Got-A-Head? Game

Rucksack Roadmap

# What the Rucksack Is For

The Rucksack is used by the player to carry objects they have picked up or been given or bought in the course of the game, while leaving their hands free. The idea is that they can then get the item out of their rucksack when they need to use it, if they remember it is there. Alongside the rucksack is the belt. The belt can also be used to carry items in ‘loops’. So, if they carry a sword, they would draw it from a loop in their belt.

From a curriculum point of view, the rucksack and the belt are there to get the player to remember, to organise and to balance decisions. The player has to use up a bit of game time retrieving stuff from the rucksack, and a smaller amount of game time getting something from their belt. The time taken to get an item they need can increase markedly if the rucksack is badly organised

The rucksack is made up of a set of rucksack pockets of varying sizes, mirroring a normal rucksack that will have a large central volume, and smaller zipped pockets on the outside. How quickly you can retrieve an item will depend upon how many items there are in a pocket and the comparative volume of the article you want to extract and the volume of the pocket you are retrieving it from. You also really want to know which pocket the item is in. If you have to look in multiple pockets this will take extra time. Therefore, the player should think about how they want to organise their rucksack when the put a new item in it so that they can get hold of it as quickly as possible when they need to use it. Obviously, they will need to have some items to hand much more often than others, so they have to make a decision about what they need to use most often. If they go into a new situation, they may want to move things around in their rucksack in advance as ‘bad things’ might happen if they are caught without say, ghoul repellent when they walk through the wrong graveyard.

The belt should allow them to carry guns, swords, knives, axes, hammers and such like while keeping their hands free. The maximum length of any item looped on the belt should be 2/3 of the waist height of the player. Getting something out of the belt is obviously going to be much, much quicker than having to get things out of the rucksack.

The other thing the player will have to bear in mind is how heavy the rucksack will be. If the rucksack is too heavy the player will move more slowly or have lower dexterity and is more likely to get hit by an opponent or will be less able to get over an obstacle. In addition, the player will get tired more quickly and have to stop more often.

## Adding Objects to the Ruck Sack or Belt

An object can be picked up by right clicking on an object; “Pick up” will be one of the right click menu options When the player selects pick up, they will then have a sub-menu, with the options, right hand, left hand, belt and rucksack. If the user selects belt or rucksack the player may be able to transfer the object to their belt or rucksack.

### Restrictions on Adding an Object to the Rucksack

The player cannot put an object into a pocket of the rucksack if

* The object’s dimensions are larger than the pocket’s dimensions
* The sum of the volumes of the objects already in the pockets plus the volume of the object to be added exceeds the volume of the pocket.
* The sum of the mass of all the objects already in the rucksack plus the mass of the new object exceeds 30kg.

### Restrictions on Adding an Object to the Belt

The items hung from the belt must weigh less than 5 kg. The object must have dimensions smaller than 75 cm x 25 cm x 10cm

## Removing Objects from the Rucksack or Belt

You can access the rucksack or belt by clicking on the rucksack symbol on the screen. This will display the ‘rucksack window’ showing a large image of the rucksack and the belt and all the objects currently hanging from the belt.

To find the object in the rucksack, you must click on a pocket. You then get a list of objects held in that pocket. You then have to right click on the object to select it.

All the objects on the belt are displayed automatically on the rucksack window, so to select one you just click on it.

To retrieve the object on the belt, you right click on them, then you can select to move the object to the player’s left hand, right hand, or perform any other action permissible on the object immediately.

# Script Components Implementing the Rucksack and Belt

My knowledge of Unity isn’t great. But I guess what I would implement as a class in C# should be implemented as components and then built into prefab and prefab variants that can be turned into game objects. But, when I say Component, Prefab, or Prefab Variant, please remember I’m a newbie at this and I am guessing.

## Game Objects That Can Be Stored In a Belt or Rucksack

All objects in the game which can be picked up, given or bought must have the following properties

* Its maximum height, width and depth
* Its circumference
* Its mass
* A name

Except for the circumference I presume these are all present in standard Unity components. I’d set the circumference as , where the axes are the object’s height, width and depth.

On this subject, ideally, I’d like to add the volume as a property. There are methods of working out the volume of the tetrahedrons in the mesh, but that looks like another day’s work. In the meantime, if you’ve done everything else, we can add a volume property and enter it manually and then add a volume check to the rucksack’s functionality.

