Creating dynamic video-game inventory, structuring stored objects and displaying them through the graphical user interface.

Akvile Krikstaponyte
Affiliation not available

***Abstract*—This paper investigates data structures' use in the game development. The project aim was to create an inventory system, that automatically groups, counts and names game objects dynamically in an efficient way. Moreover, this paper presents the design and implementation of the game inventory system, developed with the use of data structure known as dictionary. Furthermore, to demonstrate how the system works, graphical user interface was created. System appears to be very fast with large amount of objects, both in insertion and retrieval. This was achieved by storing a single instance of the object and incrementing only a count. Some functions used recursion principle, to avoid looping if not necessary.**

Introduction

In computer science, a specific way of storing and organising the data is known as data structures. Each data structure can be designed and implemented to fit the specific purpose and needs. There are many data structure types, such as arrays, linked lists, queues, stacks, heaps, binary trees, hash tables and more. From the fist glance, data structures themselves might seem not that useful, however they are indispensable when operated in particular applications with certain algorithms, e.g., sorting, searching, deletion, insertion and so on. It allows to access and manipulate data more efficiently. [1], [2]

Data structures are widely used in game development. In video-games, operating data with certain algorithms can be used to create complex interactive experiences for players or model real-world situations and objects. [2]

For example, data structures can be used for a scene management in games. Video-games can get way too complex for any hardware to handle at once, e.g., thousands of polygons are rendered, physics has to be processed, special effects have to be drawn and updated in milliseconds. Therefore, data structures in combination with algorithms are used to speed up and optimise the performance. [2]

Furthermore, data structures can be used for artificial intelligence (AI) in video-games. Various data structures and algorithms are used to control the behavior of dynamic game elements, even in simple AI systems. [2]

Data structures are also used for the dynamic physics in video-games. Physics in games handles realistic representations of laws of physics (e.g., gravity) in object and environment. Those forces result in object movement or interaction with other objects or environment, e.g., collision. Physics and collisions holds their own set of data structures and algorithms (e.g., rigid bodies, point masses, algorithms to apply external forces, resolve interactions, etc.) that are carried out to allow objects to interact in a real time with each other and their environment. [2]

Moreover, the use of data structures can be seen in a video-game inventory system development, since game inventory systems strive to be dynamic, with quickly accessible and retrievable objects. Most of the seen examples would be implemented with dictionaries instead of just static arrays. Arrays can offer quick insertion and retrieval, however size has to be predetermined. Dictionaries, on other hand are dynamic, and will likely to be much smaller in size than arrays. Furthermore, dictionaries require Keys to retrieve elements, therefore route is often direct and as O(1). Some inventories do use linked lists, however those seem to be rather slow.[3]

Seeing how useful data structures and algorithms can be, we chose to take an advantage and use it in our own game production. This work will focus on gathering more information on already implement inventory systems with the aid of data structures and algorithms, designing and improving our own already existing inventory system for the game we were developing for the past few months.

Related work

It is very satisfying in a video-game to loot and equip one's character. For that a placeholder for collected items throughout games are usually implemented and called inventory systems.

In game development inventory system is not as complex in comparison to other operations such as rendering or physics. However, it is still an important aspect for most adventurous games. Data as game objects has to be organised and manipulated in a way to complement the overall functionality of the game and it's robustness, as well as be presented to the user in understandable fashion.

According to developers' discussions in forums, there is no "correct" way to implement inventory systems and it is highly dependent on the context and purpose of the game and it's overall functionalities. However, when researched, it appears that the most common way to structure inventory systems seemed to be by using arrays, lists, or more complex structures as dictionaries. All of the listed examples below are mostly drawn from the programming forums - discussions amongst game developers.

Arrays and Lists

More basic data structures used for inventory systems are arrays and lists. One example suggests, the building of a stackable inventory system with the use of lists. Here, different types of items can be stacked till pre-set limit, before filling the inventory slot. First of all the base class for all inventory items are defined, where different types of items are inherited from this class. FIGURE 1

Then item classes (e.g., "Plant.cs") are created, where item can be stacked 50 per inventory slot (see Figure 2.).

