

# **Visual Item Database**

**Version 3.1**

**[Changelog](#)**

**Asset by: bloeys**

# **Content**

1. [Thanks](#)
2. [Introduction](#)
3. [The General Workflow of the Visual Item Database](#)
4. [For Designers](#)
  - a. [Creating and Opening the Database](#)
  - b. [Creating and Updating Prefabs](#)
  - c. [Adding an Item to an Existing GameObject](#)
  - d. [The Item Type Control Window](#)
  - e. [Item Subtypes](#)
  - f. [Item Type Groups](#)
  - g. [Duplicating and Deleting Items](#)
  - h. [Searching for Items](#)
  - i. [Showing an Item's Icon](#)
5. [For Programmers](#)
  - a. [A Very Important Note](#)
  - b. [The General Layout of the Item System](#)
  - c. [Changing the Look of Items in the Database](#)
  - d. [Creating Instances of Items at Runtime](#)
  - e. [The ItemDatabaseEditorWindowUser Class](#)
  - f. [Dealing With Item Subtypes](#)
  - g. [Item Names Enums](#)
  - h. [Using a Different Field to Display Icons](#)
  - i. [Changing Where Assets are Generated](#)
  - j. [Transferring Items from VID 3.1 to VID 3.2](#)
  - k. [Adding New Properties to Items](#)
6. [Useful Bits of Information](#)

## 7. Contact Information

# **Thanks**

Before we get into business, I would like to sincerely **thank you** for buying this asset and supporting me, it really means a lot.

If you like the asset then please post a review on the asset store, it really helps!

I hope you find it useful and easy to use.

# **Introduction**

In this document, I will cover (hopefully) everything you will need to know to use this asset to its fullest and will try to make it as easy to understand as possible.

The documentation will be separated into two main parts, one aimed mainly for designers and non-coders who will be interacting with the database and another for the programmers who might need to know how it works and modify it to better suite their needs.

Now even though I will try to cover everything, there might still be some things that you might want to know or ask about, in which case please feel free to contact me through any of the means provided in the contact section and I will try to help the best I can.

# **The General Workflow of the Visual Item Database**

In this section I will explain how the Visual Item Database (VID) was designed, and how it is supposed to be used. This is to help you get a sense as to how to use it correctly, and to clear any presumptions in case you are coming from a different tool (Game Data editor or Datablocks, for example).

The VID is in a sense a Unity data-managing tool. It was made to help developers manage their in game items in one central place, and not have to jump back and forth between different places and/or programs (e.g. Unity and Excel).

When using the VID, items should ideally only be edited in the database (editor window), and not in external places like the inspector. Don't be afraid of making many items, even if they have just small differences between them, as the VID was designed to handle very large item counts.

Another important thing about the VID is the fact that you **do not make assets of items** (like in ‘Datablocks’ for example); instead, items are retrieved for you from item lists by certain functions.

**At edit time** items can be added as components (‘ItemContainer’ Component), these components contain a variable that contains the item itself. The VID has buttons (in the interface) that do this for you automatically.

**At runtime**, items can be retrieved by using one of a number of methods provided to the programmer. These are discussed in more detail in other sections of the docs.

Though you have the ability to create prefabs of your items, if you so desire, it is **not necessary**.

This is how items are supposed to be dealt with in the VID. It might sound a bit weird if you are coming from another tool, but it is very easy, especially when you start using the tool, as the interface is very clear and self-explanatory.

With that being said, I do recommend that you carefully read the documentation to get the best experience possible.

