Ultimate

crafting system

by **polyperfect**

Have a Suggestion?

info@polyperfect.com

Table of Contents

Thanks!

First of all, thank you for purchasing our pack, we really appreciate that! We are putting a lot of effort into this.

Check out our **Facebook Page** for any news.

Handy Links ;)

Other Low Poly Packs

Low Poly Animated Animals

Low Poly Animated People

Low Poly Animated Dinosaurs

Low Poly Animated Prehistoric Animals

Low Poly Epic City

Low Poly Ultimate Pack

Low Poly War Pack

Poly Universal Pack

Poly Halloween

Poly Movie Set

Toolkits

Ultimate Crafting System

2D Packs

Low Poly Icon Pack

Fancy Icon Pack

2D SDF Nodes

Follow us

Facebook

Instagram

Youtube

Polyperfect.com

Updates

VERSION 3.1

New slot constraints: RequireAnyCategory, RequireAllCategories, RequiresItem
TransferUtility no longer throws error if trying to transfer more than source has
Fixed instance handling for Battery using Version 3 Slot Constraints

VERSION 3.0.3

Handling for null text to fix text display error in Item Slot Components while the game is running.

VERSION 3.0.2

Updated the readme with scripting examples, and added a few accompanying demo scripts

VERSION 3.0.1

Fixed text overflow when using items with long names
Recipe categories and Used In/By lists now only show objects from active fragments
Removed some obsolete code

VERSION 3.0

Requires Unity 2020 LTS or higher, up from the previous 2019 requirement

- Item fragment editor UI overhaul
- More powerful, faster searching
- Unified animation components under the name ElementAnimation, for easy animating of position, rotation, scale, and pivots
- Category editing now only shows items/categories from the active fragment
- Item Instances are now saved with InventorySaver scripts
- ItemSlotComponents use a new Constraint system to limit or change items inserted

- ItemSlotComponent contents are now visible in play mode (new [ReplaceWithPeekInPlaymode] attribute)
- New OnItemStackCrafted and OnCraftingSuccess callbacks on Crafter
- Tons of bugfixes and speed improvements

Major API Changes:

- Item Worlds can have their items, categories, and recipes enumerated directly
- Recipe data is now accessed like categories—extension methods GetRecipeInputs and GetRecipeOutputs added for ease
- Crafter's item comparison method is now (only) accessible through code
- Component-based slots use BaseItemSlotComponent—ItemSlotComponent now inherits from this
- Slots can now be linked using events via ItemSlotLink.LinkSlots, or directly in memory via LinkedItemSlotComponents
- All Slot types must implement a Changed event
- ItemSlotComponents now leverage Constraint Collections. You can add new types of constraints by inheriting from StackInsertionConstraint

VERSION 2.0

IMPORTANT UPDATE GUIDE: After updating, Unity will sometimes mess up the interactions on prefabs like the Crafting Station. To fix it, simply do Assets > Reimport All.

First-Person Demo
Support for Shaped Crafting / Table Recipes
Placement System and Conveyor Belts
Item Instance / Unique Item Support
Energy / Item Transfer System
Command System

Improved editor performance with Version Control
Improved versatility of various scripts and functions
Tons of bugfixes

VERSION 1.20

IMPORTANT UPDATE GUIDE: If updating from Ultimate Crafting System 1.1x to 1.2x or higher, you will need to delete the following file: "Assets/polyperfect/Common/ - Code/PolyPerfect.Common.asmdef" Please confirm the capitalization and path before deleting.