## The Rucksack

There may be many different rucksacks with different configurations of pockets in the game. As I said earlier, I presume these will need to be implemented as prefabs and prefab variants, with special scripted components added to handle properties that are not handled elsewhere.

To simplify this for coding purposes, I have broken this down into three types of prefab (I hope).

* The Rucksack Prefab that represent the strapping that goes on the character’s back. So, it’s pretty boring and its only job is to support the big central Rucksack pocket.
* The Rucksack Pocket Prefab is the more interesting bit. Its properties determine how much stuff you can put in. It also can carry smaller rucksack pockets on the outside of it.
* The Rucksack Pocket Fastening Component or fastener that allows you to specify different fastenings for a rucksack pocket; e.g. zips, draw strings with toggle, buckles etc. Some fastenings allow you to get into a rucksack pocket faster than others. For fun, we might create magical fasteners that do nasty things if you try to open them the wrong way. We could also get fancy later and add a probability that the fastener can come undone and let contents drop out.
* One or more “Shove In” components that provide rules to decide whether a given game object can be put into a rucksack pocket. The idea is that different variants of this component can be used so that we can have magic rucksacks that like Wee Granny’s Bag or Hermione Granger’s Bags can hold infinite amounts of stuff, including kitchen sinks.
* A “Rummage About” component which governs the order in which items in a rucksack pocket can be got out of a rucksack pocket. The idea is that different variants of this component can be used so that we can have magic rucksacks that give out what you need first time, every time and real world rummaging about components that mean you have to take every item out before you find the little item that you really needed that has fallen to the bottom of the rucksack.

When you construct a rucksack from these pockets, you start off by constructing the smallest pocket with a fastening as a property. You can then create the bigger rucksack pocket objects with their fastening and add any smaller pockets which are to be attached to them. You carry on doing this until you construct the central rucksack pocket. You then construct the rucksack object and give it the biggest rucksack pocket as a property.

### Building up A Rucksack with Lots of Pockets

The rucksack is then made up by adding extra rucksack pockets on to one of five points of the first pocket. Those five faces are “top”, “bottom”, “left”, “right” and “front”. The sixth ‘back’ face of the initial pocket normally faces the back of the wearer (or the larger pocket to which a secondary pocket is attached), so no extra pocket can be put there.

More than one extra pocket can be added to a face.

### The Rucksack Prefab & Component

The Rucksack Prefab will need a Scripted Rucksack Component. That component script is very simple. It has three properties, that can’t be varied after it is instantiated.

* The main rucksack pocket
* The time it takes to take the rucksack off, in whatever Unity’s standard units of time are.
* The time it takes to put the rucksack on (which must be longer than the time it takes to take the rucksack off).

### The Rucksack Pocket Prefab & Component

#### Rucksack Pocket Component Properties

The Rucksack pocket Prefab will need many more properties than the Rucksack, so some will definitely need to be implemented through a scripted component. I’m not sure which properties should reside in standard game components and which need to be scripted. Unless I’ve stated otherwise, they are all invariant after a pocket game object is instantiated out of a prefab.

##### Heights, Width, Depth

The first set describe the dimensions of the pocket. The three values define the cuboid that represents the pocket’s dimension. These are orientated in relation to how a person would wear the rucksack when standing up. Height is aligned to the person’s backbone, so vertically. Width is left to right across the person’s back. Depth is orthogonal to the other two horizontally away from the person’s back. Their units are whatever UNITY defines its game units as (metres?)

##### Volume

Volume is derived from the first three, i.e. height x width x depth. This determines the total volume of things that the pocket may hold.

##### MaxWeight

MaxWeight is a measure of the maximum weight that the pocket can support before it gets ripped off the rucksack. This will be in the UNITY mass units, (presumably kg).

##### Fastener

Fastener is the fastener prefab variant.

##### Contents

Contents is the list of game objects that are in the rucksack pocket. I’m not sure what C# script type you would use to fit in with the Unity component inspector, but In the C# world, this would be a SortedDictionary<GameObject>.Values The idea is that the Rucksack does not necessarily give out objects in the same order that they are put in. A SortedDictionary is composed of {key, value} pairs. The keys would be the order that the objects can be extracted. The values would be the objects themselves, hence the “.Values” list is what you want to display.