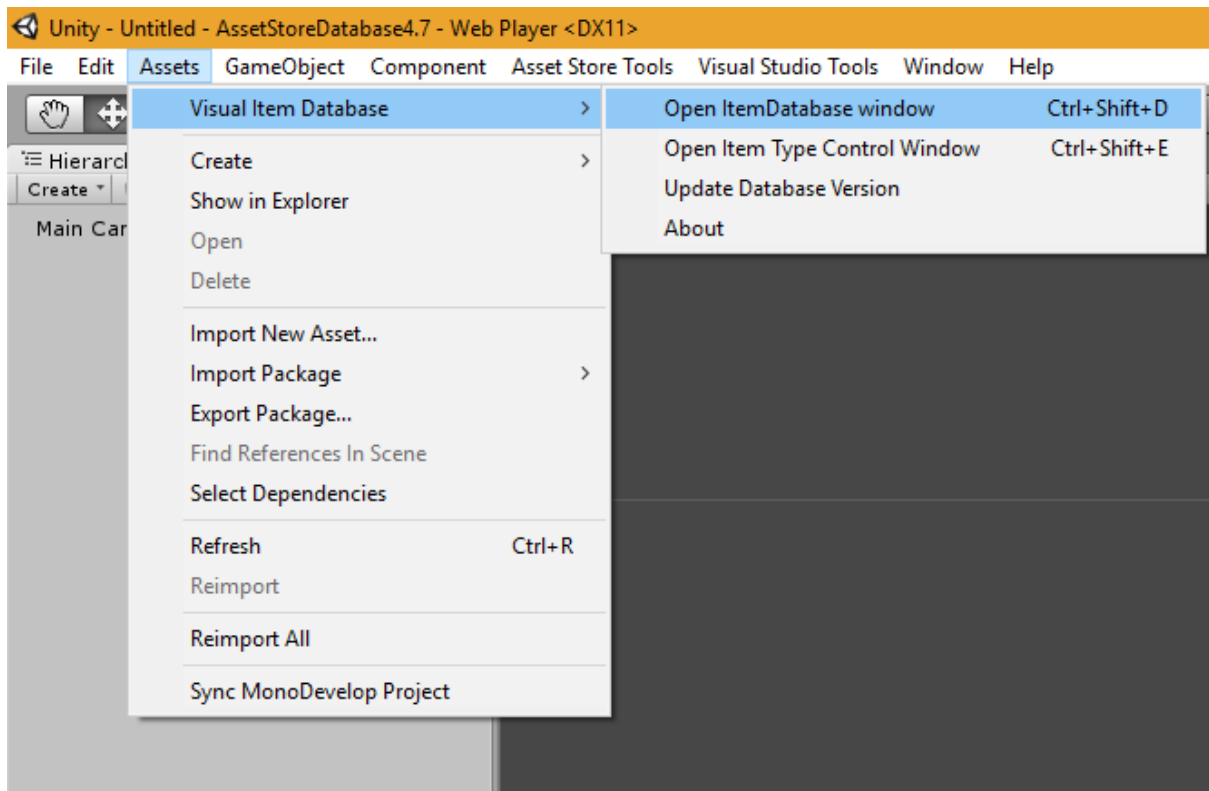
# **For Designers**



# Creating and Opening the Database

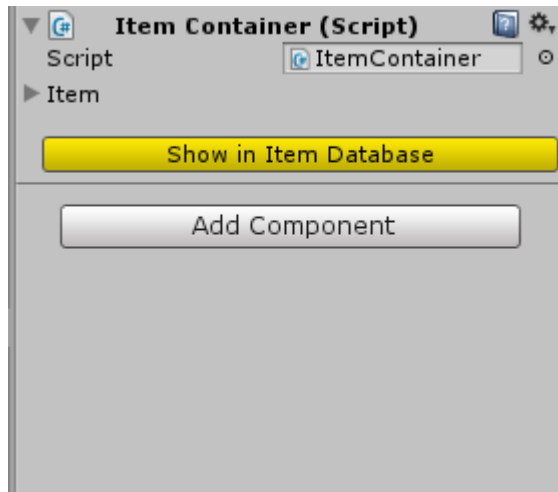
Clicking on 'Open ItemDatabase window' will create the Database if it does not exist and then open it.

This menu item is located under Assets/Visual Item Database/



The Item Database can be opened in three main ways, which we will go over one by one.

- 1) By clicking on the 'Open ItemDatabase window' menu item.
- 2) By clicking on 'Open in Item Database' in the inspector window of an item object/prefab, which will jump to the items page and select the item



- 3) By using the shortcut 'Ctrl+Shift+D' on Windows or 'Cmd+Shift+D' on Mac.

# Creating and Updating Prefabs

An important note before going into the prefab business, is that you absolutely do **not** need to create prefabs of every item to be able to use them in game. In fact, it is **not** the recommended way to use the VID.

All items can be created at runtime by writing some code without actually having a prefab of it to instantiate, though this is related to code land and therefore we will not go into it here.

Once you have created an item in the database that you want to make a prefab of, you can simply click the ‘Update Prefab’. This button will create a prefab of the item if the prefab does not exist, and it will update the prefab with the new values if a prefab does exist.

Though do keep in mind that **prefabs are linked to items through name**, so if the name of an item changes in the item database, the prefab with the old name will not be updated and instead a new prefab will be created.

If you would like to update all the items of the current type, then you can use the ‘Update Type Prefabs’ button



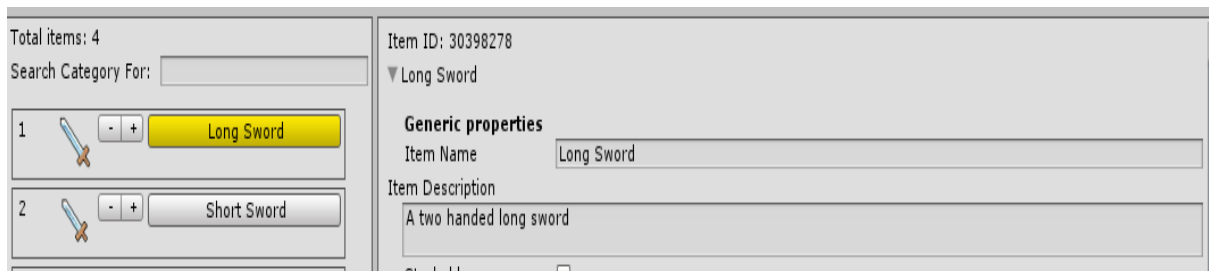
This button will create/update prefabs for all the items in the current item type. Though this is not advised if you have items that you do not want to create prefabs of.