The Item World objects have become Item World Fragments. See the Item World Window and Making Gameplay sections for details.
Misc bugfixes
Assembly Definitions and namespaces changed to use “Polyperfect” instead of “PolyPerfect”

VERSION 1.12

Category Editor Bugfix and testing

VERSION 1.11

Small fixes + new Pickup Video

VERSION 1.10

Drop and Pick Up Item Improvements

VERSION 1.07

Fixes: Inventory Opener Revision and Category Label Update

VERSION 1.05

Misc Fixes and Item World made Prefab + new FPS video

VERSION 1.01

Minor bug fixes

VERSION 1.10

First release

Overview

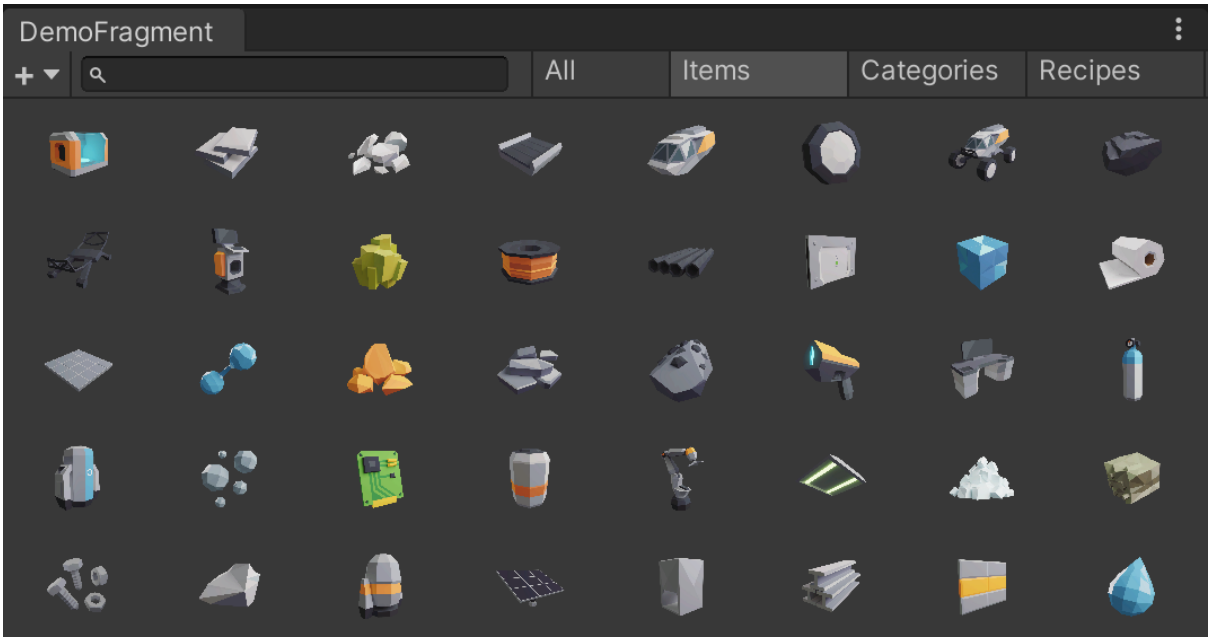
What does it do?

The Ultimate Crafting System is designed with ease of use in mind, and allows for the creation of “Item Worlds”, which are collections of Items, Recipes, and Categories that make up the core of versatile crafting games. Items, recipes, and their data can all be created directly in-editor. Of course, programmers are free to extend the system even more too, as the source code is included!

How do I get started?

The heart of the Ultimate Crafting System is the Item World Window, which you can open from the Window dropdown at the top of the screen. From here, you can edit your Item World by creating new objects or editing existing ones.

Item World Window



Navigating the Window

After opening the Item World Window, you will be prompted to select or create a new Item World Fragment. After selecting one, you'll be able to create, view, and edit items. Create new items by using the + button in the top left, or filter existing objects using the buttons and search field to its right.

Naturally, by typing in the Search field, you will filter the displayed objects by name. Add a space to search for additional matching names. You can use "t:TypeName" to search for objects that have a particular name. For example, the query "ham t:item" will search for any objects with a name containing "ham" that are of a type name containing "item".

You can click and drag objects from the Item Window into slots or fields in other parts of the Unity UI to register them with scripts and other such things. If you simply click an object, you will open it for editing in a small window. Alternatively, if you click with the right mouse button, you'll have the option to Clone or Delete a particular item.

