What this asset is:

Factory Builder Template is a code template allowing you to quickly create a factory builder-like game, you can find here basic machines types typical for this genre like a miner, smelter, crafting machine, items container, conveyor belt.

To get started please play around with the demo scene in both editor and runtime mode. To see how code is used in practice please see how basic machines are implemented and how their prefabs are constructed.

All items and crafting recipes are using scriptable objects to make them easy to create and modify. You can change models and how machines communicate their current state without any coding thanks to unity events that some machines utilize (f.e. see smelter prefab and how events are connected to particle system to turn it on when the machine is working).

Thank you for buying my asset!

If you have any problems, found any bugs, have some questions or suggestions please let me know at: ***piotrtplaystore@gmail.com***

Table of contents:

Scripts:

Machine.cs

The most important class in the project, every object that can send/receive/process items inherits from this base class, also this class keeps track of all machines in the current scene (used by FactoryBuilderMaster.cs to update machines and save load system).

Variables:
static List<Machines> - list containing all machines in the current scene
List<MachineInput> inputs - list containing all machine inputs
List<MachineOutput> outputs - list containing all machine outputs

Functions:
public virtual bool ReceiveItem(Item item, MachineInput input) - the method used to receive an item from another machine, item can be accepted or declined

public virtual void MachineUpdate() - the method used to update all machines, when writing code here be aware that it can run on a separate thread so make code thread-safe. If you don't want to run this method on a separate thread but on the main thread uncheck the option called Run On Separate Thread in FactoryBuilderMaster

public bool IsInitializated() - MachineUpdate() will be only called if the machine returns true in this function, machines set this flag at the end of their Start method

public int GetMachineID() - ID of machine that is supposed to be unique and persistent if a machine doesn’t change its transform, ID is based on object position and rotation used in save load system

MachineInput.cs

Machine input class used to be able to connect other machine output with this machine input.

Variables:
Machine Parent - a reference to the parent of this machine input

Functions:
public bool ReceiveItem(Item item) - called when some output tries to put an item into this input, input passes item to parent machine and checks if it can be accepted

MachineOutput.cs

Machine output class is used by the machine to send items to another machine input.

Variables:
Machine Parent - a reference to the parent of this machine output
MachineInput ConnectedTo - machine input to which this output is connected to

Functions:
public bool TryToSendItem(Item item) - the method used to try push item through output into connected to it input

FactoryBuilderMaster.cs

Singleton object that keeps track of all machines in the current scene and updates them with set frequency is also used to call code from a different thread on unity main thread if there is such need. Here you can also find a list of all possible crafting recipes and a list of all possible items definitions (you have to fill these lists manually in the inspector changing their order will result in broken save files!). I have chosen this way because now I'm 100% sure that recipes and items that are going to be used are loaded and the template can serialize and deserialize them as indexes in these lists. There is also an experimental feature where you can run machines logic on a separate thread however it sometimes creates bugs, one thing to note while using a separate thread UpdatesPerSecond can't be higher than the ups of the game.

Variables:
public int UpdatesPerSecond - how many MachineUpdate() will be called each second, use-value lower than your framerate to gain performance. If you use high-speed conveyor belts you might want to set it to your game fps

public CraftingRecipe[] AllRecipes - the place where all recipes are stored and can be accessed during runtime used for save load system and useful for making lists of for example all possible recipes etc. Each new created Recipe has to be added here! Changing the order of recipes will break save files

public ItemDefinition[] AllItemDefinitions - the place where all item definitions are stored. This array is used in inventory serialization, every new item has to be added here! Otherwise, the save load system will not work! Changing the order of items will break save files

public int MachinesAmount - read-only indicates the number of machines in the current scene

public bool MeasureUpdateTime - if set to true then update time consumed by MachineUpdate() of all machines will be measured and value UpdateTimeInMs will be updated every second

public long MachinesUpdateTimeInMs - how much time was spend on MachineUpdate() of all machines in the last second in milliseconds. Value 0 means scripts consumed less than 1 ms during an update

public bool RunOnSeparateThread - makes most of the factory-related logic run on a separate thread to give better performance. Do not change this value during runtime! Known bug: do not set UpdatesPerSecond to be higher than the current framerate because items can start to clone, this function is experimental

Functions:
List<CraftingRecipe>GetAllPossibleRecipesForGivenCraftingMachine(Machine.MachineTyp e type, int inputsCount, int outputsCount) - returns the list of all possible recipes that can be used with a given crafting machine. Used by InventoryView.cs to get construct a list of selectable recipes when a player is trying to change the current recipe in the machine

Machine GetMachineByID(int id) - get Machine instance in the current scene by its ID, used in save load system

void SaveToFile(string saveFileName) - save scene data into text file at user persistentDataPath

string Save() - save current scene into a string containing scene data in JSON format

void LoadFromFile(string saveFileName) - load scene from text file stored at user persistentDataPath

IEnumerator Load(string sceneDataJSON, bool destroyMachinesNotInsideSavedData = true) - load scene from JSON data, it has to be IEnumerator and started as coroutine (see how LoadFromFile works) because code have to wait for 1 frame after placing prefabs in the scene to make sure Start() methods in all-new prefabs have been called

Inventory.cs

Inventory class used by objects that need storage. While this could be just a normal C# class I decided to keep this as MonoBehaviour so inventory can be added to any object just by attaching this script to the game object then getting a reference to it by using GetComponent<Inventory> in the main object class (see how container or crafting machine use it).

Variables:
public List<InventorySlot> Slots - a list containing all inventory slots of this inventory

Functions:
public void ChangeSlotsAmount(int slots) - change amount of slots inventory have

public Item GetLast() - get the last item from the last inventory slot in this inventory

public Item GetLastAndRemove() - get last item in inventory and remove it

public bool Add(Item item) - add an item to inventory, true if added successfully

public bool Remove(Item item) - remove the item from inventory, true if the item was removed

public int GetOccupiedSlots() - get occupied slots amount
string Save() - save inventory into JSON string

bool Load(string data) - load inventory from data in JSON format

InventorySlot.cs

Inventory slot from which inventory consists of. Used to store a maximum full stack of one item type.

Variables:
public List<Item> StoredItems - stored items list

Functions:
public bool CanAdd(Item item) - checks if can add an item to this slot

public bool CanAdd(ItemDefinition itemDef) - check if one item of a given item type can be added to slot

public bool Add(Item item) - add the item to a slot, true when items were added

public bool HaveSpaceForItems(Item item, int amount) - check if a given amount of items can be added to slot

public bool HaveSpaceForItems(ItemDefinition itemDef, int amount) - check if a given amount of items can be added to slot

public bool Remove(Item item) - remove the item from the slot, true when successful

public void Clear() - clears slot

public Item GetLast() - returns the last item from the slot, but does not remove it from the stack

public Item GetLastAndRemove() - get last item from slot and remove it

public bool SlotItemTypesMatches(Item item) - check if the type of item matches with what type of items is stored in this slot.

public int InStack() - how many items are in this slot

public ItemDefinition GetItemDefinition() - get item type this slot stores, null if empty

public bool IsEmpty() - check if this slot is empty

public bool IsFull() - check if slot is full

ItemDefinition.cs

Item definition/type scriptable class, here parameters of items are stored and used later when creating Item objects

Variables:
public string ItemName - the name of the item

public bool Stackable - is this item stackable

public int MaxInStack - how many items in the stack, used only when Stackable is true

public Sprite ItemIcon - sprite used in UI when rendering item preview f.e. in inventory view

public GameObject VisualItemPrefab - when using instanced rendering only mesh, material, rotation, and scale will be used to render items on the belt. Items on belts are not individual GameObjects so any scripts attached to the item prefab will not execute when the item is on the conveyor belt unless Instanced Rendering is turned off

public bool CastShadowOnBelt - does item on conveyor belt cast shadows

public bool ReceiveShadowOnBelt - does item on the conveyor belt receive shadows

public Vector3 PivotOffset - vector used to offset item position on the belt to correct its pivot

public float SpaceNeededInFrontOfItemOnBelt - the minimal distance between the pivot of this item and the next one on the belt

public bool RenderInstanced - use instancing when rendering this item on belt, please use instanced rendering when possible, using conventional game objects to render stuff on belts is very ineffective even with pooling in use, and with bigger setups, you will probably encounter lag spikes

Item.cs
Item class which parameters are defined by ItemDefinition inside it.

Variables:
public ItemDefinition ItemDefinition - scriptable object defining this item instance

public int ItemDefinitionID - index in an array stored in FactoryBuilderMaster which contains all possible items, it has to be serialized this way because unity serialization doesn't serialize references to scriptable objects

public void OnAfterDeserialize() - function from ISerializationCallbackReceiver interface responsible to restore proper ItemDefinition reference after deserialization

public void OnBeforeSerialize() - function from ISerializationCallbackReceiver interface responsible to remember index of item definition

CraftingRecipe.cs
Crafting Recipe scriptable object class used by crafting machines to craft new items.

Variables:
public Machine.MachineType Type - the type of machine where this crafting recipe can be used (f.e. recipes for smelters can't be selected in basic crafting machines)

public List<Ingredient> Ingredients - list of ingredients needed to craft CraftingResults

public List<Ingredient> CraftingResults - list of crafting results that will be produced from consuming Ingredients and waiting for CraftingTime

public float CraftingTime - time in seconds of how much time crafting one CraftingResults list consumes

public string CraftingRecipeName - the name of the crafting recipe, has to be unique!

InventoryView.cs

Example class of how to display UI for specific machines. This class doesn't have to be updated as long as UI containers this class uses remain the same (look in inspector how UI hierarchy works f.e. ingredients list is just scrollable panel and its contents are getting generated here based on currently selected recipe) so stuff like position, sprites can be changed without any code modifications.

Variables:
public static InventoryView Instance - this class is used as a singleton to be able to open inventory from any place in the code

public Transform ItemViewPrefab - prefab of item view game object that is used to visually represent inventory slot as image, name, and quantity used to construct pretty everywhere where showing Item in GUI is needed

public Transform GeneralInventoryViewPanel - parent object of all objects responsible for showing UI of basic inventory

public Transform GeneralContentPanel - panel where all items in inventory will be displayed as a set of ItemViewPrefabs

public Transform CraftingMachineViewPanel - parent object of all objects responsible for showing UI of crafting machine inventory and its current recipe

public Transform CraftingMachineIngredientsContent - panel which contains all ingredients of current crafting machine recipe

public Transform CraftingMachineResultsContent - panel which contains all results of current crafting machine recipe

public Transform CraftingMachineStoredItemsContent - panel which contains all currently stored items of crafting machine inventory

public Dropdown CraftingMachineRecipesDropdown - dropdown used to display all possible recipes for the current crafting machine

public Transform MinerViewPanel - parent object of all objects responsible for showing UI of a miner and whats it currently mining

public Transform MinerStoredItemsContent - panel which contains all currently stored items in miner internal inventory

public Transform MinerMiningItemsContent - panel which contains what miner is currently mining and how much it mines per second

Functions:
public void Show(Machine machine, Inventory inventory) - show inventory view of given machine or just inventory

public void Hide() - hide inventory UI

ItemView.cs
The Wrapper class was used to display information about InventorySlot bound to it. Useful to conveniently display inventories (see InventoryView.cs)

Variables:
public Image ItemIcon - a reference to Image component in ItemView prefab
public Text ItemName - a reference to Text component in ItemView prefab
public Text InStack - a reference to another Text component in ItemView prefab
public Image BackgroundImage - a reference to Image component in ItemView prefab
public InventorySlot Slot - a reference to InventorySlot this ItemView instance is responsible to display

Connector.cs
Machine used to connect couple conveyor belts into one.

Variables:
public MachineInput input1 - a reference to the first input in Connector prefab
public MachineInput input2 - a reference to the second input in Connector prefab
public MachineInput input3 - a reference to the third input in Connector prefab
public MachineOutput output - a reference to machine output in Connector prefab

Functions:
string Save() - save machine internal stateinto JSON string
bool Load(string data) - load machine internal state from data in JSON format

Splitter.cs

Splitter machine splits items from input evenly to connected outputs

Variables:
public MachineInput Input; - a reference to the input in Splitter prefab
public MachineOutput Output1 - a reference to machine output in Splitter prefab
public MachineOutput Output2 - a reference to machine output in Splitter prefab
public MachineOutput Output3 - a reference to machine output in Splitter prefab

Functions:
string Save() - save machine internal stateinto JSON string
bool Load(string data) - load machine internal state from data in JSON format

Container.cs

Machine used to store items in attached to its inventory

Variables:
public MachineInput input - a reference to the first input in Container prefab
public MachineOutput output - a reference to machine output in Container prefab
public Text text - a reference to text component to show occupied slots amount (can be null)
public ItemDefinition fillWith - debug option useful to create storage full of given item to test if for example crafting machine works properly, set it to null if not used

Functions:
string Save() - save machine internal state into JSON string
bool Load(string data) - load machine internal state from data in JSON format

ConveyorBelt.cs

Conveyor belt machine used to transport items between other machines

Variables:
public MachineInput Input - a reference to the first input in ConveyorBelt prefab
public MachineOutput Output - a reference to machine output in ConveyorBelt prefab
public float ItemsSpeed - the speed with what items travel on the belt

Functions:
public void SetupBelt(Vector3[] points, MachineInput input, MachineOutput output) -
The method used by PlayerTools and save load system when the conveyor belt is placed to setup its input, output, collider, and items path

string Save() - save machine internal state into JSON string
bool Load(string data) - load machine internal state from data in JSON format

CraftingMachine.cs

Crafting machine, crafts items based on supplied Crafting Recipe and attached inventory

Variables:
public List<MachineInput> CraftingInputs - references to CraftingMachine prefab MachineInputs

public List<MachineOutput> CraftingOutputs - references to CraftingMachine prefab MachineOutputs

public CraftingRecipe Recipe - recipes that crafting machine is using to craft items

public UnityEvent CraftingStarted - event used to communicate machine crafting started state without any coding

public UnityEvent CraftingEnded - event used to communicate machine crafting ended state without any coding

public UnityEvent OnCraftingProgressChange - event used to communicate crafting progress change without any coding

public float CraftingProgress - current progress of crafting process between 0 and 1

Functions:
public bool ChangeRecipe(CraftingRecipe recipe) - change the recipe, returns true if changing recipes was successful otherwise false f.e. when the supplied recipe requires more inputs than machine have

public bool ValidateRecipe(CraftingRecipe recipe) - function used to check if the recipe is suitable for this crafting machine

string Save() - save machine internal state into JSON string
bool Load(string data) - load machine internal state from data in JSON format

Miner.cs
Mining machines have to be connected to some ore to start the mining process. Miner is storing ores in attached inventory and tries to push items to its output if possible.