# Adding an Item to an Existing GameObject

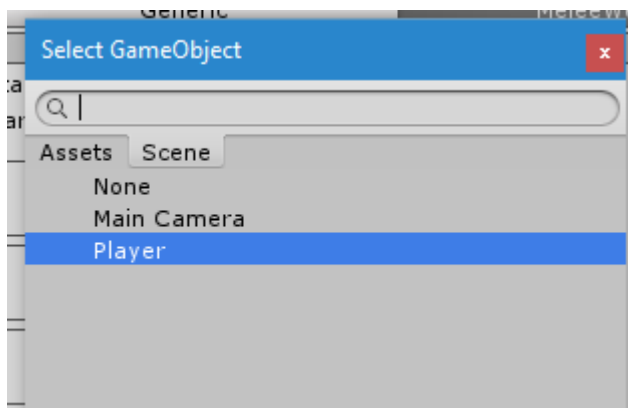
If you have an existing game object that you would like to add an item to without having to use a prefab, you can use the 'Update Object' button.

First open the database and select the item you want to add, then select the game object that you want to add the item to and click 'Update Object'.

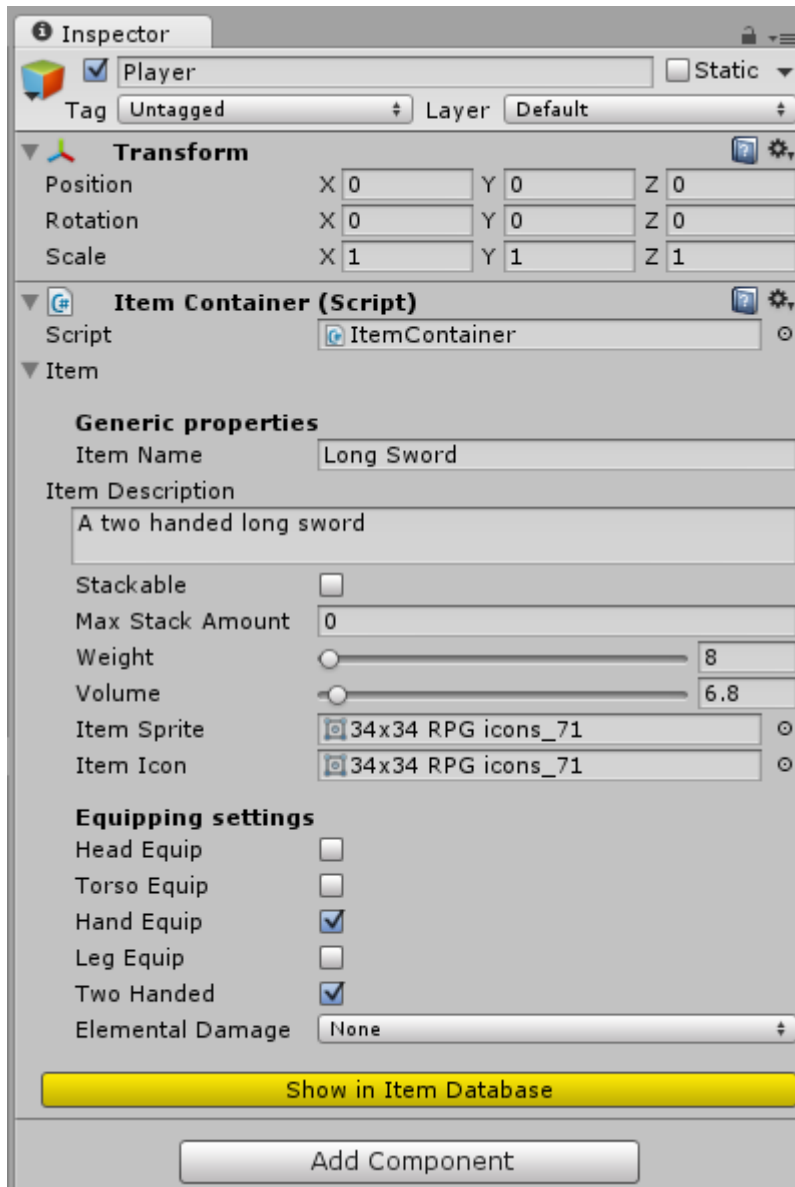
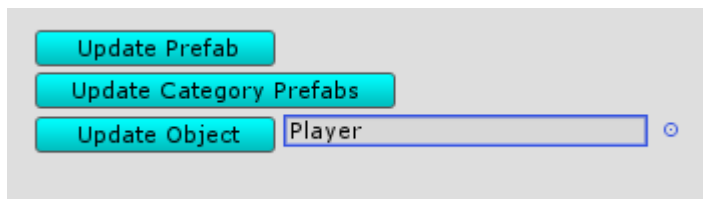
## 1. Select Item