##### ShoveIn

This property holds one or more ShoveIn component that will tell you whether you can add a given game object to a rucksack pocket. All ShoveIn components of the rucksack pocket variant must give the okay before a game object can be added to the rucksack pocket.

##### RummageAbout

This property holds a RummageAbout component that will tell you how long it will take to remove an item and in doing so, how many items you must take out of a rucksack pocket, before you find the thing you need.

##### Top, Bottom, Left, Right, Front

These are properties that you can add smaller pockets to. The pocket has to have logic that you can only add a smaller pocket to it, if that pocket has small enough dimensions that the small pocket’s back face would fit on the face of the larger pocket that it is being attached to. If two smaller pockets are attached to the same face, then the total area of their back faces should be smaller than the face of the larger pocket they are attached to.

#### Rucksack Pocket Component Script Methods

##### AddItem(GameObject) returns {bool successfullyAdded, int timeDelay}

This should be called from the menu item click, or whatever other event handler is used to put an item into a rucksack pocket. This should go down the list of ShoveIn components and execute their delegates Func<RucksackPocket, GameObject, {bool added, int time}>, adding up the returned times and anding all the Booleans, which it will then return. Obviously, it should also do things like turn gravity off (or transfer it to the rucksack) and turn kinematics on, the standard Unity stuff that you do when you pick up an object.

##### RemoveItem(GameObject) returns {bool successfullyRemoved, int timeDelay, IEnumerable<GameObject> ItemsRemoved}

This should be called from the menu item click, or whatever other event handler is used to take an item out of a rucksack pocket. This should take pass the first item in the contents dictionary to the predicate in the RummageAbout component to see if it can be taken out, and if it can, what the delay is in doing so. It should carry on doing this until it gets to the required object, or the predicate returns false, so an object in the contents cannot be removed (and so the desired object cannot be got to). The method will then return the combination of whether the desired object was removed, the time taken to do so and a container of all the game objects removed, together (returned as an enumerator). It will also need to turn gravity back on and turn kinematics off and any other things the Unity engine requires to be done when putting down an object.

### The Fastener Prefab, Variant and Component

The fastener component governs how long (if at all) a player can open a rucksack pocket or how long it takes to tie on a game object to the belt. To start with, we should probably just have a delay. I’m assuming that we will need a Fastener Prefab that we can attach the component to, and from which we can then generate several variants, which have the components’ properties set to different values.

#### Fastener Component Properties

##### OpenAndCloseDelay

The amount of time in seconds that it takes to open and close the fastener.

#### Fastener Prefab Variants

Fasteners are used for both rucksack pockets and belt loops. For belt loops they only govern the closing time.

##### ToggleDrawStringFastener

Two seconds to open and close

##### BuckleFastener

Five seconds to open and close. But these would probably be more secure than the ToggleDrawStringFastener, when we start to allow the fastener to come undone.

### ShoveIn Component

These are rules that must be executed to check whether a game object can be added to a pocket. So, it will probably be a predicate along the lines of the delegate Func<RucksackPocket, GameObject, {bool added, int time}>. So, we’d have on ShoveIn component to check for size, another for volume, another for weight, another for unknown magic or whatever. I’d just start off with one for volume, that returns true if the height, width and depth of the pocket are greater than the X,Y,Z of the object and returns the number of seconds delay in ‘shoving in’ as being the same as the number of game objects already stored in the pocket.

### RummageAbout Component

These give an order to object that are held in a pocket. In coding terms, the script could provide a C# class that implements IComparer<GameObject>. You would then pass that into the SortedDictionary constructor to correctly order the Game Objects. My suggestion would be to use the reverse order on the sum of the X, Y and Z sizes of each object. Thus, the object with the largest X, Y, Z sum would be the first out, and the one with the smallest X, Y, Z the last out. This would simulate the fact that you are likely to find the big stuff first and then get to the smaller stuff later.

The component should also have a predicate Func<RucksackPocket, GameObject, {bool removed, int time}> which should return whether the game object can be removed and how much time it will take to remove it. N.B. The game object being removed must be the first one in the contents dictionary.

## The Basic Rucksack

For the purposes we will start off with just one kind of rucksack, “Basic Rucksack”

### Basic Rucksack Properties

The Rucksack has the following pockets that may contain objects, all dimensions are height x width x depth.