Variables:
public MachineOutput Output - a reference to this machine output
public Ore ConnectedOre - ore to which this miner is connected to

public bool AutomaticallyTryToConnectToNearbyOre - miner will try to find ore nearby and automatically connect to it if current Connected Ore is empty

public float MaxOreDistance - maximum distance from miner to ore, used only when AutomaticallyTryToConnectToNearbyOre is set to true.

public bool DrawRadiusSphereWhenSelected - helper showing the radius of ore searching process when Miner is selected in the inspector in Scene View

public int ItemsPerSecond - mining speed in items per second

public UnityEvent MiningStarted - event called when mining process starts;
public UnityEvent MiningStopped - event called when mining process ends to f.e. turn drill off

public float StopMiningThreshold - miner has to be idle for at least this amount of time in second to MiningStopped event can be called. Used to prevent calling stop mining events a lot.

Functions:
string Save() - save machine internal state into JSON string
bool Load(string data) - load machine internal state from data in JSON format

Ore.cs

Ore “machine” used by miners to extract ores

Variables:
public ItemDefinition OreItem - what item can be mined from this ore

public bool IsInfinite - is this ore infinite?
public int Amount - if this ore isn’t infinite then how much OreItem can be mined here

public UnityEvent OnOreEnd - event called when the ore is saturated, you can f.e. destroy it, change its material to indicate its empty

public Miner ConnectedMiner - what miner is connected to this ore, can be set in the inspector or left empty, if this field isn’t null then other miners can’t attach to this ore

Functions:
public void MineOre() - mine one piece of ore from this source if possible

public bool CanMine() - checks if anything can be mined

string Save() - save machine internal state into JSON string

bool Load(string data) - load machine internal state from data in JSON format

FPController.cs

Simple first-person controller

Controls:
WSAD to move, space to jump, hold shift to sprint, F to toggle fly mode (space - up, Q - down), ESC to toggle the cursor

PlayerTools.cs

Player tools script used to give first-person player basic options to place belts and machines in the scene

Variables:
public FPController FpController - references to first-person controller script, used to hide/show cursor

public List<PlaceablePrefab> Prefabs - list of all possible to place by player machines prefabs

public Text InfoText - a reference to text component used to inform the player what to do next (place belt, select next point, etc) can be null

public Text InfoText2 - a reference to text component used to inform what is current building mode can be null

public Transform BeltPrefab - prefab of the conveyor belt machine

public LineRenderer BeltPreviewRenderer - a reference to belt renderer used as conveyor belt preview renderer

ISaveable.cs

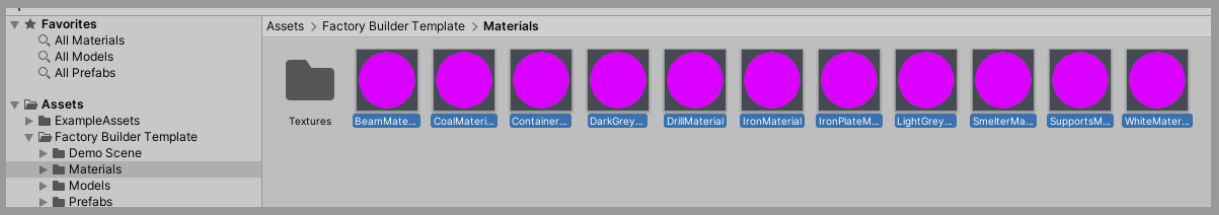
ISaveable interface used to handle object save/load operations. Saving and loading logic can be found in FactoryBuilderMaster.cs and machines classes.

Functions:
string Save() - save object data in JSON and return it as a string
bool Load(string data) - load data from string containing data in JSON format, return true when loading was successful

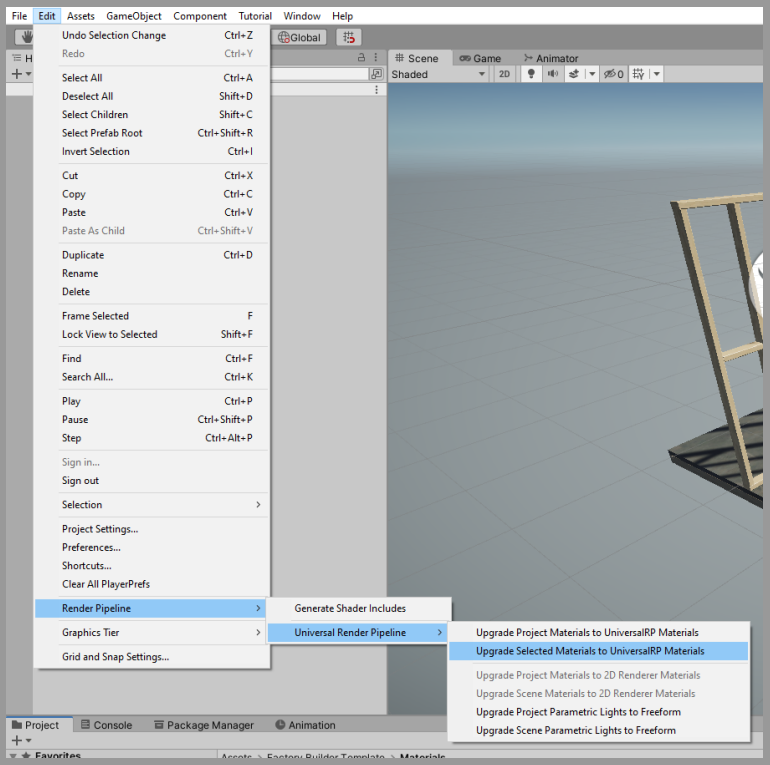
Notes and tutorials:

How to import the template into a new/existing project:

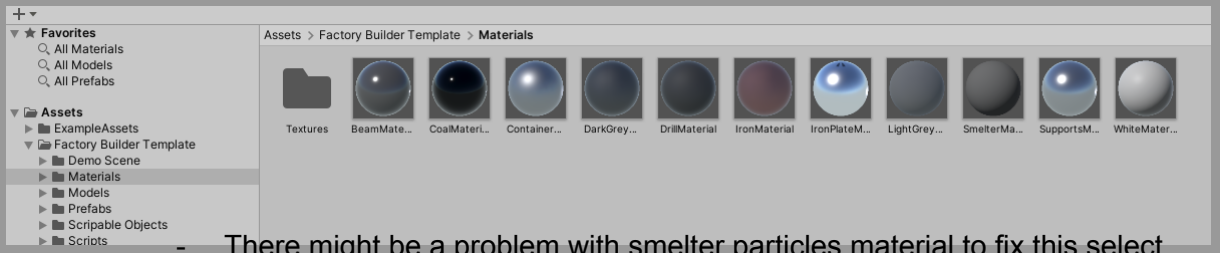
1. Download template from Unity Asset Store and import all its contents into your project
2. **If you're using URP or HDRP** you have to upgrade template materials to do so:
    - Go to Factory Builder Template/Materials and select all materials here like on the screenshot:



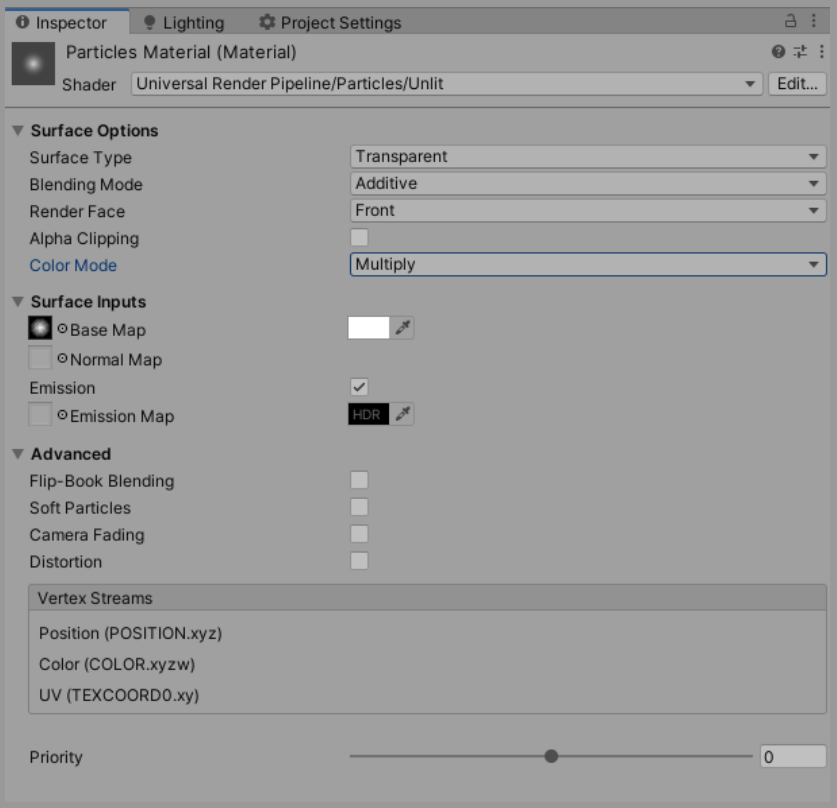
    - Then go to Edit -> Render Pipeline -> Universal Render Pipeline -> Upgrade Selected Materials To UniversalRP Materials



    - Now materials should be updated and work:



    - There might be a problem with smelter particles material to fix this select Particles Material and select options in the screenshot



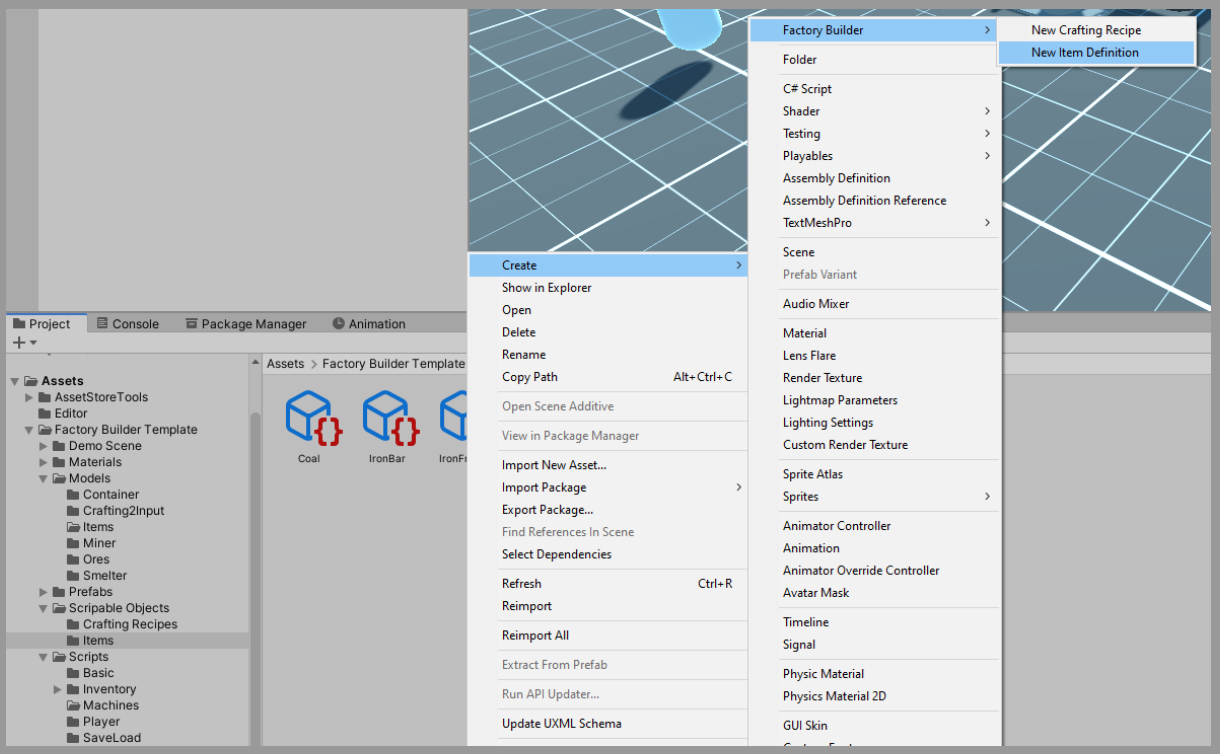
3. Now you can load up Demo Scene located at Factory Builder Template/DemoScene. My asset does not require any changes in project settings.

How to setup a new or existing scene:

1. Every scene that uses my template has to have Factory Builder Master in the scene, to add Factory Builder Master to the scene please go to Factory Builder Master/Prefabs and drag FactoryBuilderMaster into your scene.
2. If you want you can add basic player implementation and player tools that allow a player to place machines and conveyor belts, to do so put FactoryBuilderPlayer prefab into your scene.