## 2. Select Game Object

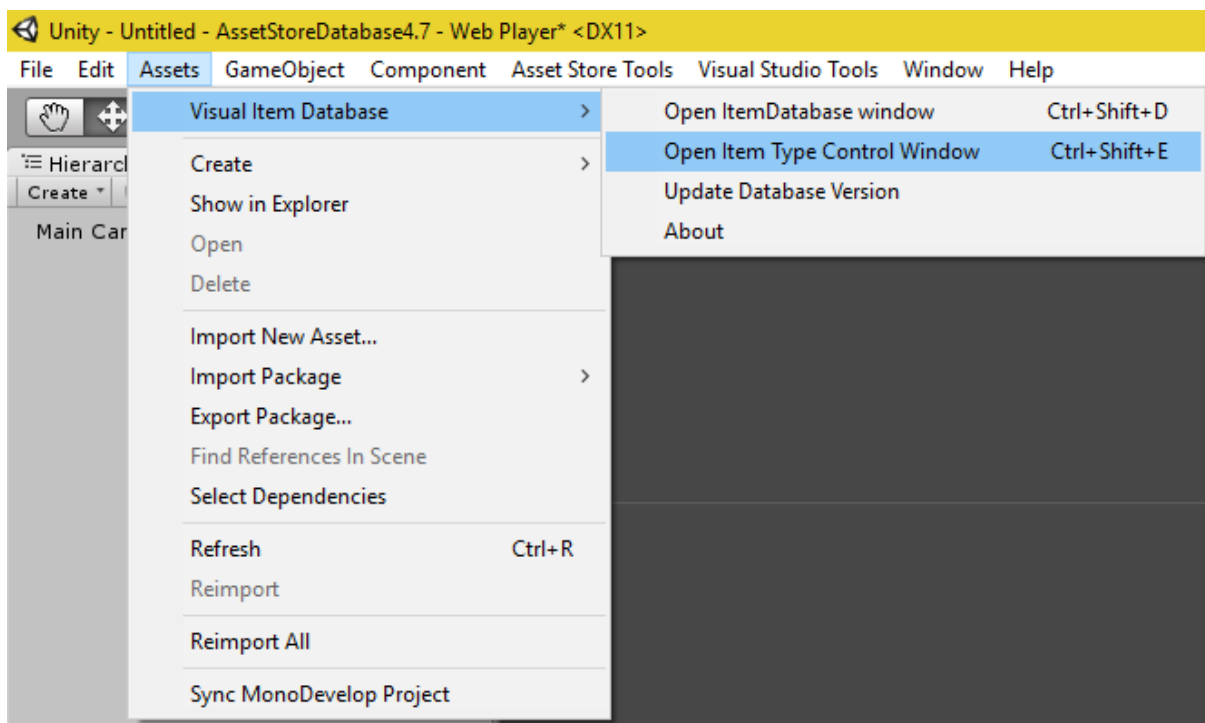


3. Press 'Update Object' and you are done.



# The Item Type Control Window

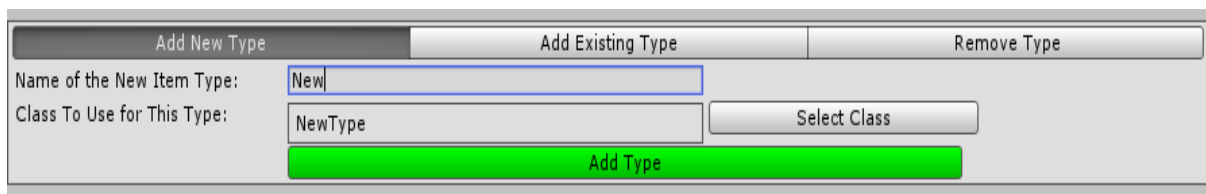
Since V1.1, adding and removing item types (previously categories) to the VID is very easy, as there is now a special window to help you with that. That window can be accessed the same way as the VID or via the shortcut 'Ctrl-Shift-E'.



## Adding Item Types:-

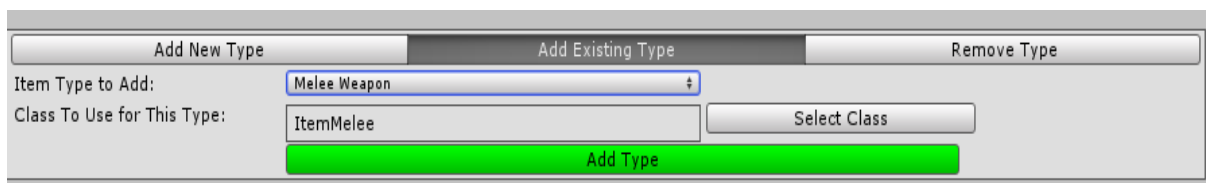
You can either add a completely new type, or you can add one that already exists in the item type enum.

Adding completely new types works by you supplying the name of the new type (e.g. Pistols) and the **serializable** class that **inherits from 'ItemBase'** and then pressing the 'Add Type' button.



The screenshot shows a software interface with three tabs: 'Add New Type' (selected), 'Add Existing Type', and 'Remove Type'. Under the 'Add New Type' tab, there are two input fields. The first is labeled 'Name of the New Item Type:' and contains the text 'New'. The second is labeled 'Class To Use for This Type:' and contains the text 'NewType'. To the right of the second field is a button labeled 'Select Class'. Below these fields is a large green button labeled 'Add Type'.