* The main pocket attached to the rucksack: 1 x 60 litres, 60 x 40 x 25cm
* Attached to the top face of the main pocket: 1 x 10 litres, 10 x 40 x 25cm
* Attached to the right and left face of the main pocket 2 x 4 litres, 40 x 4 x 25cm

The rucksack can take up to 30 kg in mass

## and Belt Loops

The belt and belt loops work much the same way as rucksack and rucksack pockets. The belt can have loops attached to it. The loops can be used to hold one item each. The loops have fasteners just like the rucksack pockets. The fasteners determine how much tine it takes to store an item in a loop. All items can be obtained from a loop in one second time of game play, although that again is arbitrary. I’ll set the maximum number of loops that a belt can have as 20, but that is entirely arbitrary.

## The Belt

Belts are like Rucksacks, but much simpler. There still may be different belts with different configurations of belt loops in the game. As with the rucksack and pocket model, I presume these will need to be implemented as prefabs and prefab variants, with special scripted components added to handle properties that are not handled elsewhere.

To simplify this for coding purposes, I have broken this down into three types of prefab (I hope).

* The Belt Prefab and Component that represent the belt that goes around character’s body. The belt’s job is to support a set of loops.
* The Belt Loop Prefab and Component have properties to hold game objects that meet certain tests. They are tightened with fasteners
* The Fastening Component or fastener allows you to specify different fastenings to tighten the loop to hold the object; e.g. draw strings with toggle, buckles etc. From a game point of view, they work in a similar way to how they do on rucksack pockets, they govern how long it takes to tie the object on to the belt.
* One or more “TiedOn” components that provide rules to decide whether a given game object can be tied on by a loop to the belt. Again, the idea is that different variants of this component can be used so that we can have magic belts that can tie on impossible stuff, like tying an elephant to your belt, which will shrink and be weightless while tied on, but grow back to full size when taken out of the belt.

### The Belt Prefab & Component

The Belt Prefab will need a Scripted Belt Component. That component script is very simple. It has these properties, that can’t be varied after it is instantiated.

* A collection of belt loops
* A maximum time to remove an object from the belt.
* A maximum weight that the belt can carry before pulling the player’s garments down.
* The time it takes to take the belt off, in whatever Unity’s standard units of time are.
* The time it takes to put the belt on.

### The Belt Loop Prefab & Component

#### Belt Loop Component Properties

The Belt Loop Prefab will need properties to be implemented through a scripted component. As with the rucksack, I’m not sure which properties should reside in standard game components and which need to be scripted. Unless I’ve stated otherwise, they are all invariant after a pocket game object is instantiated out of a prefab.

##### Length

The length of the loop, which will be the maximum circumference of the object being tied on.

##### MaxWeight

MaxWeight is a measure of the maximum weight that the loop can support before it gets ripped off the belt. This will be in the UNITY mass units, (presumably kg).

##### Fastener

Fastener is the fastener prefab variant.

##### Game Object

The Game Object if any tied on to the belt by this loop.

##### TiedOn

This property holds a TiedOn component that will tell you whether you can add a given game object to the belt loop.

#### Belt Loop Component Script Methods

##### AddItem(GameObject) returns {bool successfullyAdded, int timeDelay}

This should be called from the menu item click, or whatever other event handler is used to tie an item on using a belt loop. This should execute the TiedOn’s delegate Func<RucksackPocket, GameObject, {bool added, int time}>. Obviously, it should also do things like turn gravity off (or transfer the mass to the belt) and turn kinematics on, the standard Unity stuff that you do when you pick up an object.

##### RemoveItem(GameObject) returns {bool successfullyRemoved, int timeDelay, IEnumerable<GameObject> ItemsRemoved}

This should be called from the menu item click, or whatever other event handler is used to take an item out of the belt loop. I’m assuming the player will always get the item they want out of the loop at a constant time, say one second It will also need to turn gravity back on, and turn kinematics off and any other things the Unity engine requires to be done when putting down an object.

### TiedOn Component

These are rules that must be executed to check whether a game object can be tied on to a belt. So, it will probably be a predicate along the lines of the delegate Func<RucksackPocket, GameObject, {bool added, int time}>. So, we’d have on TiedOn component to check for circumference, another for weight, another for unknown magic or whatever. I’d just start off with one for circumference, that returns true if the loop is longer than .

## The Basic Belt

For the purposes we will start off with just one kind of rucksack, “Basic Belt”. This will have five loops of 30cm all with buckle fasteners.