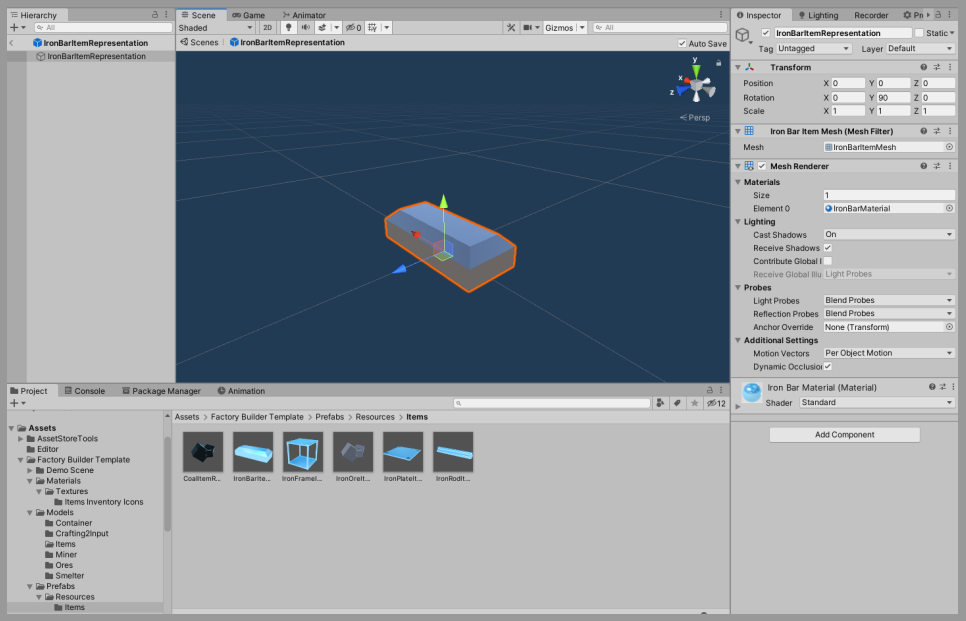
How to create a new item:

1. Go in Unity to your desired folder where you want to have your new item (I will create a new Item in Factory Builder Template/Scriptable Objects/Items)
2. Then click the right mouse button and Create -> Factory Builder -> New Item Definition (see screenshot below)



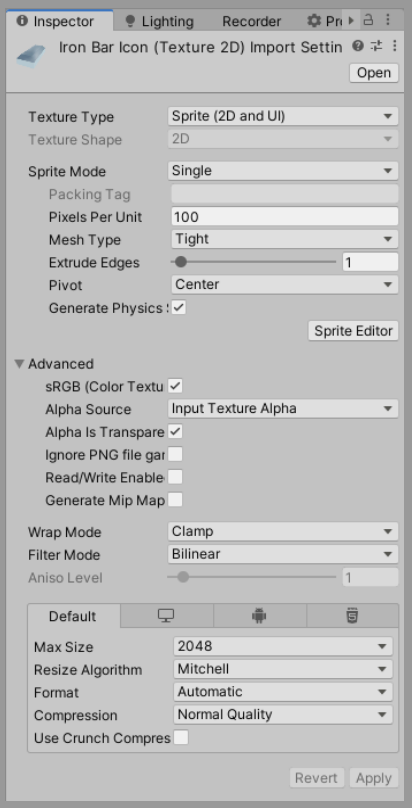
3. Name item definition file as you wish but I suggest it to be unique to have a better-organized folder

4. Before we will configure our new item we need a prefab that is used to represent our item on the conveyor belt:
    - If you want to use instanced rendering for the item (and you probably should want it because of performance reasons) its prefab has to be simple without any child objects (see f.e. how iron bar item representation prefab looks) so just prefab of a game object with mesh renderer and mesh filter without any child objects and parents etc, in the screenshot under you can see prefab ready to be used with instanced rendering:

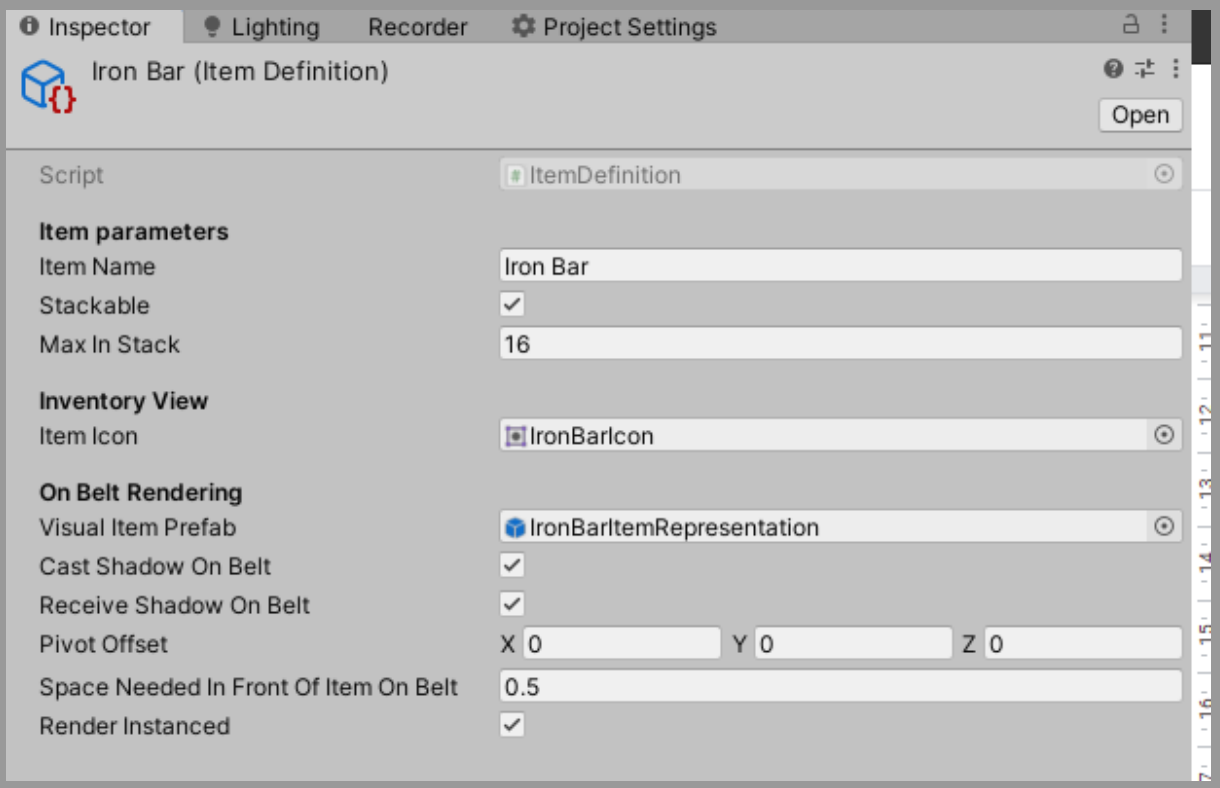


- Prefabs on the belt are rendered with prefabs pivots, you can offset their position using the Pivot Offset field in Item Definition please note don't change Pivot Offset Z component if you don't have to it can cause items poping effect
- **Your new item representation prefabs have to be located in the folder which parent is the Resources folder!** (Resources folder is a special Unity folder and everything in it will for sure be included in the Unity application build so you are sure that the prefabs you need will be in your builds, please see how I put my prefabs under the Resources folder in the template. You can have a lot of Resources folders in your project you don't have to necessarily put every new prefab in my existing folder)

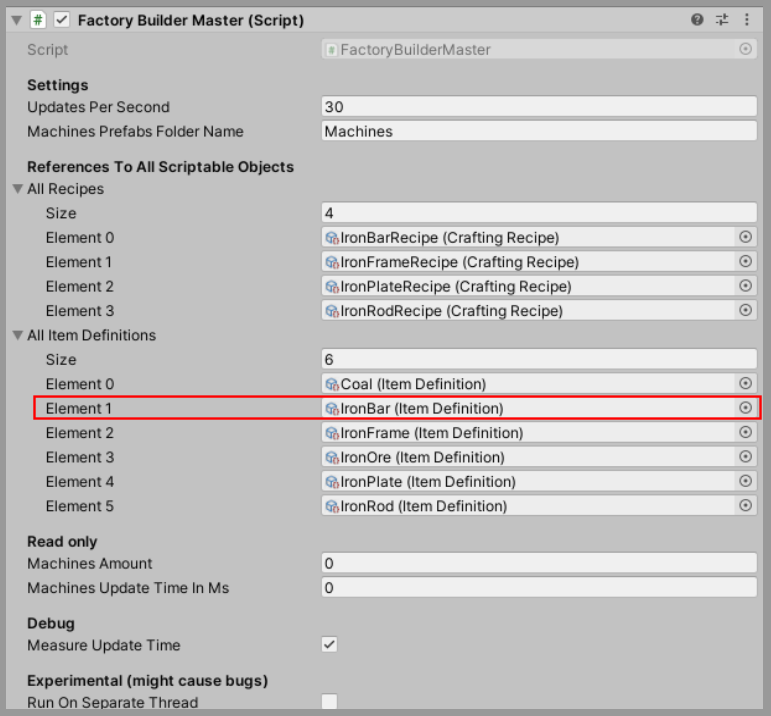
5. The last asset we need to create a new item is the Item icon in my template I use Unity renders with the size of 256x256 it's up to you what you use, one thing to remember is when importing UI icons you have to change Texture Type to Sprite (2D and UI) and probably Alpha Source as Input Texture Alpha but it depends on example icon import settings:



6. When we have all needed assets we can fill our new Item fields here you can see an example of the Iron Bar item, you also can set there other parameters like if this item is stackable how this item is rendered on the belt and how much space it needs on belt:

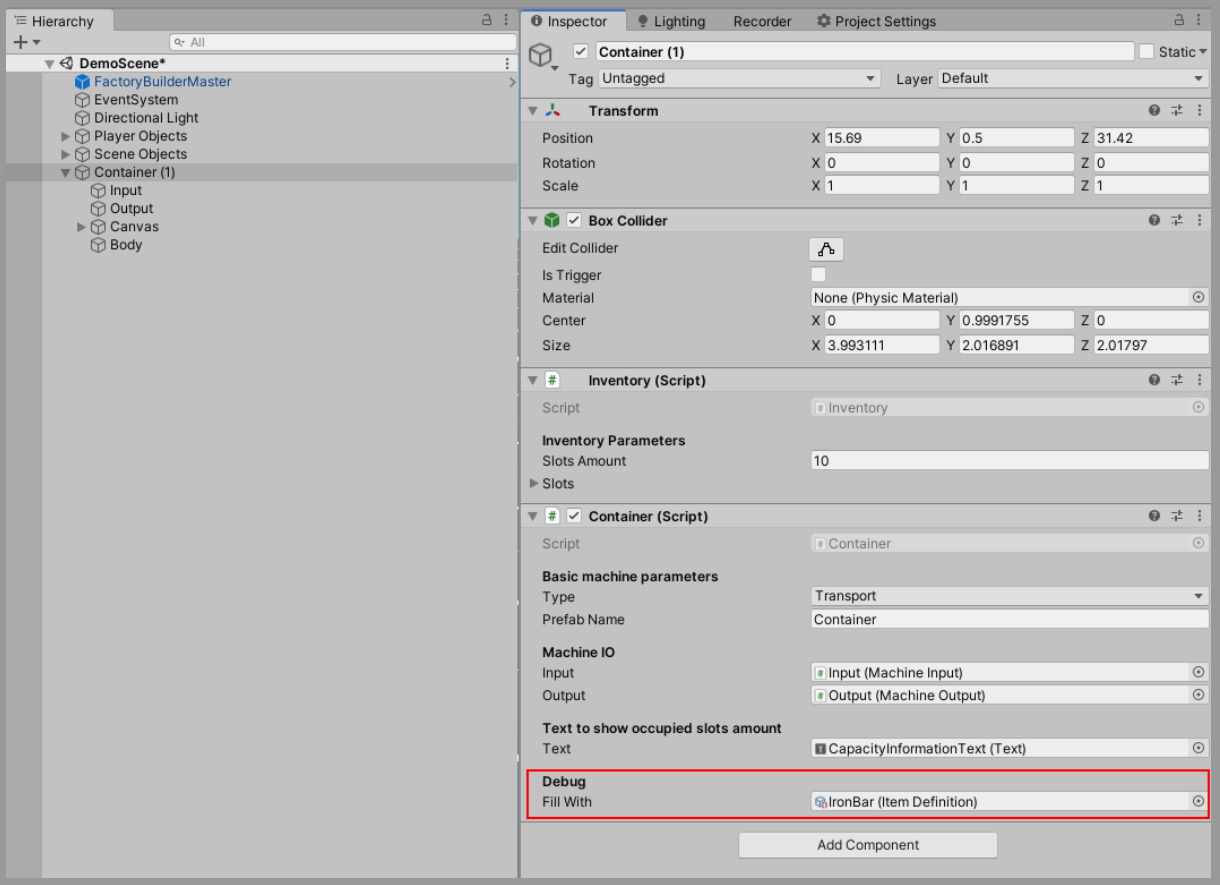


7. And finally, the last thing to do is to add our new item to the All Items list in Factory Builder Master prefab if there is no empty space left change Size from f.e. 5 to 6::



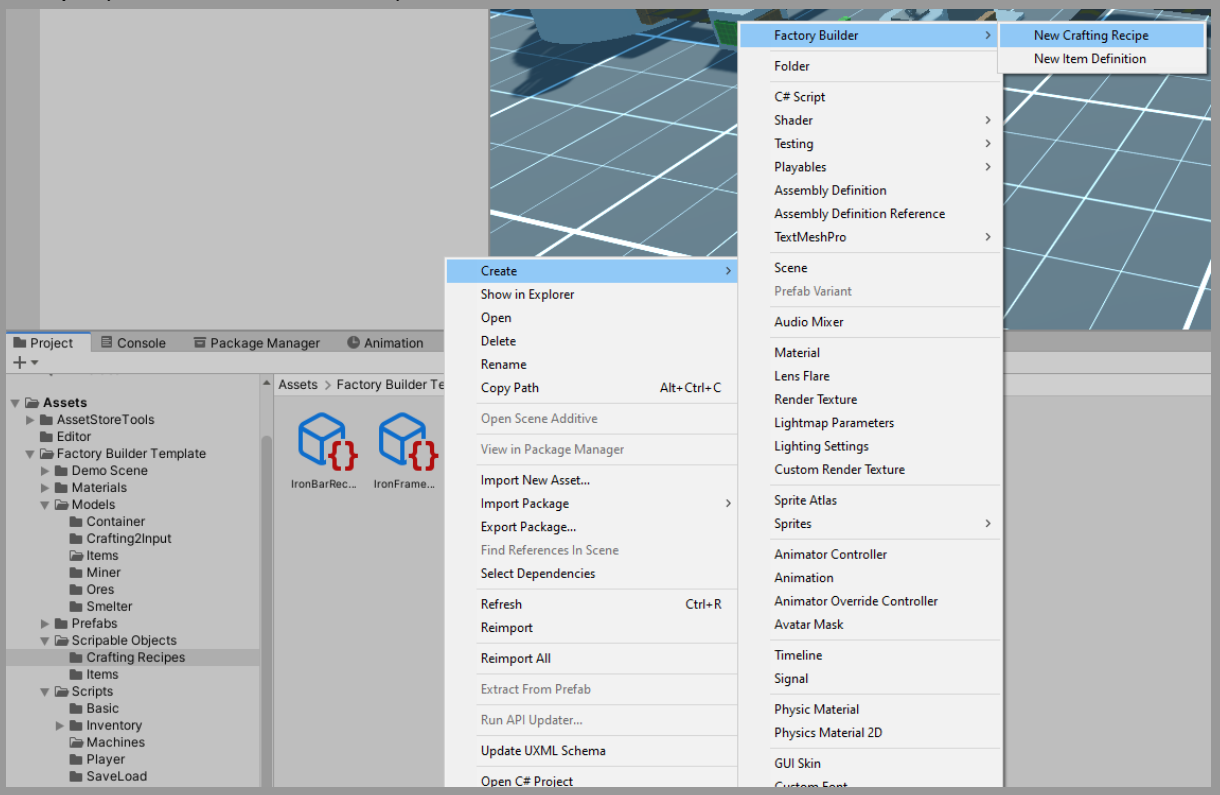
We have to do this to make the save load system work. Changing order in this list will break old save files!