So, there would be a static class created that obtains a list with all the items that could be encountered by the player. Then, the class where player would store the inventory should be created. The inventory system class, would hold a list that holds all the items that are currently stored in the inventory (see Figure 3.).

When any item is being added to the player's inventory, the "AddItem" method is called, passing in the item object, and the amount which is being added to the player's inventory.

With the "AddItem" method an inventory slot that already contains an item with the same ID as the item passed into the method, which hasn't reached the maximum stack size can be found. As much items can be put in that inventory slot as possible. If the limit is reached, and still there is more to add, or if there isn't a partially-filled slot with that item type, the remaining quantity can be added to the another slot. [4]

Another very similar example implemented with an array was found.

The base class "Item" that all of the item types are descended from is created (see Figure 4.) Then the created child classes e.g., Armor, "Item" variable can be used to store any of the derived classes.

The "Items" is an array, where any Item-derived object can be stored, e.g., Armor or Weapon (see Figure 5.). "FirstAvail" method adds items to the array, where there is a space for it. When one of the items is in use in the Inventory system, the way how to deal with them has to be determined.

The code snippet above (see Figure 6) shows how to behave with collected items. The operation "as" checks the true type of the object and returns null if it doesn't match. So if you store a "Weapon" item in slot 0 of the inventory, it will not try to use it as an "Armor" item.

Linked lists and Dictionaries

According to the wiki book "Learning C With Game Concepts/Designing A Roleplaying Game." [5] an inventory system should contain list of items (e.g., objects). Inventory should also be able to do such processes as insertion, deletion, and finding of specific or similar items. When same item is added, number of such items increases and not the instance. Each of the created items should be inserted in an item node. Each node can be linked to other nodes to form a data structure, e.g., linked lists or trees. [5], [6] In this book's example, inventory is implemented as double linked list, where node points to its own item, forward and backward. [5]

Another similar approach is to create inventory system using dictionary data structure. Data is read into the dictionary form the item class, where all the properties of items, like type, name, etc are initialized. Item type is used as a key in the dictionary. [7] In other cases developers use item itself as the key and then set the value as amount of the items that players are carrying. [8]

DESIGN

Purpose of this work is to demonstrate and create useful implementation for organizing objects. On top of that, organized objects could be showed to the player in the video-game scenarios.

Previous work

The last few months we were developing a video-game, where basic implementation of using associative array was used when storing player-held objects. The system was called backpack, and it was a core for an inventory. This backpack was able to input or find specific elements held inside. It could also hold special items, for example required by the quest, that could not be dropped out by the player. Items that player wanted to store, were picked up or stored directly from where they lay, parented to the backpack game object and reflected in predefined array. Predefined array size, was the size of the backpack capacity, which is a very common and realistic in-game practice.

Previous work limitations

This backpack system had few limitations, therefore more advanced structure was needed. First of all, all objects that were stored initially by the system or picked up by the player were just parented to backpack's gameobject. This meant, that all game objects would be stored in both the array and the backpack transform. That, theoretically leads to a longer object reitrieval time, in case where object id for the retrieval is not known. So for every not known object retrieval, the complexity is O(n). Furthermore, backpack system was not organizing objects in any way, they were just added to an empty array slot (which was also needed to be found) in the array. Object types were also not recognized nor set in the objects themselves. This meant that they could not be grouped to groups as food, weapons, clothes etc, but also that a single item from the specific group could not be retrieved. Backpack system was also not able to visualize objects that are currently stored inside, therefore player could not know what and how many of the items are currently inside. Lastly, backpack system was not a separate class and implemented together with the player class, which is in general not the best practice.