Adding categories of existing types works exactly the same way except that instead of you naming a new type, you simply choose one.



The screenshot shows the same software interface, but with the 'Add Existing Type' tab selected. The 'Item Type to Add:' field now contains a dropdown menu with 'Melee Weapon' selected. The 'Class To Use for This Type:' field still contains 'ItemMelee'. The 'Select Class' button and the large green 'Add Type' button are still present.



## Removing Types:-

Removing types works just as you might expect, you select a type and press a button and it and the items in it are completely removed, though there are a few things to keep in mind when doing this.

- Unless you have some kind of version control (you really should) this act generally **cannot** be undone.
- Prefabs of items of the deleted item type and items on objects **remain**.
- The same type can be added again, but the items previously in it are lost.
- The ‘Remove Item From Enum As Well’ check decides whether the type in the ‘ItemType’ enum is removed.
- All subtypes of the removed main type are removed as well.

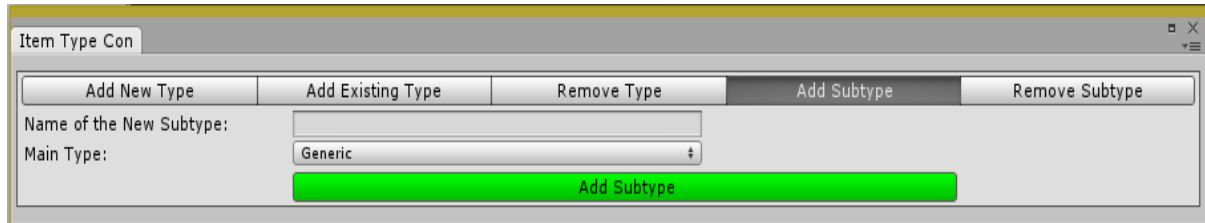
# Item Subtypes

With the update to V2.0, item subtypes has been introduced. **Item subtypes allow further filtering** of items in a type. Before going in to details regarding the actual addition and removal of item subtypes, we will see some **important properties and facts** about them.

1. Item ordering in a subtype is completely independent of the ordering in the main type.
2. Searching in a subtype only searches the items in that subtype.
3. When a new item is added while the user is in a subtype that item is automatically added to that subtype.
4. An item **can** be in multiple subtypes at the same time.
5. A type can have any number of subtypes, but two subtypes cannot have the same name.
6. Two subtypes belonging to two **different** main types **can** have the same name.
7. If a subtype is removed, the items are **not** deleted from the main type.
8. If a type is removed, all of its subtypes are removed as well.
9. If in a subtype, the 'Update Type Prefabs' button will only update the prefabs of the items in that subtype.

## **Adding Subtypes:-**

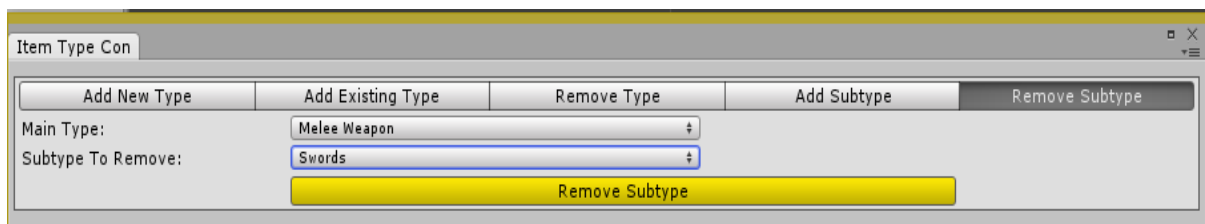
The process of adding a subtype is very similar to adding a new item type. All you have to do is open the 'Item Type Control Window' and go to the 'Add Subtype' section.



Input the name of your new subtype and select the main type it will belong to, click the button and once Unity finishes refreshing you are done!

## **Removing Subtypes:-**

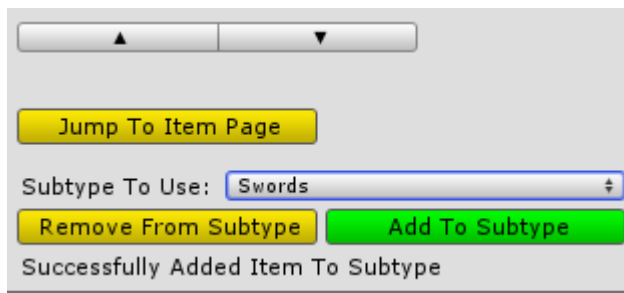
Removing subtypes is again very similar to removing types. Select the main type first, and then select the subtype you want to remove from the list of subtypes belonging to the main type you have selected.