8. If you want to quickly test a new item in-game world you can fill a container with it and check how it looks on conveyor belt etc. To do so put container prefab in-game world, select it, and in inspector fill Fill With field with your new item definition.

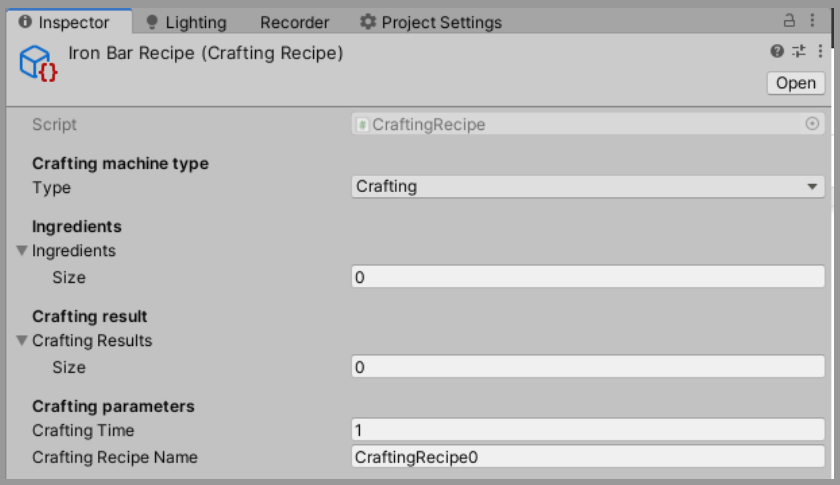


How to create a new crafting recipe:

1. Go in Unity to your desired folder where you want to have your new recipe (I will create a new Recipe in Factory Builder Template/Scriptable Objects/Crafting Recipes)
2. Then click the right mouse button and Create -> Factory Builder -> New Crafting Recipe (see screenshot below)



3. Name crafting recipe file as you wish but I suggest it to be unique to have a better-organized folder
4. Now it's time to set crafting recipe parameters to your needs. Click on the newly created recipe you will see all its parameters in inspector like in the screenshot:



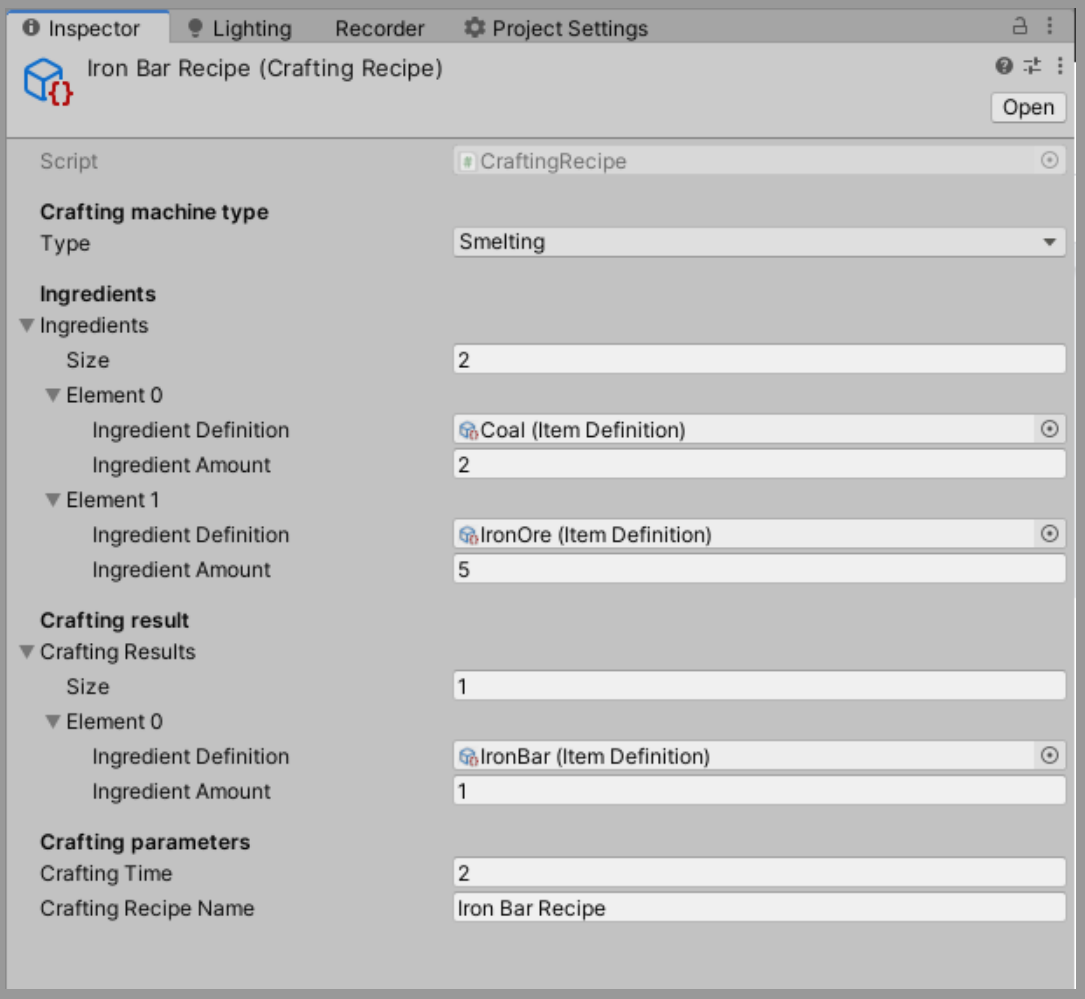
5. Let's create an Iron Bar recipe that requires 2 ingredients and outputs 1 product:
   - First, we have to decide which type of crafting machine this recipe requires because I'm creating a recipe that requires smelting I will select that if I would not select smelting type then this recipe would not appear as a selectable recipe in smelters UI.

   - Set Ingredients Size field to 2 and Crafting results Size to 1. Now Unity created empty fields which have to be filled with items definitions and how much of each ingredient we need to craft. Let's set the first ingredient to be Coal with the amount set to 2 and the second ingredient to be Iron Ore with the amount set to 5.

   - When ingredients are set we have to set Crafting Results. Because this recipe crafts iron bar I will set it as result with the amount set to 1.

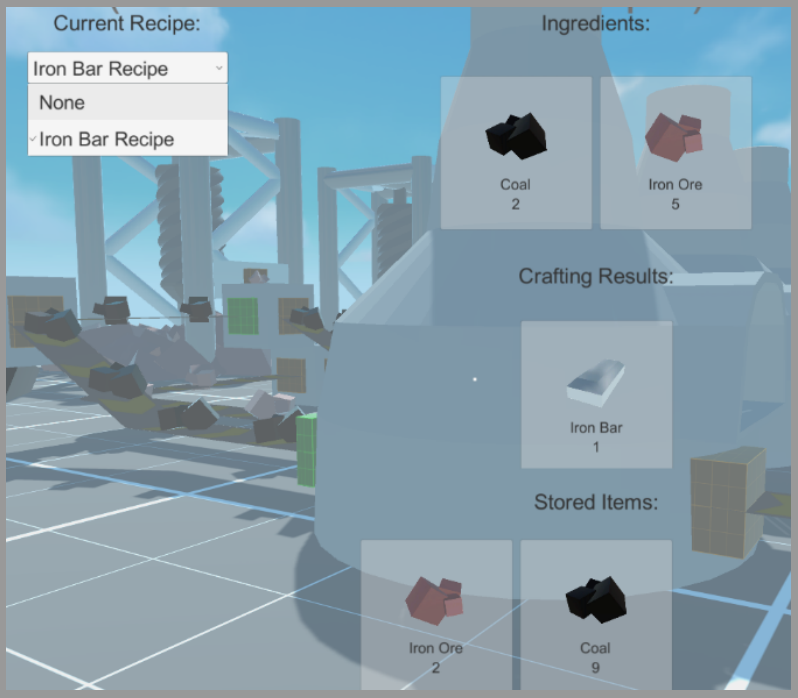
   - The recipe is almost done now we just have to set Crafting Time (in ingame seconds) and Crafting Recipe Name please remember **recipe name field has to be unique across all recipes!**

   This is how properly setup recipe looks like (you can find it in my template):



6. Note name of the scriptable object file does not have to match with the Crafting Recipe Name field like in my case

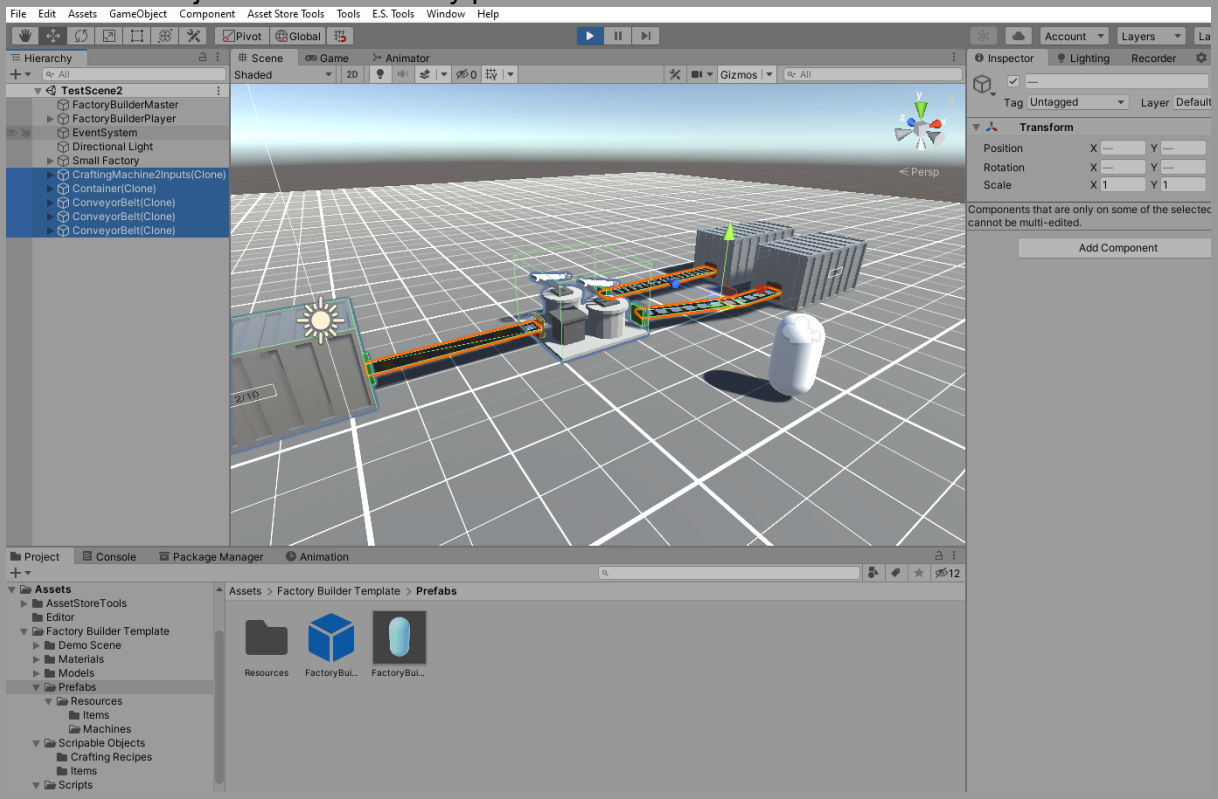
7. When the recipe is made we just have to add it to an array that contains all possible crafting recipes in FactoryBuilderMaster prefab:
    - Go to FactoryBuilderTemplate/Prefabs select FactoryBuilderMaster
    - In the inspector, you can find the All Recipes field you have to add there newly created Recipe, if there is no space left change the size from let's say 4 to 5 and add a new recipe to the last place
    - I know adding recipes there is not super handy however that was the easiest for users and the reliable way I found to be able to serialize deserialize scriptable objects references (same applies to all items definitions list) also it's handy to display all possible recipes for given crafting machine type
8. Done! Now you can hop into the game and see that the recipe is selectable in smelters:



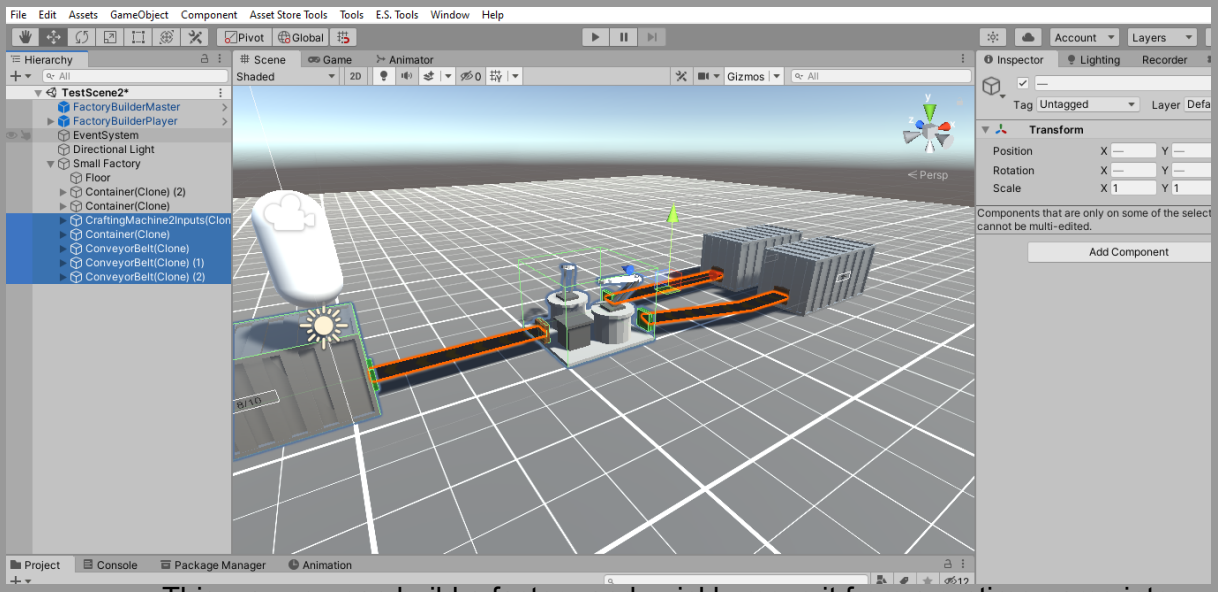
9. One thing to note after changing order (adding recipe as the last item in All Recipes is fine) in All Recipes list **will break your save game** because the index at which given recipe is stored is saved into save files and then restored based on this index, so after changing order when you will load save file machines will have different recipes set in them.

