performance gap? https://stackoverflow.com/questions/381621/using-arrays-or-stdvectors-in-c-whats-the-performance-gap
- Comparison between Array, Linked Lists and Vectors: https://vlsiuniverse.blogspot.com/2016/04/array-linked-list-vector.html#:~:text=Vector %3A Vector is a data,size let us say N.
- Vector Performance Management: https://www.acodersjourney.com/6-tips-supercharge-cpp-11-vector-performance/

Tasks Undertaken

Planning



Task 09 - Spike Summary Report

Class Descriptions and Notes

Before discussion regarding the planning phase of the Zork(ish) inventory system it's important to note a change regarding the UML and final solution. The only notable mention in this regard is that addItem (and implicitly it's wrapper functions) no longer return an InventorySlot but rather construct a new InventorySlot on the Heap before assigning a reference to said InventorySlot to the lists tail. This lovely UML mistake cost 2 days of headache as the InventorySlot was falling out of scope when the function returned resulting the slot being lost and the data unreadable, a true emotional rollercoaster.

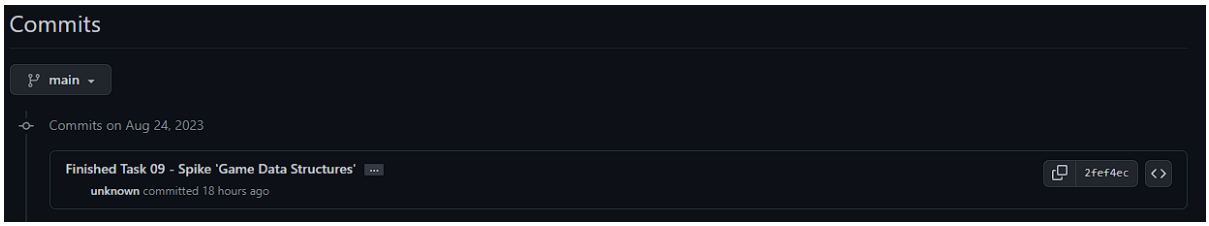
InventoryItems are simple placeholder classes for the moment containing three string variables for name, type and description, an InventoryItem has the ability to write these values. The Inventory itself (discussed below) is a collection of InventorySlot items each containing a reference to an item, the next slot in the list, a set of properties for the Inventory (manager class) to assign and reference and the ability to print it's own values.

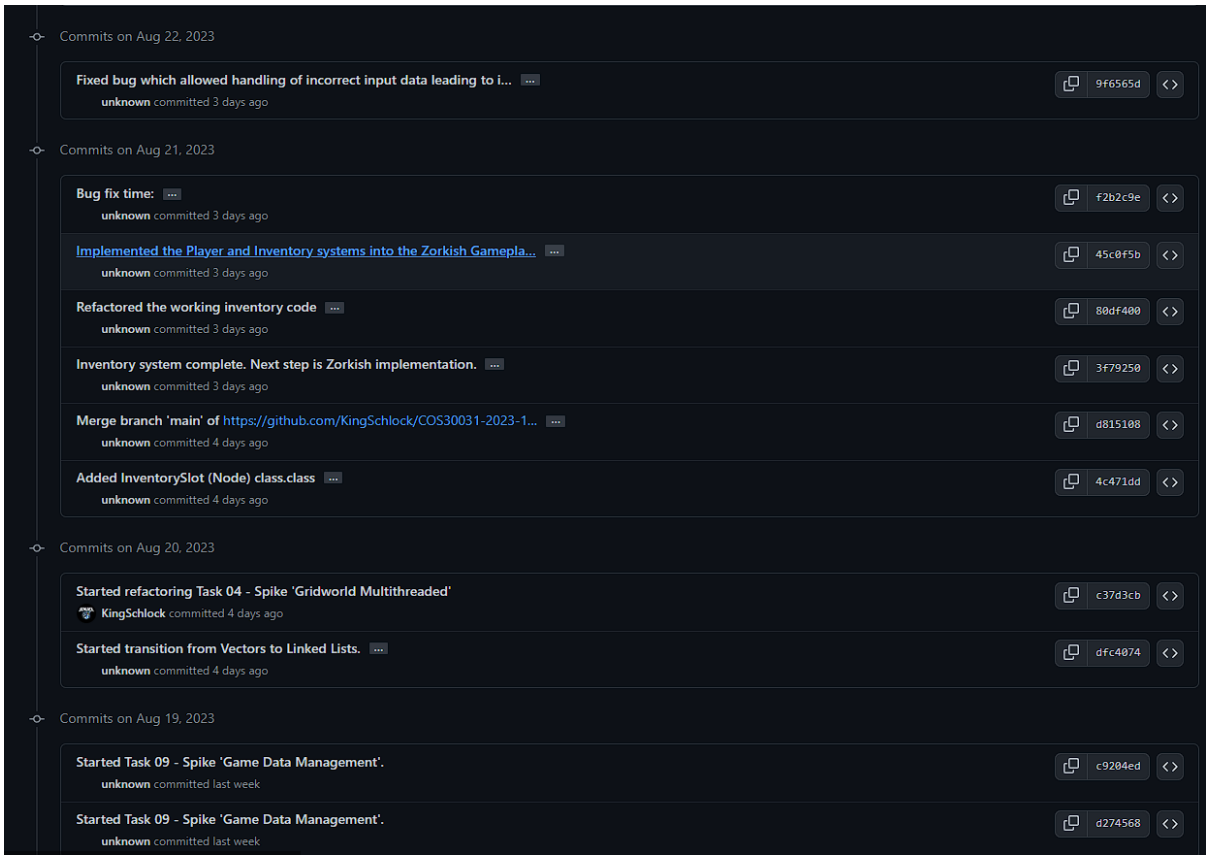
The Inventory class is the Node manager for the Forward Linked List data structure which has been implemented to manage the inventory. The Inventory holds the functionality to add new InventorySlot items to the list, traverse the list and find Nodes containing InventoryItems and remove those items from the list with query methods. All of this functionality is encapsulated within a Player object which contains one reference to an Inventory and wrapper functionality for that Inventory item.