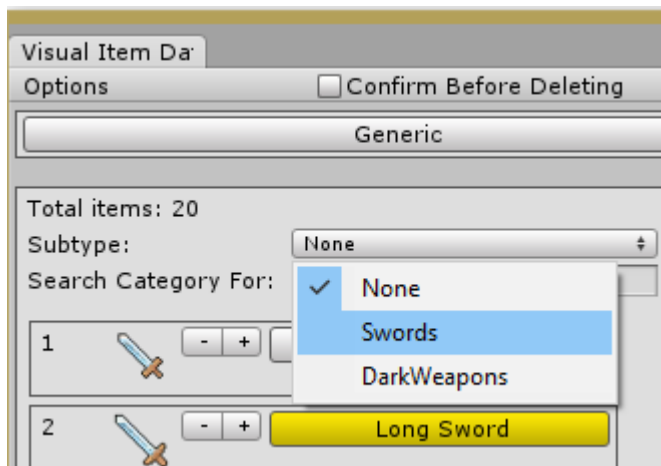
Confirm your decision and the subtype will be removed.

## Using Subtypes:-

Once created, items can be added to a subtype by selecting the subtype and pressing the green 'Add to Subtype' button. An item can also be removed from the selected subtype by pressing the yellow 'Remove From Subtype' button.



To change the view to a subtype simply select a subtype from the subtype dropdown above the search bar.

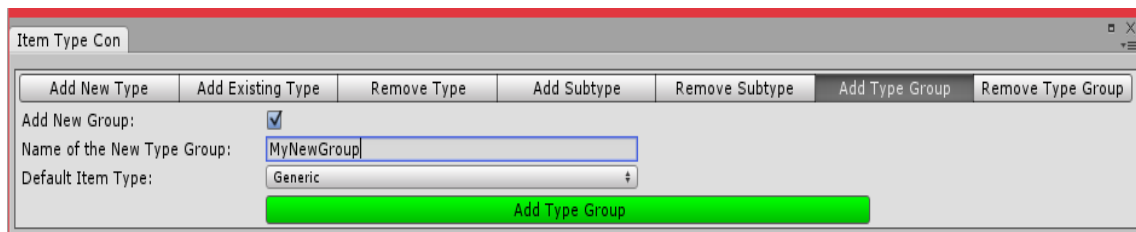


This will then only show the items in that subtype and with their own unique order.

# Item Type Groups

With the V3 update, item type groups have been introduced. They are basically subtypes but for item types instead of items.

Item Type Groups are very useful if you have many item types, they allow you to see only a select few at a time and help you stay focused on the groups you are working with.

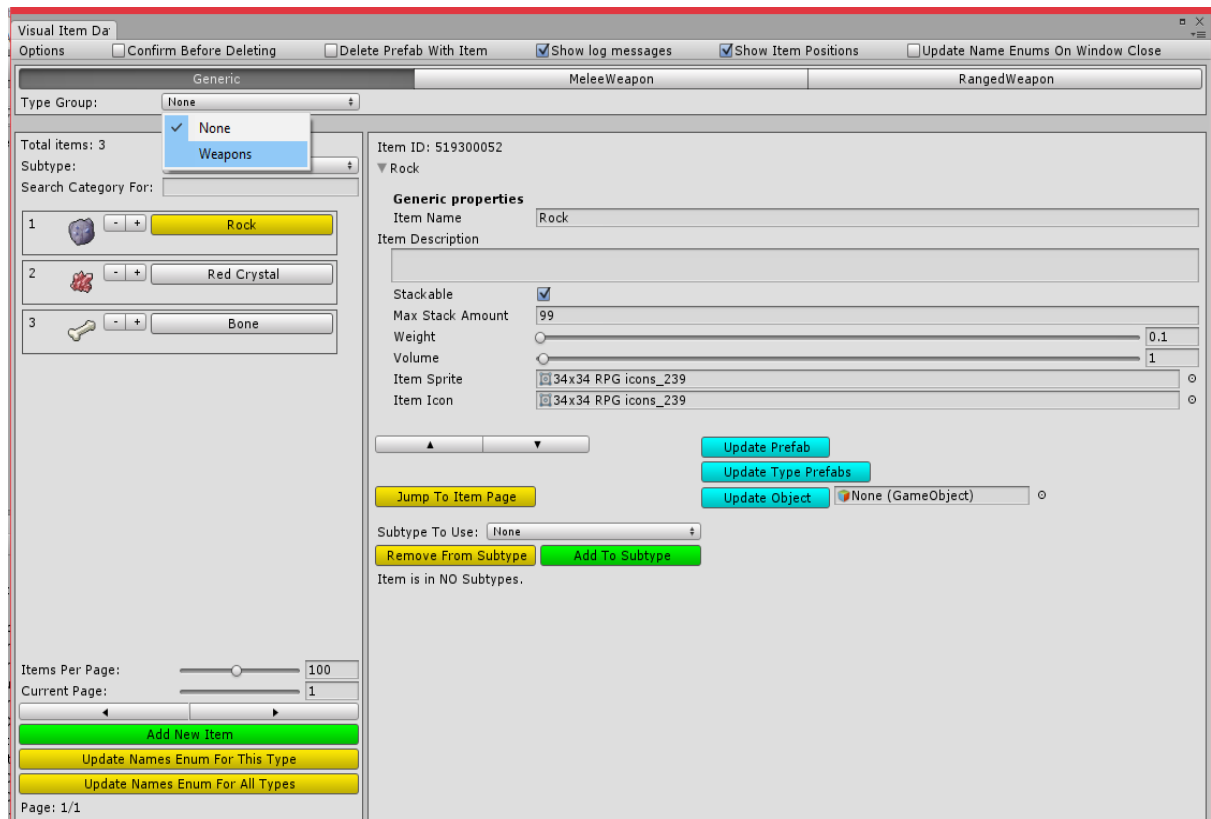


They work very similarly to subtypes, and the general workflow to adding a new group is:

1. Create a new type group with a default item type
2. Add whatever item types to your new item group.

You can either add a completely new group by using the tick at the top, or simply add an item type to an existing group. Removing works the same way.

Once this is done, you can select a certain group to show, like what is shown below:



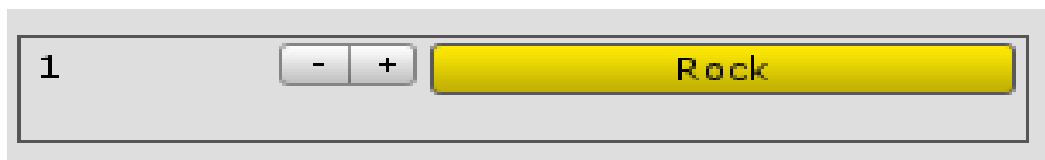
Selecting weapons will now filter to only the melee and ranged weapon item types, since they were added there. Item types can be added/removed to and from item groups at will and with ease.

# Duplicating and Deleting Items

The Item Database window gives you the ability to easily duplicate and delete items.

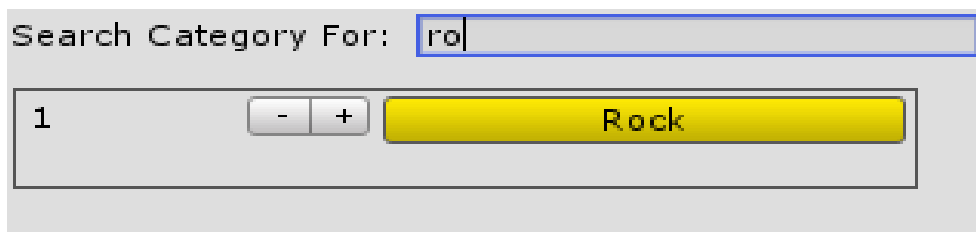
It is especially useful to duplicate an item when the new item you want to make is similar to an existing one, except that some values are different, in which case duplication can greatly increase the speed of creating new items.

Deletion and duplication is done using the ‘**minus**’ and ‘**plus**’ buttons, respectively. Both of which are located next to an items name.



# Searching for items

You can search for an item in the current type (or subtype) by typing its name (or part of it) in the search field and all items that contain the typed letters will appear in the results.



The image shows a search interface. At the top, there is a text input field labeled "Search Category For:" containing the text "ro". Below this, there is a table with one row. The first column of the table contains the number "1". The second column contains a yellow button labeled "Rock".

If say you are on the first page, and the item you searched for and selected is on the tenth page, you can directly jump to the item's page by clicking on the 'Jump To Item Page' button.

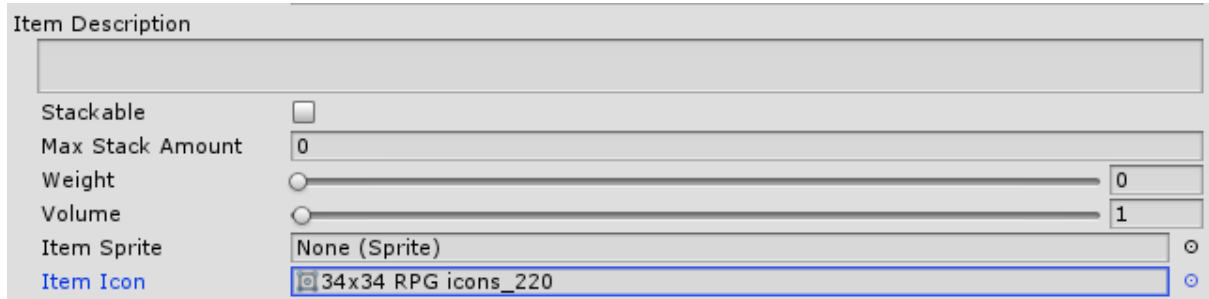


Note that the search filed is **not** case sensitive.



# Showing an Item's Icon

By simply filling the 'Item Icon' field



The screenshot shows a window titled 'Item Description'. It contains several fields: 'Stackable' (checkbox), 'Max Stack Amount' (text box with '0'), 'Weight' (slider), 'Volume' (slider), 'Item Sprite' (text box with 'None (Sprite)'), and 'Item Icon' (text box with '34x34 RPG icons\_220'). The 'Item Icon' field is highlighted with a blue border.

The icon of the item will appear next to its name in the editor.



This can also very easily be changed to use some other field to show icons from, as will be discussed in the '*Using a Different Field to Display Icons*' programming section.