Design requirements

Inspired that backpack objects could be displayed through the inventory in an organized fashion to the player, we have decided to redo the backpack system completely. 3.2 Previous work limitations showed that:

1) Objects should be stored once, if exactly the same object is already store;
2) There should be an indication of how many of each items there are inside;
3) Retrieval should be faster than O(n) - looping through all objects to find one;
4) Objects should be named and grouped to object types.

To fulfill the first requirement, new object should be checked among same type of objects in the backpack, and compared if it has exactly the same parameters. If for example, two objects are similar, although the size is slightly different, they can be overwritten and counted as exactly the same.

System should also be very dynamic. Considering that previous work used a simple array, the first improvement should be to create a dynamic structure, theoretically allowing to store as many different objects as possible. This would also give another improvement - the upgrade of backpack capacity without any restructuring of already existing data.

New system should be able to do at least the minimum of what previous system was able to do:

1) Store object;
2) Initially store objects, before game starts;
3) Retrieve object;
4) Drop all objects, excluding special items required for the game quests.
5) Reinitialization of the backpack (change all objects in the backpack with different graphical style objects)

Lastly, to demonstrate how the new system works, inventory GUI should display what object types and how many of them backpack is storing.

4 Implementation

This chapter will cover how and why specific decisions were made while implementing backpack - inventory system. All other elements as prefabs, models, code, icons, textures, and some sounds were also created by us. However, only the backpack system and GUI will be covered in this part.

We have started by implementing the switch between the use of the new backpack and the old backpack systems. The new backpack system was created in a separate class, called Backpack, instead of keeping the code in the Player class. Firstly, to use dynamic data structure type, instead of associative array we have chosen to use Dictionary to store game objects. Dictionary as the List is a dynamic structure, which does not need predefined size to be known before adding new objects. After, it was decided to use the ObjectType (apple, rock, bottle etc.) as the Key (string) and GameObject (Unity GameObject) as the value in that dictionary. The purpose of that was to be able to find object according to the type. Therefore, all objects that can be picked up and stored by the player, received new property of ObjectType. Later, to add new feature and fulfill the second requirement (knowing object type amounts), another dictionary was created to store object amounts, where Key was again ObjectType, and the value was the amount of it.

Figure 7.

Instead of using two dictionaries, it is possible to keep the game object as the Key and value as the amount (as was seen in example inside Related Work), however, this would require to find object type trough separate function in order to group objects according to the ObjectType.

4.1 Storing new objects to the backpack

The storing of new object into the backpack (dictionaries) was simple at the first glance. By default, the Key and new game object would go directly to backPackObjects dictionary. Right after, the same Key and amount of that object would go to backPackObjectAmount dictionary. To put object in the backpack, it was required to check if it is already existing or not. If the type did not exist at all, the new Key value was added to the first dictionary, e.g. Apple_1 (apple of type 1). After, same Key and amount of that object was added to the second dictionary.

Figure 8.

More complexities revealed, when trying to store the same ObjectType object but with different parameters. See 4.2 Checking if there is the same object in the backpack

for more details.

4.2 Checking if there is the same object in the backpack

If object did exist in the dictionary, it was required to know if it is similar or exactly the same object. Therefore, new function CheckIfObjectIsTheSame(gameObject), that checks if given object is the same as already stored in the backpack was created. This function, was set to be bool type, to simply retrieve the answer, if object is the same or not. If answer is true, function finds which Key in the dictionary has the same object, and amount of that object is incremented by one in backPackObjectAmount dictionary (second dictionary that keeps the amount of objects). Otherwise, if answer is false, meaning that object is not the same, but such type exists, Key of the same object type is incremented by one (two types of apples plus one equals to Apple_3) and added to two dictionaries.