Editing Objects

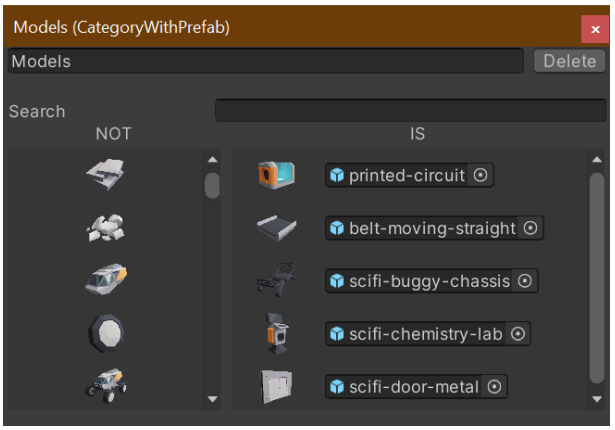
Items

If you create or open an object, this window will appear. You can edit the name of the object at the top, as well as delete it easily.

The first main section allows you to assign the Item to categories and give it associated values like icons or numbers. Drag a category from the left to the right to associate the Item with that Category, or vice-versa to remove that association. Double clicking a Category will open that Category up for direct editing, for convenience. The image in the "_Icon" Category will be what is used for displaying the item in the Item World Window, as well as other parts of the interface.

The bottom section allows you to see which recipes create the item, as well as which recipes the item gets used in. If you click a Recipe, it will open it up for easy editing.

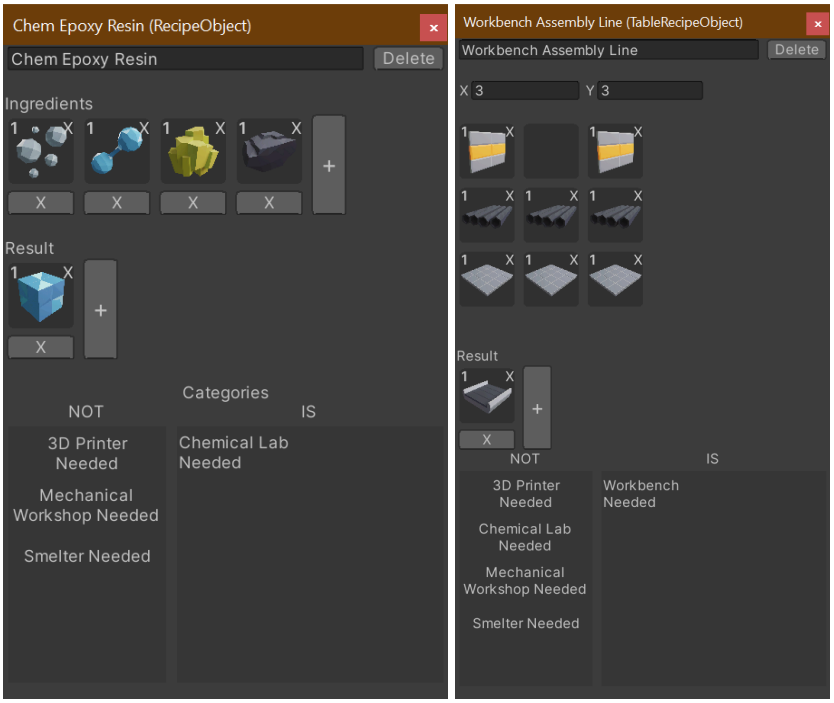
Categories



A Category can simply act as its namesake, but it can also allow you to associate data with objects. Editing one is straightforward--drag an object from the list on the left to the right, and that object will be associated with the current Category.

A simple use case would be to add descriptions to each item, or other flavor text to add depth to the world. For a more complicated use, you could have a “Model” Category that has a GameObject field, and scripts can grab that GameObject from the Category to bring the object into the 3D game world.