How to copy factory build in runtime into the editor

1. Officially Unity does not provide any way to do this through code.
2. My way of doing this:
    - Build the factory you want in-game mode then select all newly placed game objects in the hierarchy panel like on the screenshot:



    - Then press ctrl+c, exit play mode, and press ctrl+v in your hierarchy and move your copied objects to some parent object if you want:

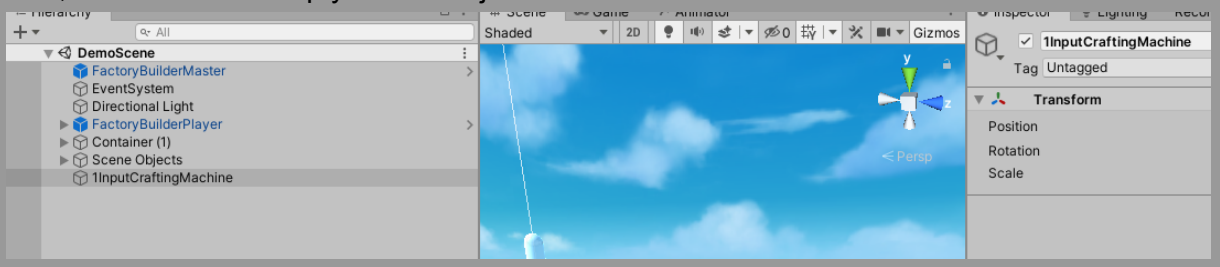


    - This way you can build a factory and quickly move it from a runtime game into an editor. Alternatively, you can just manually place all machines and connect their inputs and outputs, and set conveyor belts points.

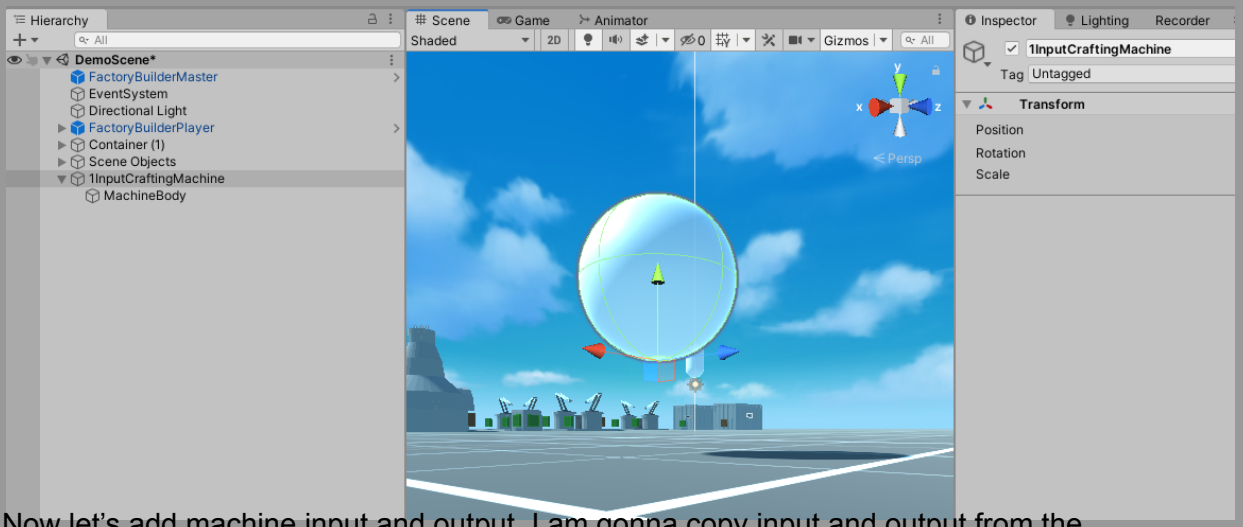
How to create a new machine with existing classes

Here I will show how to create 1 input and 1 output crafting machine with a custom model.

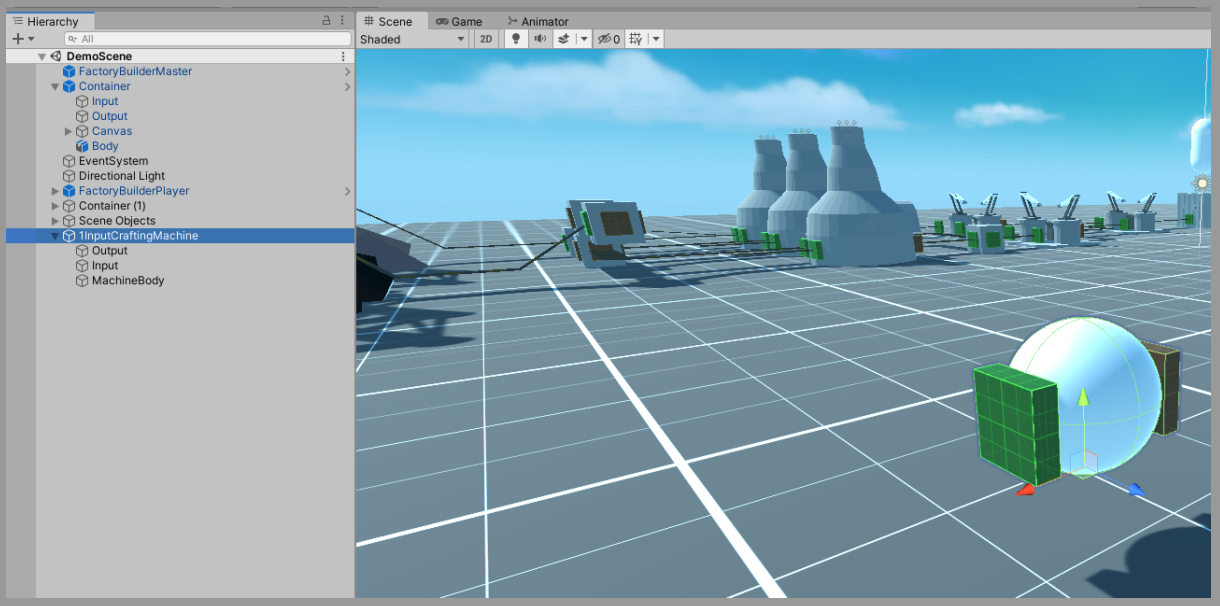
1. First, let's create an empty Game Object.



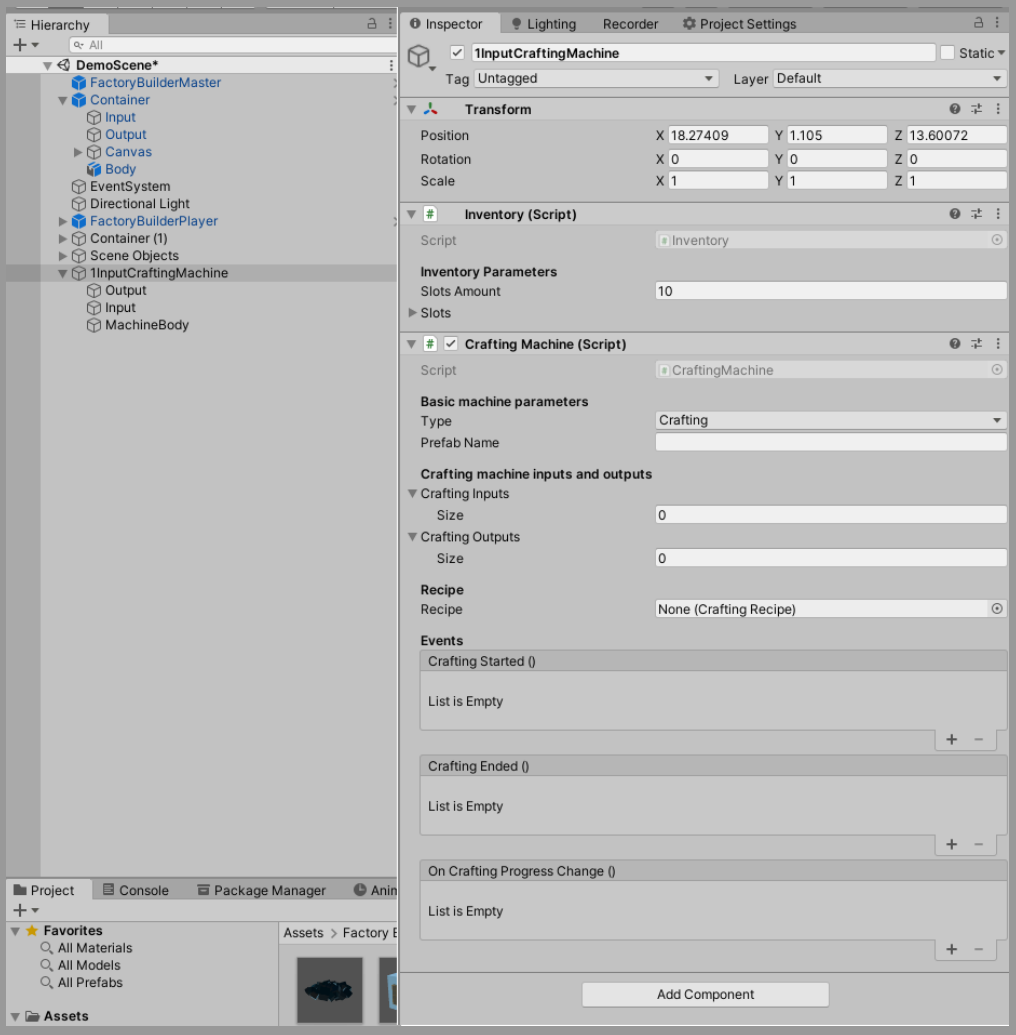
2. Add machine model as a child object to the empty Game Object that will work as machine visual body in my case that just a sphere, you probably want to shift model local position a bit to make pivot of the 1InputCraftingMachine to be the bottom center of the model



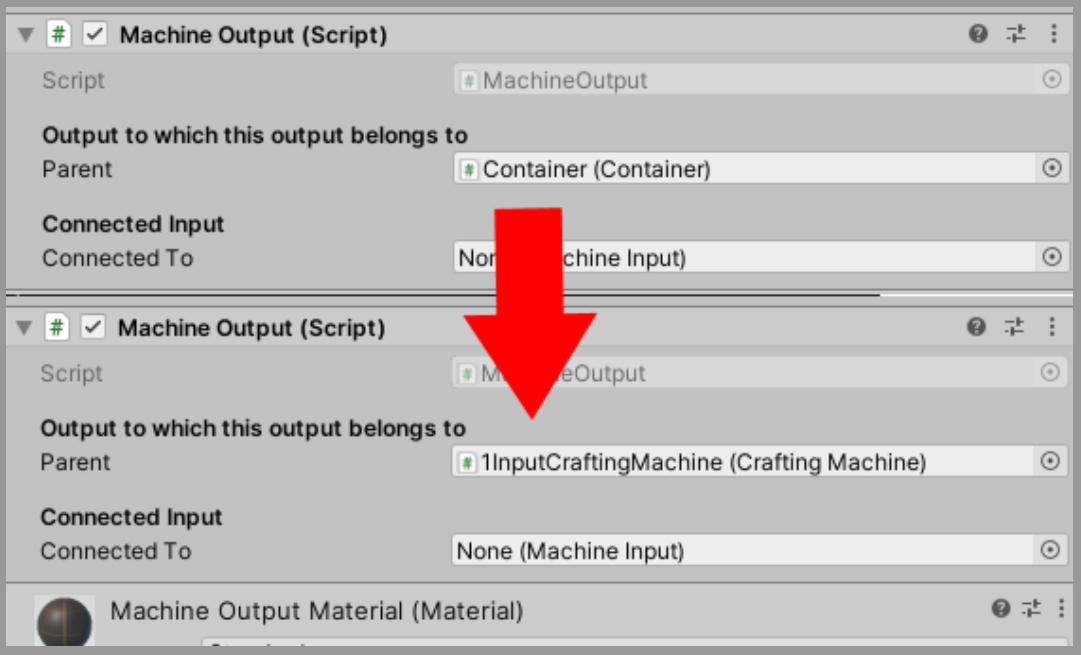
3. Now let's add machine input and output, I am gonna copy input and output from the container prefab