The use of ObjectType gave possibility to recognize what type of object it is, therefore, CheckIfObjectIsTheSame does not need to compare all objects, but only of that type. For example if the new object is of type “Apple”, it only checks among different apples of Apple_Type_1, Apple_Type_2 etc. Function CheckIfObjectIsTheSame, finds if object is the same or not by checking its size, tag, ObjectType, mesh and weight, therefore individuality of object is not lost. Function does not need to check whether majority of set property requiring components exists at all (for example mass of Rigidbody

component), since if object is in the backpack already, it means that it did had these properties. However, some exceptions needed to be made, for example some objects stored in the backpack can use animations, therefore mesh type is different than a normal skinned mesh type. Therefore checking if mesh component exists was a requirement. The function could be easily advanced in precision, by also checking material, texture, what sounds it has, etc. If one of the predefined checks is not true, object is categorized as a similar and not the same. Picture below shows part of this function, that returns answer true or false.

Figure 9.

4.3 Retrieving the object from the backpack

TakeObjectFromBackPack(string objectKey) was created to be able to take out a single, specific object from the backpack, founded by the inserted Key into the function. Function was only ran, if received Key was not an empty string or if there is at least one object in the backpack. The retrieval from the dictionary is very simple, since no looping is needed (in general). By default, object can be retrieved only when specific Key is known. If it is known, it is only a single line of code, where retrievedObject is equal to backpackObjects value retrieved by the Key. However, if exact Key is not known, only ObjectType, some looping is required. This is useful, for example when retrieving any apple from the dictionary, when player presses a button to eat some apple (assuming that Apple_1 or any random apple type no longer exists, it will retrieve Apple_2 or other). The list of same ObjectTypes is created by another function called GenerateSameTypeObjectList(objectKey). After function is called, list and count of unique ObjectTypes as apples is generated. After that, loop finds the first ObjectType (most often it is looped once, since e.g. type of Apple_1 is most common as any new apple inserted in the dictionary will receive such key value automatically in this implementation) and instantiates such game object in front of the player.

Figure 10.

This function could be optimized by checking whether specific Key is given (Bottle_2 and not only Bottle). Therefore, if the specific Key is given, only required check before the retrieval is to check if that Key exists in the dictionary at all. Our specific implementation required that special objects would be given to player to hold right after they are taken out from the backpack. For example, it was required that player drinks water after water bottle is taken out. In this case specific Key is give to the function, and retrieved object was sent to the Player class HoldObject(gameObject) function.

Figure 11.

There were more special cases, for example when player already holds object. Object held in hands was dropped out automatically and new special object would be held in hands by the system instead.

Lastly, dictionaries needed to be adjusted to reduce or and remove Key completely if ObjectType was the last of it's specific kind. For example, if Apple_3 amount was 3, and one was taken out, value will be reduced by 1 in the dictionary. If Apple_3 amount was 1 and one such apple was taken out, two dictionaries would remove Apple_3 Key completely.

Figure 12.

Later, new useful functionality was added to take out specified amount of objects. Override method of TakeObjectFromBackPack(string objectKey, int amount) was created, that iteratively will call TakeObjectFromBackPack(string objectKey) until object amount becomes zero. If given amount is greater than available object amount, recursion will be stopped after all existing objects are dropped.

4.4 Removing all objects from the backpack

It is useful to drop all items from the backpack, considering player's time (if he or she wants to drop all items out) and that items can be taken away, lost or any other in video-game possible situation. Function ThrowoutAllObjects() was created in iterative fashion, without the use of loops. Function will call itself after a single item was dropped out to drop the next item. Before it calls itself, it does few checks and adjustments of dictionaries. To track how many items and how many ObjectTypes remains, after dropping a single item, dictionaries are adjusted as in 4.3 Retrieving the object from the backpack.

As in majority video-games, a quest items (in our case the specialItem) cannot and should not be removed from the backpack by the player this was also implemented in this backpack system. The check and implementation of special object was problematic. Multiple approaches were considered, one of those was to skip special item if it is found. Then, function would call itself, with a different Key and try to remove that object. However, it appeared that the constant check if item was special and empty recursive function call (if there is a huge cluster of special items) was not needed. It was decided to store special items in the beginning of the dynamic List objectTypes and when recursive function hits the first special item, function stops. After that, the only remaining items that stays in the backpack are the specialItems.