Recipes



Recipes naturally are the core of a crafting game. They let you specify how someone can turn some group of items into another. By default, there are two types of recipes you can create--Standard Recipes, which use some collection of ingredients in any order; and Table Recipes, which require items to be placed in a matching arrangement in order to be valid. You can drag items from the Item World Window into the slots in the Ingredients or Results sections to register them, and click the number to edit the amount required.
At the bottom you'll find the same Category-editing view as in the other editing windows. The most common use for Categories and Recipes is to restrict which recipes can be used in what contexts, such as only allowing large items to be crafted on a workbench, or hot items to be made with a furnace.

Making Gameplay

The scripts included in the Ultimate Crafting System are all rather generic, though due to its simplicity there are a number of scripts for UGUI, Unity's traditional UI System, included as well. To get the most out of UCS, you'll want to check out the Scripting section below, but here are some of the broad methods you'll use outside of code.

Item World

In order for item data to be stored and accessed, it is kept in an Item World. By default, the Item World used is set up on a prefab called Poly Crafting World, on which you'll find an **ItemWorldReference** component. On this, you can set the Item World Fragments you'd like to use in your scene. All the fragments you select will be combined into one Item World that all the scripts in the scene access.

Item Stacks and Slots

Most of the scripts assume the use of ItemStacks, which are just an item with a quantity. For example, many of the scripts have an empty slot in them that you can drag an item into. After this, you can edit the number of items by clicking on the number itself. Some slots though are specifically for Categories, or others.

Inventories

An “Inventory” is simply a collection of Slots that can have items inserted or extracted. The **ChildSlotsInventory** component will automatically collect all Slots on child GameObjects. This can then be referenced by anything for insertion, saving or whatever else you want. One inventory can include others--for example, in the demo scene, the player has a ChildSlotsInventory on it, as well as one on the hotbar and one on the hidden inventory page. Thus, the player inventory contains both the hotbar’s slots as well as the hidden slots, while each of them contains only their own. This means with one inventory saving script you can save the contents of both.

Generic Scripts

These scripts don’t really do too much by themselves, but they’re used by many of the other scripts in the pack. Chief among these is the **ItemSlotComponent**. The ItemSlotComponent allows a GameObject to hold on to a singular item stack, as well as specify some constraints, like requiring a particular category. Some common uses for the ItemSlotComponent are for displaying items in… well, a slot in the UI. Or otherwise just keeping track of available space in a player’s inventory.
Another generic script is the **Crafter**. The crafter just exposes various methods useful for crafting to other scripts, as well as to Unity Events. Because of this, you can have Unity GUI Buttons link to it and craft things.

Event System

Unity GUI leverages a very powerful event system, and this same event system can be used for interacting with objects. The Demo Scene in the project makes heavy use of

this Event System. By attaching one of Unity's EventTrigger components to an object, you can receive various events like Click, Drag, and others. These are used in conjunction with the demo script called PlayerCommander, which lets you send commands like move, place, and interact, to the player.

Unity GUI (UGUI) Scripts

Various UGUI-specific scripts are included. These cover common tasks like hooking up inventories to slots in a UI, and so on. All UGUI-specific scripts are prefixed with "UGUI", so you can find them easily by searching in the Add Component menu.

Saving and Loading

The **InventorySaver** script allows you to quickly save and load inventories to a file on the device. You can specify the file name to be used, as well as what extension to give it. The JSON format is used so it can be interacted with easily. The file is saved to the Application.PersistentDataPath, which means it will work regardless of which device you're on. If you disable automatic saving and loading, you can call those methods from code easily as well, via SaveToFile and LoadFromFile. Programmers can also get save objects directly, so that you can use other serialization approaches as necessary.

Placement System

Included in the 2.0 Update is a versatile object placement system. Rather than simply creating an item where you click, placeable items (as designated by the Placeable category) are used by the **ItemPlacer** component to visualize and fine-tune the position with various constraints, such as grid snapping, or connecting to ports on other objects. These ports have other uses as well.

Vessels

At the core of the placement system are Vessels. A Vessel is simply a container that holds onto objects while they're being placed. Vessels can have various scripts attached to them to modify the way that the placed object position is set. For example, the **Placement_GridSnapper** component will let you specify an arbitrary grid that the vessel will snap to. The typical way of specifying which vessel to use is to specify it in an Item Category. The Assembly Line items for instance have a Vessel that will snap them to the ends of nearby existing Assembly Lines.