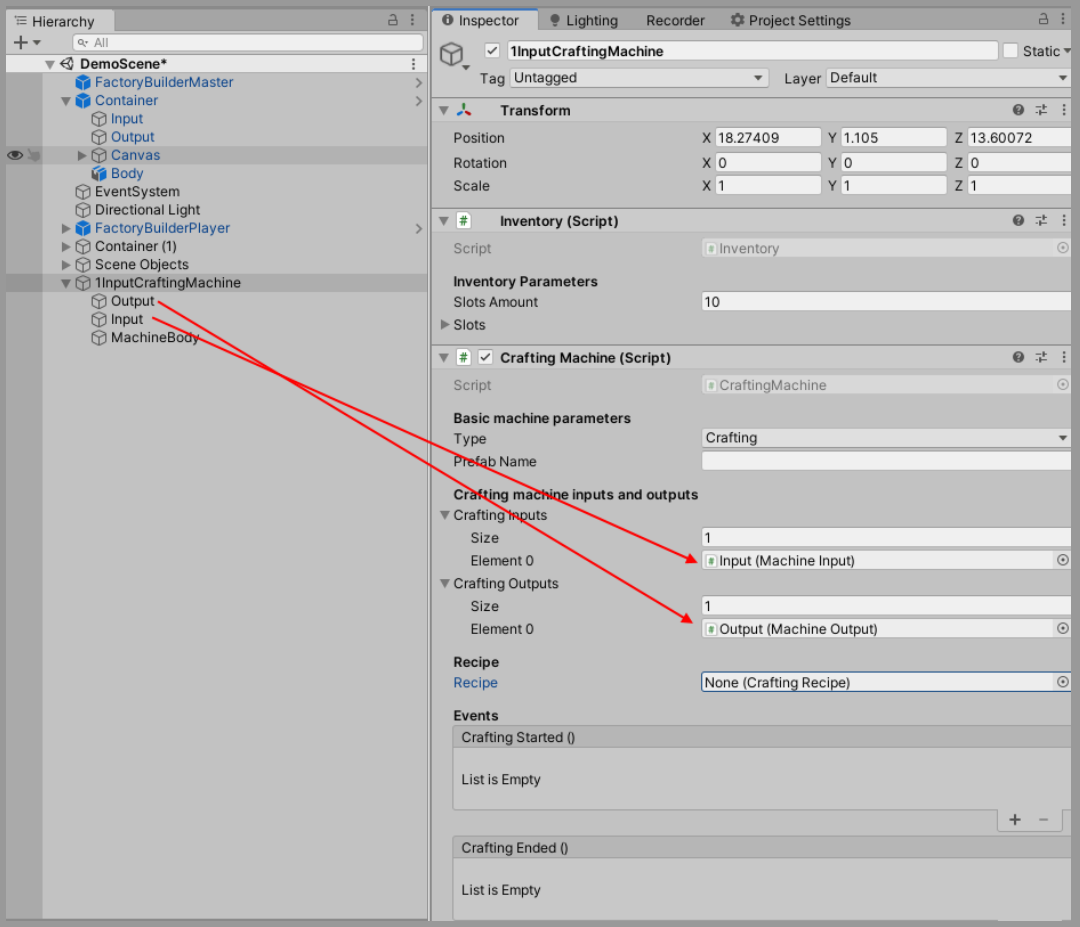
4. Now let's add CraftingMachine script to 1InputCraftingMachine:



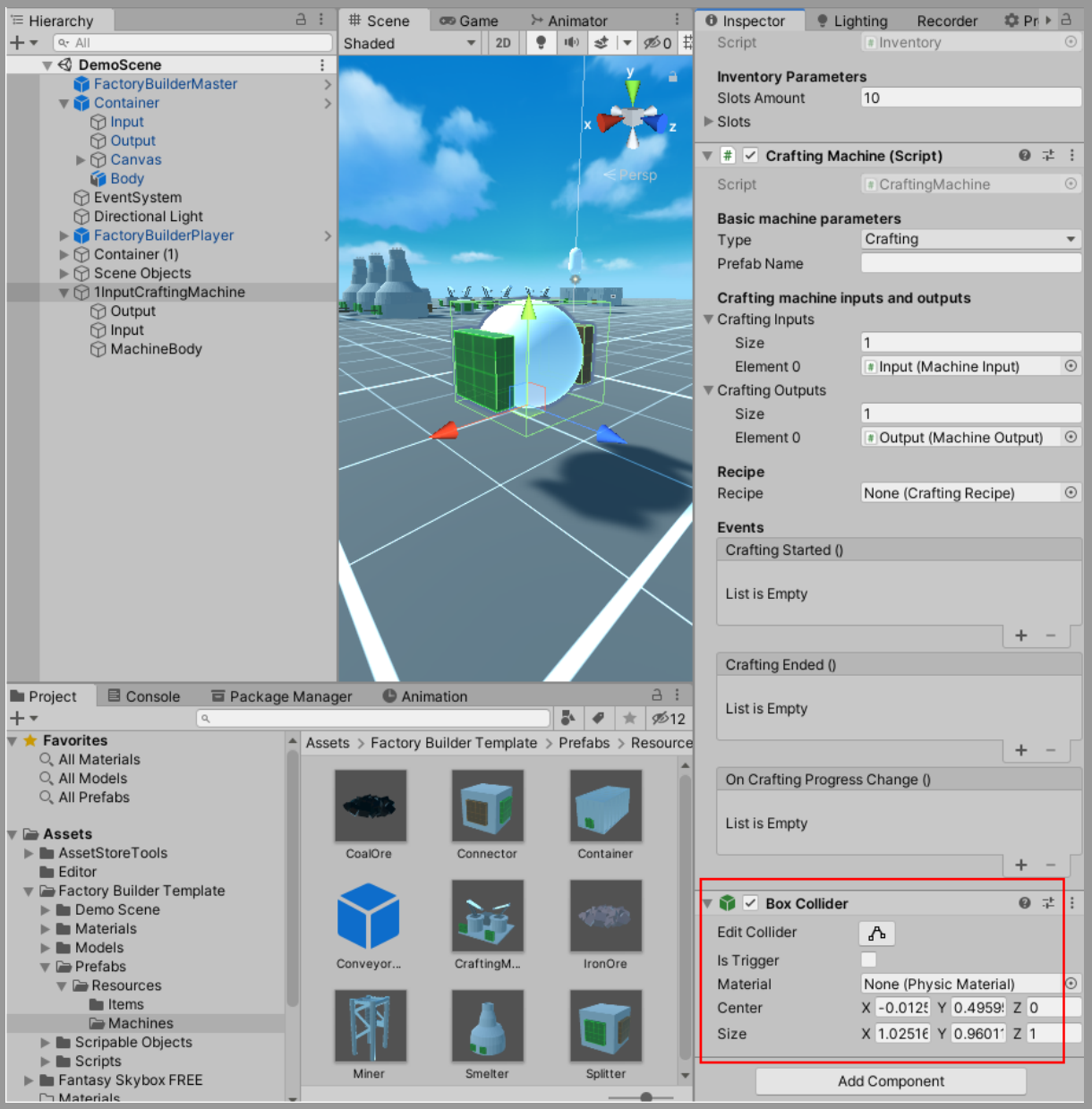
5. Just copying input and output in point no 3 is not enough we have to change their parents to do so select Output and change the Parent field. Do the same with the machine Input object:



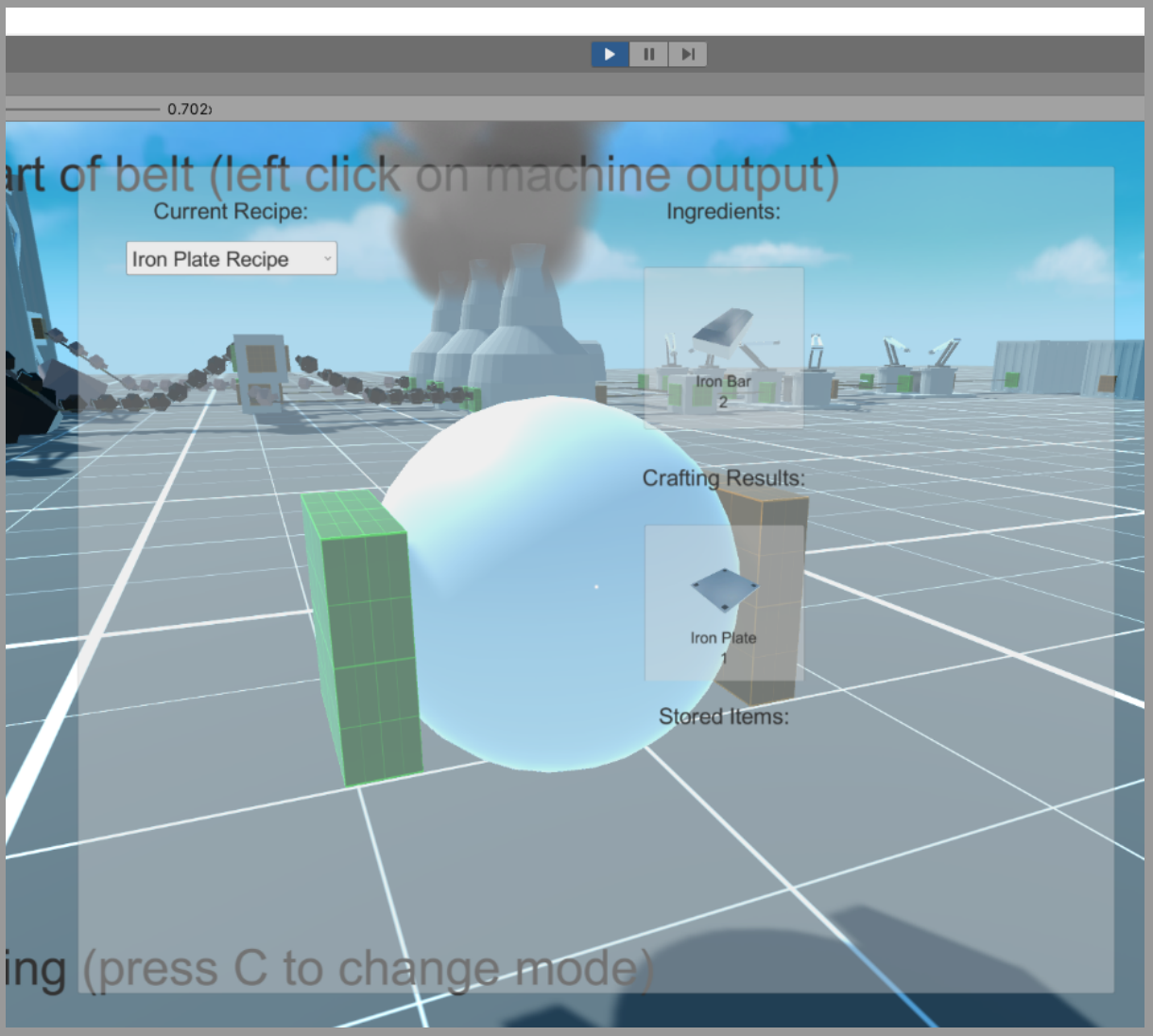
6. Now we have to register machine inputs and outputs inside the CraftingMachine script we added in point no 4:

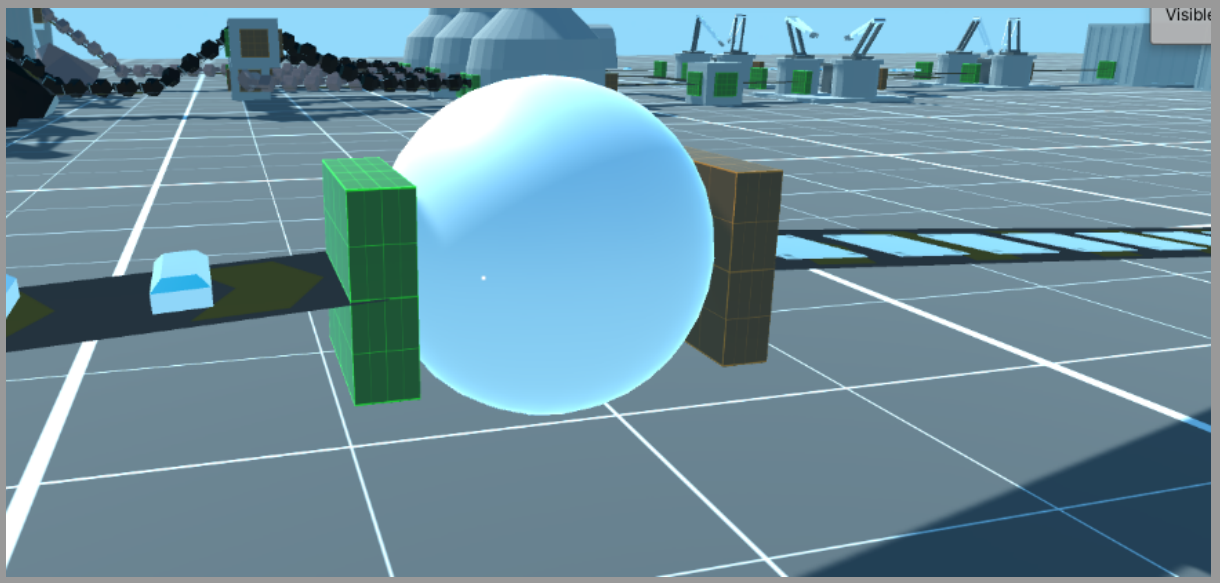


7. The only thing left to do is to add a collider to 1InputCraftingMachine you have to do this otherwise you will not be able to open close machine UI.