4.5 Puting initial objects into the backpack

When experimenting, and placing huge initial amounts of object into the backpack, Unity GUI limits were reached (errors were given), as a huge amount of objects were shown in the Inspector, where they were placed. It was realized that this is not a smart approach. Second list initialBackPackObjectAmount that holds amounts (integer) of initially placed objects were created, see picture XXX. Therefore, a single object could be placed in the inspector, and amount of such objects indicated by that integer. After, code was adjusted to not to loop (initially on start) through this amount when adding new items. Instead initialBackPackObjectAmount array numbers were reflected to backPackObjectAmount(Key) dictionary, when initial objects were inserted. More convenient and clean method would be to access the dictionary directly from the Inspector and two additional lists of initial gameo bjects and amounts would not be required. However, Unity currently does not support serialization of dictionaries (used

in the inspector, to reflect properties held in the data types), therefore, dictionaries are not accessible through the Inspector.

4.6 Inventory GUI

The new backpack system received the graphical user interface (GUI) showing how objects are grouped, named and counted. This required to create some graphical elements as object icons, basic inventory background and text fields. Since the grouping, object count and naming was prepared by the dictionaries, only remaining implementation was to display this information on-screen to the player. This required the creation of numeric grid (pre-defined positions for icons) to store the icons when player turns on the inventory. Picture below shows how the inventory GUI functions. Three different green apples are in different sizes or weights, therefore they are grouped as different objects.

Figure 14.

Inventory is made to be flexible and dynamic - icons and names does not need to be assigned before in the game engine. Therefore, if object that is completely new and created by someone else these elements will be still shown in the GUI inventory. This was made by leaving important and individual properties inside the object itself.

4.7 Implementation conclusion

All functionalities of the old backpack was created and tested in our video-game. In addition to that, new functionalities as inventory GUI for the player was created. At this point it is minimal, and might experience some issues, however it is a good and pleasing milestone for us as the developers. Full code of the backpack can be seen in the appendix A while GUI code can be seen in appendix B.

5 Play-Testing

It is hard to say how fast this system is, however in most of the video-games inventories will not include ten thousands of objects (also in backpack of the player). Nevertheless, the loading time of the game is very fast, when using this new system with 100 000 000 initial objects in the backpack. When system is disabled completely (and backpack is empty), the loading time does not differ from when such big amount of objects is loaded. Both loading times were approximately 3 seconds. Theoretically such huge amount could also be taken out from the backpack, although in the testing, hardware started struggling after approx. 5000 objects were taken out.

Figure 15.

6 Conclusion

6.1 Critique

In comparison and appart from new features, this system is much more complex than previous backpack system. Many more checks and operations are done per object in insertion and retrieval. On top of that, since no multiple same objects are held by any array, nor “physically” in the game hierarchy, new objects have to be instantiated again in real-time. On top of that, their transform properties as the position, rotation transform-parenting etc. has to be set. While the old system had to just unparent the object from the backpack gameObject and enable it. It might be, that due to instantiation of the new and more resource-craving objects (many at the same time) “performance decrease spikes” could appear in some situations. Although, we did not experience this in our small scene with simple prefabs. Furthermore, as mentioned before, there is no need for ten thousands of objects to be held in the backpack, therefore overall performance might be worse than just having predefined array and parented gameObjects in the backpack. This is not clear - test to compare both systems should be made.

6.2 Current limitations and future works

GUI is not finished, and it will malfunction if objects are taken out or stored when it is on. This should be fixed, by refreshing lists and reflecting that back to GUI immediately. This will be also useful, when implementation of rearranging the objects in the inventory will be made. Furthermore, all basic functions as taking out one item, taking out all items, equipping item, eating food etc. of inventory should be made. Majority of these functions are already created, therefore the remaining part is to work on the GUI and tie these options to appropriate inventory buttons.