Placement (Post)Processors

The modifications for placements are handled in code by implementing IPlacementProcessors or IPlacementPostprocessors and attaching them as components to the vessel. Processors are typically used for modifying the placement data or adding custom data, while postprocessors are usually used for modifying the object based on its placement data. While the aforementioned grid snapping in the Vessel section is a great example of a processor, the **Placement_TransformMatcher** is the best example of a postprocessor. It takes the placement data, which is made of position, rotation, and some other bits, and actually applies it to the object. By having this separation, it allows for smoothed translation and rotation of the placement visualization, without affecting the actual placement data when the item is actually set down.

Ports

The primary use of ports is for assisting object placement, making sure that objects snap together without relying directly on a grid. You can add a port to an object as a MonoBehaviour called a **ConnectorPort**. Placement Vessels that include a port snapper will connect to it, if their identifier is compatible. You can create a port identifier in the project view using Right Click>Create>Polyperfect>Port Identifier and specify which it can connect to. A **ConnectorPortGroup** allows you to have a port that can behave in multiple ways or attach to different types. Speaking of different behaviors…

Alternative Port Uses

Ports aren't just great for positioning, they also allow for relationships and communication between objects. The way we demo this is through the electricity system. The Solar Panel has a slot at the back for batteries, and when a battery is attached, the **TransferPort** component allows for transferring power from what the solar panel generates to the battery. The battery itself has a port group--one port can receive power with the **ReceieverPort** , and the other can output power just like the Solar Panel.

Item Instances

An item instance is a unique version of an item that can have its own data, different from those of its kind. You could for example have not just a Diamond Sword, but a Diamond Sword with a +5 Spunk and Wind Aspect. Working with instances necessarily requires coding however. To create an Instance, you'll want to use IItemWorld.CreateInstance, which will clone all the properties of the original item and give you the ID of the instance. The unique data of instances can be set reliably with the cousin SetInstanceData method. A DeleteInstance invocation when the item tragically meets its end is necessary to make sure all this cloned data gets cleaned up. Item instances and their data are saved with the builtin InventorySaver script. Only categories that have data types registered with serialization functions will be saved. You can add these via InventorySaver.RegisterSerializeDeserializePair.

Command System

Another part of the 2.0 Update is the inclusion of a Command System that allows the first-person demo and top-down demos to use the same interaction prefabs while responding in different ways. In particular, when told to interact with something, the top-down demo has the player move to, and when in range, interact with. In contrast, the first-person scene has the player interact with the target immediately if in range. You can create new commands by inheriting from either ICommand or BaseCommand, and adding the appropriate handling to whatever your ICommandable is.

Technical Details

While the Ultimate Crafting System is designed with ease of use in mind, more advanced users such as programmers looking to extend the various systems may find some of the following useful.

Runtime IDs

Every object in the Ultimate Crafting System has associated with it an ID, which is used for comparing or otherwise manipulating objects in the various systems. Runtime IDs are a blittable type, so they can be freely used in Unity's Job System, and do not accrue any garbage through use. An additional benefit is they play nicely with separate Asset Bundles using the Addressables system--standard ScriptableObjects are often duplicated as they are put into multiple bundles, making them inconsistent for references. However, because the runtime Item World only holds on to their IDs, a duplicated Item can be referenced from different Asset Bundles and still be treated as one.

Object Representation

The objects that you work with in the Unity Editor are ScriptableObjects, which makes them play nicely with the Unity UI, Version Control, and other features. However, as hinted at previously, at runtime what is used to identify an object is, and should only be its Runtime ID. The data you set for a given object is instead stored in a lookup, with the Runtime ID as the key, much like in Unity's ECS implementation.

Category Data

As just described, the data for an Item, for example, is not actually stored on that Item. Instead, in edit mode, it's stored within the Category. This means you can theoretically add an entirely new Category to your game through the use of Addressables and Asset Bundles, without updating the game client. When you drag a Category when editing an Item, behind the scenes, it actually edits the Category's ScriptableObject.