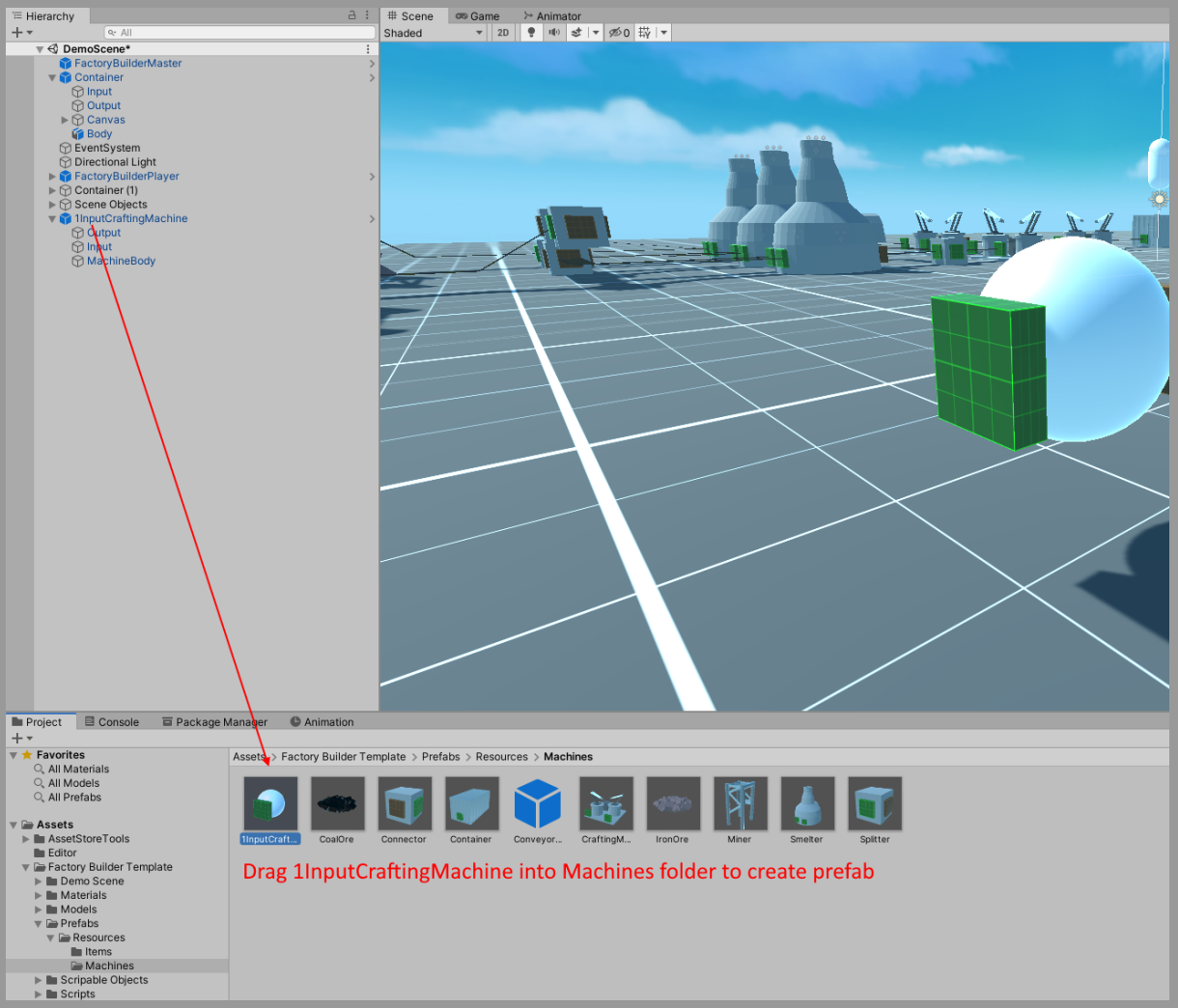


8. Now basic machine setup is completed! You can now hop into the game and start crafting with this machine

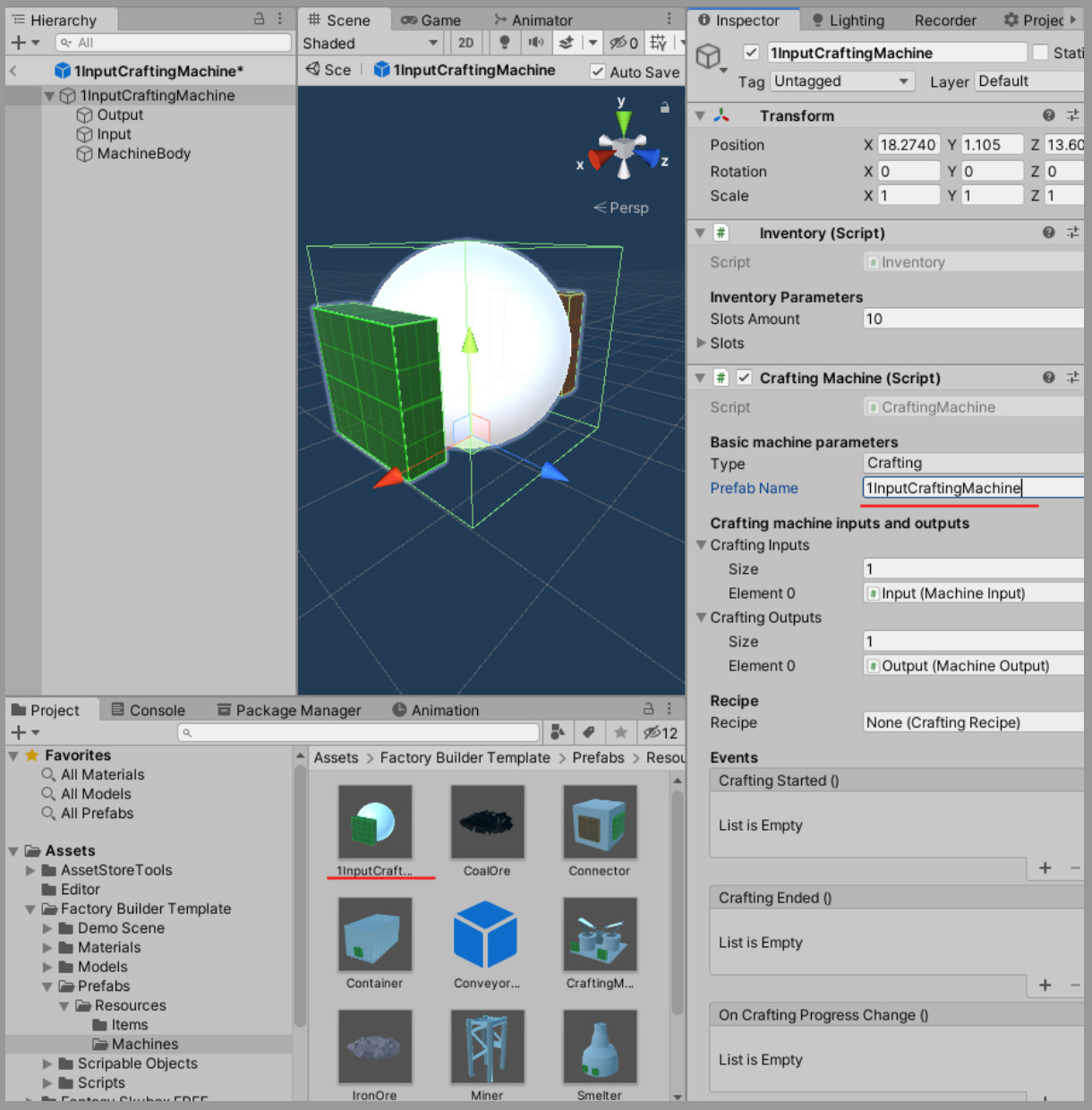




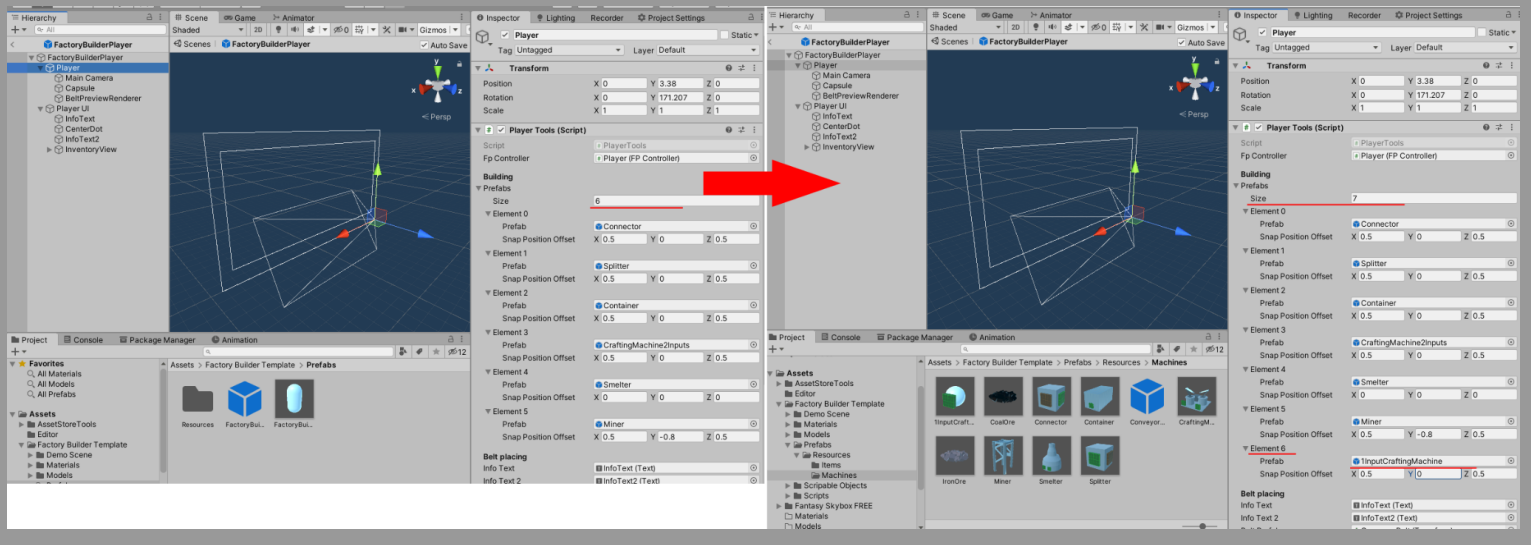
9. However, we have to complete couple more steps to allow the player to place this machine with Player Tools and make the machine work with the save load system
10. To do so we have to convert our newly created machine into a prefab to do so we have to just drag the machine from the hierarchy window into the machines folder (and we have to drag it into the **machines** folder! otherwise loading scene will fail because prefabs will not be found)



11. Now open this new prefab and change the PrefabName field to match with the name of the prefab file you have to do this to make the save load system work

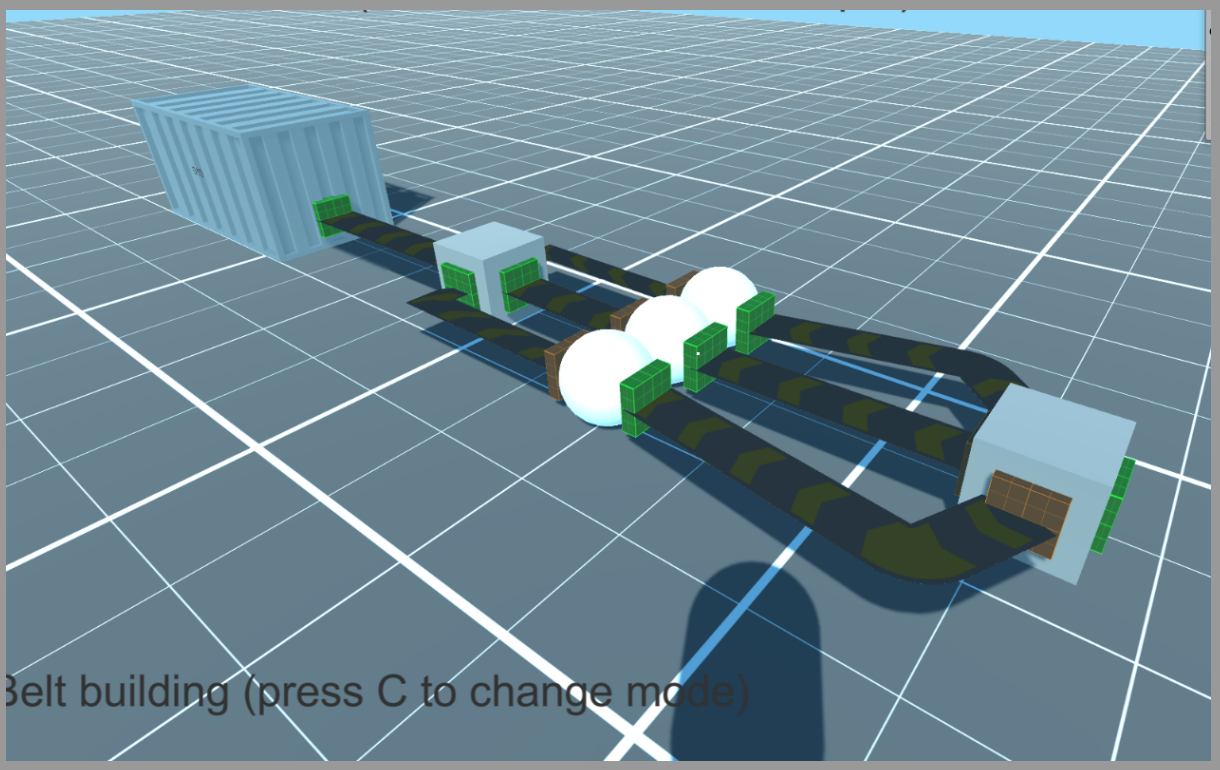


12. Now the last step is to allow the Player to place this machine runtime to do so open FactoryBuilderPlayer prefab select Player object and add a new prefab in the Prefabs list



If snapping on the new machine is not working properly you can use Snap Position Offset to adjust the snapping position.