Task 09 - Spike Summary Report

Implementation

Git Commit History





Task 09 - Spike Summary Report

Code

```cpp
//Gameplay state definitions
STATES GameplayState::update() {
    std::string action;
    std::string object;

    std::cout << ">> Action: ";
    std::cin >> action;
    std::cout << ">> Object: ";
    std::cin >> object;

    std::cout << std::endl;

    // Using this big chunk of if-else statements until command processor is build
    if (action == "quit" && object == "game") { return STATES::QUIT; }
    else if (action == "show" && object == "highscores") { return STATES::VIEW_HoF; }
    else if (action == "show" && object == "inventory") {
        _player.showInventory();
        return STATES::GAMEPLAY;
    }
    else if (action == "take" && object == "twigs") {
         _player.takeItem(_bundle_of_twigs_ptr);
        return STATES::GAMEPLAY;
    }
    else if (action == "take" && object == "petrol") {
        _player.takeItem(_petrol_and_lighter_ptr);
        return STATES::GAMEPLAY;
    }
    else if (action == "drop" && object == "twigs") {
        _player.removeItem(_bundle_of_twigs_ptr);
        return STATES::GAMEPLAY;
    }
    else if (action == "drop" && object == "petrol") {
        _player.removeItem(_petrol_and_lighter_ptr);
        return STATES::GAMEPLAY;
    }

    else { return STATES::GAMEPLAY; }
}
```

The update Gameplay state containing a placeholder command system which is extremely limited. However the implementation of taking and remove items from the inventory is abstracted nicely.

Main.cpp | Inventory.h

GameDataStructures | Inventory

```cpp
#pragma once
#include "InventorySlot.h"
#include "InventoryItem.h"

class Inventory {
private:
    InventorySlot* _head = nullptr;
    int _size = 0;
    int _max_size = 64;

public:
    Inventory();
    Inventory(InventorySlot* head);

    InventorySlot* getHead();
    InventorySlot* getTail();

    void addItem(InventoryItem* item_ptr);
    void deleteItem(InventoryItem* item_ptr);
    void show();
};
```

The Inventory class is the linked list manager with functionality to find the tail, add and remove items

Main.cpp | Inventory.h | Inventory.cpp

GameDataStructures | Inventory

```cpp
void Inventory::deleteItem(InventoryItem* item_ptr) {
    InventorySlot* previous_ptr = nullptr;
    InventorySlot* traversal_ptr = _head;

    while (traversal_ptr->getItem() != item_ptr) {
        previous_ptr = traversal_ptr;
        traversal_ptr = traversal_ptr->getNext();
    }

    previous_ptr->setNext(traversal_ptr->getNext());
    traversal_ptr->setNext(nullptr);
    _size--;
}

void Inventory::show() {
    std::cout << "Zork(ish) :: Your Inventory " << std::endl;
    std::cout << "----------------------------------------
    std::cout << std::endl;

    InventorySlot* traversal_ptr = _head;
    while (traversal_ptr != nullptr) {
        traversal_ptr->getItem()->printDescription();
        std::cout << std::endl;
        traversal_ptr = traversal_ptr->getNext();
    }
}
```

```cpp
#pragma once
#include "InventoryItem.h"

class InventorySlot {
private:
    InventoryItem* _item;
    InventorySlot* _next;

public:
    InventorySlot();
    InventorySlot(InventoryItem* item_ptr);

    InventoryItem* getItem();
    void setItem(InventoryItem* item_ptr);
    InventorySlot* getNext();
    void setNext(InventorySlot* next);

    void about();
};
```

The Node is represented through the InventorySlot object. A very basic object whose whole purpose is to hold and object and a reference to another slot.

```cpp
#include <iostream>
#include "InventorySlot.h"
#include "InventoryItem.h"

InventorySlot::InventorySlot() {
    _item = nullptr;
    _next = nullptr;
}

InventorySlot::InventorySlot(InventoryItem* item_ptr) {
    _item = item_ptr;
    _next = nullptr;
};

InventoryItem* InventorySlot::getItem() { return _item; }

InventorySlot* InventorySlot::getNext() {
    if (_next != nullptr) { return _next; }
    else { return nullptr; }
}

void InventorySlot::setItem(InventoryItem* item_ptr) { _item = item_ptr; }
void InventorySlot::setNext(InventorySlot* next_ptr) { _next = next_ptr; }

void InventorySlot::about() {
    std::cout << "data: " << _item << std::endl;
    std::cout << "next: " << _next << std::endl;
}
```

There's still a few things left to make this Inventory system work in a game setting which will be implemented for the next stage of Zorkish. This includes garbage collection on the inventory, the destructor and node deletion wasn't implemented as the game loop is so short the inventory closes before anything too bad happens…

What was Learned?

> 💡 This Spike revolved around understanding different methods of data encapsulation within a game, with emphasis on the decision making process regarding choosing the most correct container under a given circumstance.

Additionally, the task allowed me to

- Refresh my UML knowledge
- Understand pointers and references in greater depth
- Improve debugging skills
- Expand the scope of my code performance knowledge to include container types
- Make an Inventory for a game :)