At runtime, the Category is looked up by its Runtime ID, and then the data within it is looked up via the Item's ID. Because it's done via ID, the user must be aware of the type of data that it is expecting the Category to have. Categories without any data simply have a bool value of "true".

Interfaces

Instead of using ItemSlotComponents or the like directly, when coding it's recommended to use their implemented interfaces if possible. For example, an IInsert<ItemStack> will let you interact with anything that can have an ItemStack inserted in code. Unity expanded its serialization capabilities in 2020+, though there are still some limitations, so in some situations, if you want to be able to drag and drop references you'll have to use the component type or an abstract class.

Scripting

UCS contains many demo scripts for you to look at, all available in the polyperfect/Crafting System/– Code/Demo folder. While they're written to be understandable and there's a lot to be learned from looking through them yourself, we've selected a few here to go into more detail and help get you started writing your own code to leverage the power of UCS. This assumes you have a late-beginner to intermediate understanding of scripting in Unity.

Default Inventory Script

```
public class DefaultInventory : PolyMono
{
   public override string __Usage =>
     "Fills the target inventory with the provided items on Start";
   public List<ObjectItemStack> ToInsert;

   void Start() => CreateAndInsert();

   public void CreateAndInsert() =>
     GetComponent<BaseItemStackInventory>()
     .InsertPossible(ToInsert.Select(i => (ItemStack)i));
}
```

This script will simply insert the items you specify into the attached inventory on Start. It inherits from PolyMono, which is necessary to get Item Slots and the like to display properly in older versions of Unity. When inheriting from this, you have to specify a Usage, which will show a convenient comment in the UI.

There's two main important lines though, the first is the List<ObjectItemStack>, which shows a modifiable collection of item stacks. Normally when dealing with items in code you'll want to use the literal struct, ItemStack, which contains an ID and a Quantity. Using ObjectItemStack instead lets you drag and drop Unity Objects (including items from the Item Window), into them. The line that makes use of it casts them to the more common ItemStack, then inserts them into the inventory.

Since this example uses a lambda in the .Select part, you'll want to make sure you only use code of this style in response to events rather than every frame, to cut down on garbage collection.

Text Data Display Script

```
public class TextDataDisplay : ItemUserBase
{
   public override string __Usage => "Displays the text in a Text
component.";

   public CategoryWithText Category;
   public Text TargetElement;
   public string DefaultText = "No Description";

   void Start()
   {
       var slot = GetComponent<BaseItemSlotComponent>();
       slot.Changed += HandleSlotChanged;

       void HandleSlotChanged()
       {
           var textAccessor =
              World.GetReadOnlyAccessor<string>(Category.ID);
           if (textAccessor.TryGetValue(slot.Peek().ID, out var text))
               TargetElement.text = text;
           else
               TargetElement.text = DefaultText;
       }
   }
}
```

While another short script, this shows two critical parts of working with UCS—registering with slot events, and accessing the data in Categories for Items. This script allows you to display the contents of the text data of an item. Instead of inheriting from PolyMono, it inherits from ItemUserBase. All this does is give you a more convenient way of accessing the scene's Item World, using "World", instead of a possible alternative ItemWorldReference.Instance.World.

The fields let you specify which Category to draw the text/string data from, as well as which Text object to affect. To start off, it grabs any BaseItemSlotComponent on the object, and whenever its contents change invokes a method to change the text. This

works by taking the ItemWorld and getting a Category-based data accessor using the ID of the relevant field. This accessor is used like any other dictionary, so we take advantage of its TryGetValue method to check if the item inside the slot (looked at via the slot's .Peek() method) has some text data, and if so sets the TargetElement's text to it.

Using a method like this, you can for example make tooltips—instead of updating the text whenever a slot is changed, you could use Unity's Event System's raycasting to check what's under the cursor, or use a script that has an IPointerEnter handler that sets which slot should be inspected for text data.