13. The new machine is finally done and you can hop into game view place it, save it, load it!



Loading saving scene in the cloud or other places:

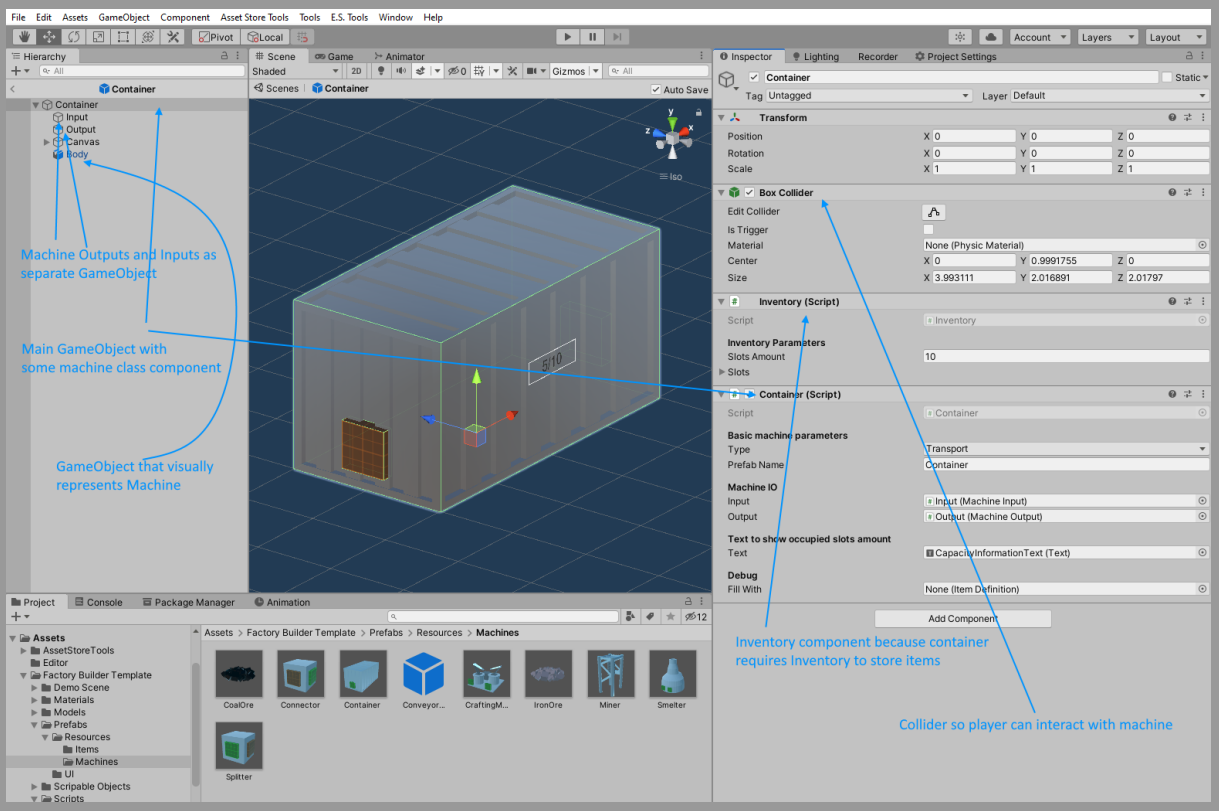
My asset natively does not support saving or loading scenes from external servers however it can be easily done because scene data is just a JSON file. So you can download data from the server and then call **FactoryBuilderMaster.Instance.StartCoroutine(Load(stringWithSceneDataInJSON));** from whatever script you want one note it has to be called on unity main thread.

How every machine is constructed in general

Every machine is constructed with the same scheme:
-Main GameObject that has some Machine class component
-If the Machine has inventory then the main GameObject has an Inventory component attached to it
-Main GameObject has a collider so the player can interact with it
-If the Machine can receive items it has MachineInputs as children of the main GameObject (every input game object name have to be locally unique! you can't have three Inputs named Input but f.e. Input1, Input2, etc otherwise save load system will break)
-If Machine can send items it has MachineOutputs as children of the main GameObject
-Child gameObject that serves as machine visuals

Then it’s saved into a prefab so it can be reused, spawned, etc

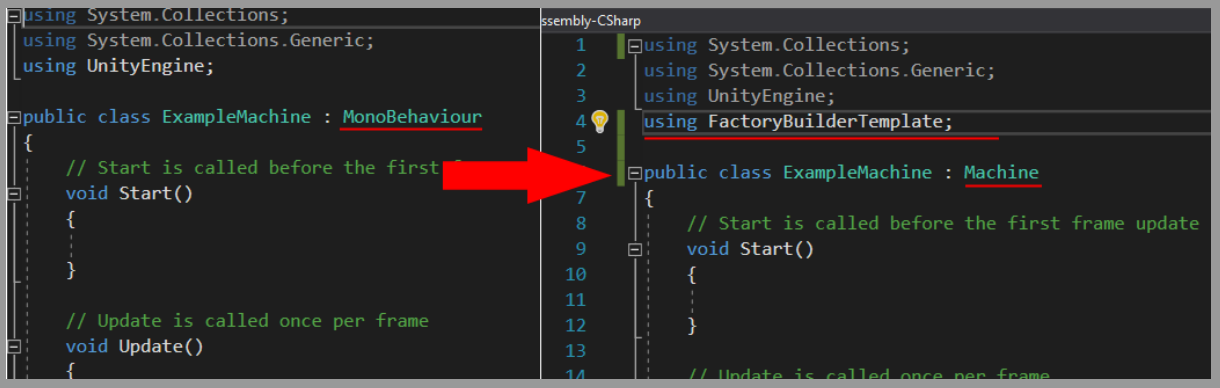
For example Container prefab:



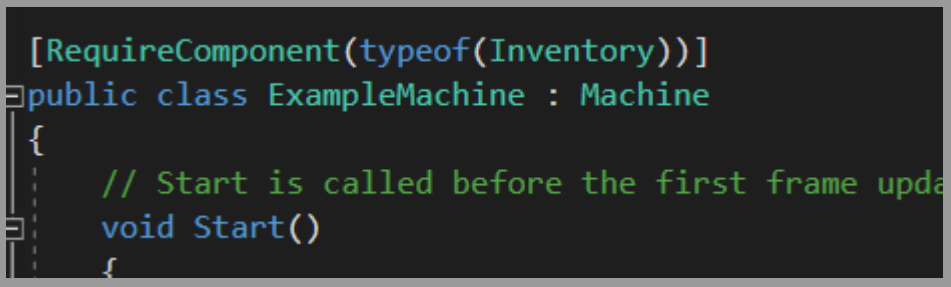
How to create custom machine script

This tutorial is for more advanced users that know how to code in C#.

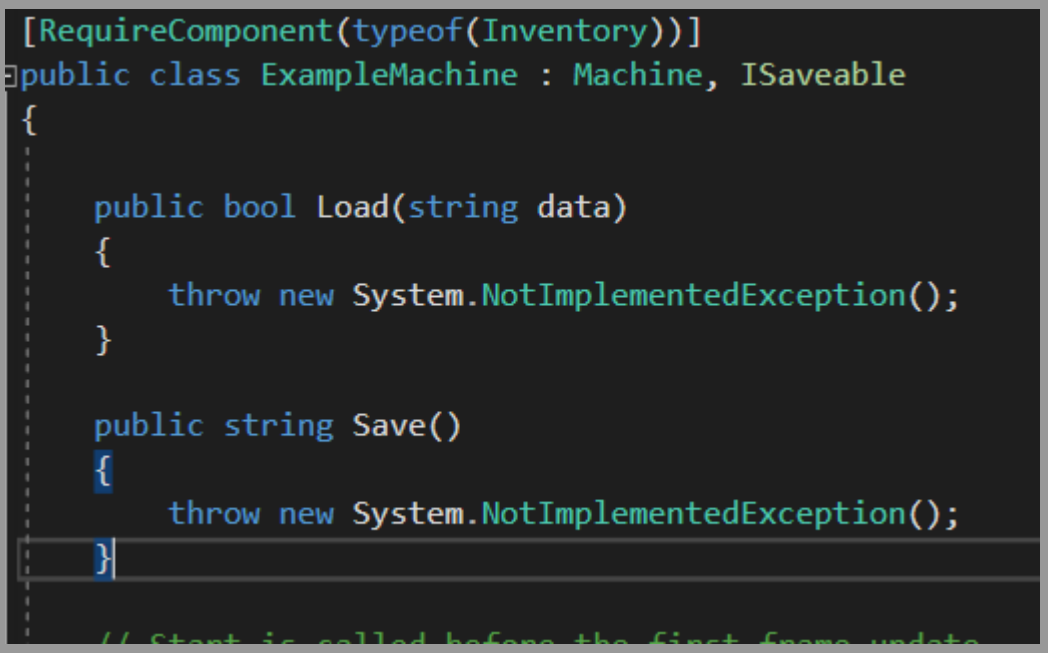
1. Create script and make it inherit from Machine class instead of MonoBehaviour (remember about “using FactoryBuilderTemplate”)



2. If you want the machine to have inventory add RequireComponent at the top



3. If you want a machine to be saveable add an ISaveable interface to this class this should create two new methods in class Load and Save



4. I will make a machine that has two Inputs and one Output so I have to declare such variables I also grabbed inventory reference in this step:

```
[RequireComponent(typeof(Inventory))]
public class ExampleMachine : Machine, ISaveable
{
    public MachineInput Input1;
    public MachineInput Input2;
    public MachineOutput Output;

    private Inventory inventory;

    // Start is called before the first frame update
    void Start()
    {
        inventory = GetComponent<Inventory>();
    }
```

   In the future, you will have to assign values for these inputs and outputs manually in Unity inspector once you will add machine inputs and outputs game objects to your main machine game object

5. Now is important part every machine have to set the IsInitializated flag to true once it is initialized and every machine has to register its inputs and outputs! Please don't forget about these steps because without IsInitializated set to true MachineUpdate() will never be called:

```
[RequireComponent(typeof(Inventory))]
public class ExampleMachine : Machine, ISaveable
{
    public MachineInput Input1;
    public MachineInput Input2;
    public MachineOutput Output;

    private Inventory inventory;

    // Start is called before the first frame updat
    void Start()
    {
        //register machine IO
        Inputs = new List<MachineInput>();
        Outputs = new List<MachineOutput>();

        Inputs.Add(Input1);
        Inputs.Add(Input2);
        Outputs.Add(Output);

        //grab internal inventory reference
        inventory = GetComponent<Inventory>();

        //set IsInitializated flag
        IsInitializated = true;
    }
```

6. Once our new machine can receive and send items it's time to check if some other machines want to send us items to do so override the ReceiveItem method and MachineUpdate so we will be able to process items in the future:

```
[RequireComponent(typeof(Inventory))]
public class ExampleMachine : Machine, ISaveable
{
    public MachineInput Input1;
    public MachineInput Input2;
    public MachineOutput Output;

    private Inventory inventory;

    // Start is called before the first frame update
    void Start()
    {
        inventory = GetComponent<Inventory>();
    }

    // Update is called once per frame
    void Update()
    {

    }

    public override bool ReceiveItem(Item item, MachineInput input)
    {
        return false;
    }

    public override void MachineUpdate()
    {

    }
```

   The value returned by ReceiveItem tells if our machine accepted the given item. For now, our machine does not accept any items.
7. It's time for machine logic I will make this machine simple and its logic will be:
   a. If there is space in internal inventory accept any incoming item
   b. If the inventory is not empty try to send items to the output
8. Let's implement this logic first let's cover the ReceiveItem method:

```
public override bool ReceiveItem(Item item, MachineInput input)
{
    return inventory.Add(item);
}
```

   Now our method returns true when an item was successfully added to the machine internal inventory (so when inventory had some space left), false otherwise.

9. Now it's time for logic inside MachineUpdate():

```
public override void MachineUpdate()
{
    //grab last stored item in internal machine inventory
    Item lastItemInInv = inventory.GetLast();
    //try to send this item to machine output
    if(lastItemInInv != null && Output.TryToSendItem(lastItemInInv))
    {
        //if machine output accepted item remove it from internal inventory
        inventory.Remove(lastItemInInv);
    }
}
```

inventory.GetLast() will return null if inventory is empty that is why there is a check for lastItemInInv being null

10. Now it's time to save and load the machine internal state
    Other machines use custom internal classes that are used only to hold machines states and this will be used here too. So first we have to create a Data class that will hold information our machine needs to restore itself after saving. In this case, it will be inventory data and where machine output is connected to. We don't have to remember where our inputs were connected because other machines remember that (our machine input is connected to some machine output which remembers its state like this machine output)
    Here is our data class:

```
[RequireComponent(typeof(Inventory))]
public class ExampleMachine : Machine, ISaveable
{
    [System.Serializable]
    private class ExampleMachineData
    {
        public string InventoryData;

        public int OutputConnectedToMachineInputID;
        public string OutputConnectedToInputGameObjectName;
    }

    public MachineInput Input1;
    public MachineInput Input2;
    public MachineOutput Output;

    private Inventory inventory;

    // Start is called before the first frame update
    void Start()
    {
```

There is one really important thing Unity JSON can only serialize basic data that's why our data class consists of strings and ints. This data alone is meaningless but we know what these values mean and we have to interpret them. We store our machine output by remembering to which machineID it is connected and the name of the connected machine input game object name to handle machines with multiple inputs like crafting machines. That's why Machine Input game objects have to have unique names in local space.

11. So let's first save our data into JSON

```
public string Save()
{
    ExampleMachineData data = new ExampleMachineData();

    //save internal inventory into JSON string
    data.InventoryData = inventory.Save();

    //save where output is connected if it is connected somewhere
    if(Outputs[0].ConnectedTo != null)
    {
        data.OutputConnectedToMachineInputID = Outputs[0].ConnectedTo.Parent.GetMachineID();
        data.OutputConnectedToInputGameObjectName = Outputs[0].ConnectedTo.name;
    }

    //parse data into JSON and return
    return JsonUtility.ToJson(data);
}
```

First, we are saving internal inventory into JSON then saving data about machine output connection, and finally returning data in JSON format

12. Now its time to load data from JSON string

```
public bool Load(string data)
{
    ExampleMachineData containerData = JsonUtility.FromJson<ExampleMachineData>(data);

    //load inventory from restored data
    inventory.Load(containerData.InventoryData);

    //connect machine output using special function that searches for possible input by giving machineID and input name
    Output.ConnectedTo = TryToFindMachineInput(containerData.OutputConnectedToMachineInputID, containerData.OutputConnectedToInputGameObjectName);

    //nothing went wrong return true as loading was successful
    return true;
}
```

First, parse JSON data into the data class we created before then load inventory then connect output. As you can see there are existing methods making life easier like methods dedicated to search for machine input based on machineID and input name. Loading inventory is also easy just one line of code.

13. Our machine class is done now you can create a machine prefab and use it in the game world (see How to create a new machine with existing classes just use this class instead of CraftingMachine).

In ExampleMachineData you can put as much data as you want as long it is serializable by Unity JSON handler if it's not you have to find some workaround f.e. ItemDefinitions references are not serializable that's why I am using a global array with all of them and then when the game is saved I am just storing the index in this global array to save inventories that's why changing AllItems list in FactoryBuilderMaster breaks old save games.

14. If you modify Machine Awake() or OnDestroy() function be sure to call its base implementation (f.e. base Awake() etc) because in Awake every machine is registered in the global array and OnDestroy() is removed from it

How to execute code from other thread on Unity main thread

You can use this code snipped in any script you want:

```
FactoryBuilderMaster.RunOnMainThread.Enqueue(() => {
    //your code
});
```

I mostly use it to invoke Unity Events when the Run On Separate Thread option is enabled.

Thank you for buying my asset!

If you have any problems, found any bugs, have some questions or suggestions please let me know at: ***piotrtplaystore@gmail.com***