UGUI Crafter

```
IEnumerable<RuntimeID> GetRelevantRecipes() =>
    World.RecipeIDs.Where(recipeID => MadeCategories.Any(category =>
        World.CategoryContains(category, recipeID)));

public IEnumerable<RuntimeID>
  GetCraftableRecipesGivenInventory(IEnumerable<ItemStack> inventory)
  {
     …
     foreach (var recipeID in GetRelevantRecipes())
     {
       var recipe = World.GetRecipeInputs(recipeID);
       var craftAmount = crafter.GetMaxCraftAmount(inventory, recipe);
       if (craftAmount > 0)
           yield return recipeID;
     }
  }
```

This script is a big part of the demo scenes. It's a demo of how you can go about making the actual crafting part of a crafting game. The section shown above covers getting items from the World, and inspecting aspects about them.

The most critical part is iterating over the World.RecipeIDs. Accessing any of the IDs fields (ItemIDs, CategoryIDs, RecipeIDs) in the World lets you enumerate all of the objects within an Item World. The way this is used here is finding out all of the recipes in the Item World that the player can make, so that they can be displayed.

The GetRecipeInputs() and GetRecipeOutputs() methods are actually just extension methods that wrap the normal category-based data accessors mentioned in the first scripting example. This is a super important concept--categories are your friend, and can contain all the types of data you'd want.

Transfer Port Script

```
var initialSourcePeek = toExtractFrom.Peek();
var sourceHadAnything = !initialSourcePeek.IsDefault();
var transferredAmount =
       transferer?.TransferPossible(TransferAmount)??default;
var sourceIsNowEmpty = ExtractFrom.Peek().IsDefault();
var destinationHasAnySpace =
       toInsertInto.RemainderIfInserted(initialSourcePeek).IsDefault();
if (!transferredAmount.IsDefault())
   TransferEvent?.Invoke(transferredAmount);

if (sourceIsNowEmpty)
   OnSourceEmpty?.Invoke();
if (sourceHadAnything && !destinationHasAnySpace)
   OnDestinationFull?.Invoke();
```

This part of the TransferPort script shows off a number of ways of counting, transferring, changing, and reacting to items. Moving items from one place to another, either conceptually or literally, is super important for logic in crafting games.

As in a previous example, Peek() is used to look at the contents of lots. Here, the .IsDefault() extension method is used to check if the contents of a slot are nil, meaning that the slot is empty. Methods such as RemainderIfInserted will let you know the remaining space if an insertion were to happen, and you can then use that information to alter your logic for things.

The "transferer" variable is just a wrapper for an object that calls TransferUtilities.Transfer(source, destination, amount)--this takes into account the remaining space in the destination and only transfers what there's room for. Unlike

the Remainder methods described above, this one returns the amount transferred, rather than what remains.

The script uses all these values to send a bunch of useful events when certain conditions are met, which can be hooked into anything. For example, you could have a sound play when a container is full, or when a resource has been fully extracted.

Though it's not used in this script, the InventoryOps class contains a bunch of useful methods for transferring and querying collections of items.

Demo Scenes

Controls

Mousewheel - select the object in the quick-bar
Right button - placing the item
Left Button - moving the items in the inventory

First-person
E - release/lock the mouse
space - jump
WASD - movement

FAQ

They’re not stupid questions, just stupid answers from us.

When I try to use crafting stuff in a new scene I get an error saying something about no item world, what do I do?

In order to use the crafting system, you need a reference to an Item World in the scene and set the Item World Fragments you’d like to use on it. The easiest way to do this is to drag the “Poly Crafting World” prefab into the scene. It contains a reference to the automatically edited Item World, as well as some UI for dragging items with the mouse cursor.

I see mention of objects being “Placeable”, but I don’t see how to do it in the code or scene, can you place objects like the 3D Printer and Chem Lab?

Yep, you can place objects in the demo scene by highlighting them with the mouse wheel, then right-clicking to enter Place mode. Then you can left click to place, or use the mouse wheel to rotate the object depending on how the Vessels are set up. See the Placement System section for more info.