# **For Programmers**

# A Very Important Note

As of V1.1, code generation is being used to add/remove categories to the VID. This code generation uses special comments placed in key places in the some of the VID files.

They are single line comments that look like this:

```
}//#VID-AIE
```

They **always** start with ‘#VID-’. **Never** change their position in anyway or place anything **between** special comments that are the same except for the last letter being either ‘B’ or ‘E’ since that could seriously mess up the code generation and make life difficult.

Aside from that editing these files is fine, just do not play with the special comments and you are good.

# The General Layout of the Item System

All the default classes of the item system are inside the 'ItemSystem' namespace.

The 'ItemSystem' namespace consists of the five item scripts, item container script and item system utility script among others. The **item utility script** should be used in a similar way to a game manager object, where **only one instance of it should ever exist at any given time**.

The five item scripts can be used to make categories that you see in the editor window, and are what you actually see and edit in the editor window.

Now the default system relies on using interfaces along with inheritance for the item scripts. All item scripts inherit from the 'ItemBase', which gives them their shared properties.

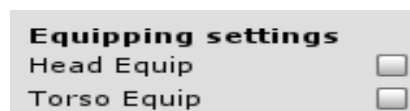
However, unique properties (like damage) are gained through the implementation of wanted interfaces, so for example if you want to deal damage you can implement the IDamage interface, which makes sure you have everything you need.

Even though the item scripts use interfaces, interfaces are **not** needed for the functioning of the Database editor window **at all!** Meaning you can completely get rid of them and the editor window will be unaffected.

Item container scripts include a variable of type 'ItemBase', which can hold any of the item types since it is their parent. To find out which item type to cast to, you should check the item type first to figure out what you should cast to, and then you can access the unique properties of the item (if any).

# Changing the Look of Items in the Database

If you want to turn a variable into a slider in the database, put some space between items or simply put a header like:



All those and more can be done by using the built-in Unity attributes, like the 'Range' and 'Header' attributes.

```
[SerializeField, Range(1, 100), Header("Unique properties")]  
int condition = 100;
```

The full list of attributes can be found in the Unity scripting API under 'Attributes'.

Though not tested, things like property drawers should also work flawlessly, so you work with the database just as you work with the inspector, except that you change things directly in the item scripts.

# Creating Instances of Items at Runtime

As previously mentioned in “Creating and Updating Prefabs”, you do not have to make prefabs of items to be able to instance them at runtime, you can make an instance of an item directly from the database, and there are methods to do that for you!

Say you wanted to make an instance of an item called ‘Heavy Sword’, the way to do this is very simple.

All you need to do is call the ‘GetItemCopy’ method in the ‘ItemSystemUtility’ class.

Pass to it the name of the item you want (or the id of you happen to have it) and its type, and it will take care of everything else, from retrieving the item to copying the properties.

```
ItemMelee heavySword;  
  
void Start()  
{  
    heavySword = (ItemMelee)ItemSystemUtility.GetItemCopy("Heavy Sword", ItemType.MeleeWeapon);  
}
```

In case you want to avoid casting, you can simply **use the generic versions** of those methods to directly get the wanted type, which is **recommended**.

```
void Start()
{
    ItemRanged bow = ItemSystemUtility.GetItemCopy<ItemRanged>("Bow", ItemType.RangedWeapon);
}
```

If you want to get a random item, you can use the ‘GetRandomItemCopy’ methods instead.

In addition to all of these, you also have the ability to get a list of all the items in a type/subtype by using the ‘GetAllTypeItems’ and ‘GetAllSubtypeItems’ methods like so.

```
void Start()
{
    //Gets all the ranged items
    List<ItemRanged> rangedItems = ItemSystemUtility.GetAllTypeItems<ItemRanged>(ItemType.RangedWeapon);

    //Gets all the items in the 'Bows' subtype
    List<ItemRanged> bows = ItemSystemUtility.GetAllSubtypeItems<ItemRanged>(RangedWeaponSubtypes.Bows.ToString(),
        ItemType.RangedWeapon);
}
```

Do note that these two methods **could** be slow (depending on your item counts), so **mind when you use them**.

Any changes done to such instances do **not** affect the original item data in the database.



There also exists two extra variants for the subtype overloads of ‘GetRandomItemCopy’ and ‘GetAllSubtypeItems’ that allow you to specify a number of subtypes (that belong to the same ItemType) that the items will be retrieved from.

For example, if you have three subtypes in your Melee weapons: Bronze, Iron, Diamond. And you wanted to get a random item from either the Iron or Diamond subtypes, then you can use these variants in the following way.

```
void Start()
{
    ItemMelee weapon = ItemSystemUtility.GetRandomItemCopy<ItemMelee>(ItemType.MeleeWeapon, MeleeWeaponSubtypes.Iron.ToString(), MeleeWeaponSubtypes.Diamond.ToString());
}
```

Simply set the main ItemType and the subtypes to use (you can input as many subtypes as needed to the function) and one of the subtypes will be randomly selected and used.

Since V3.2, there now exists ‘GetItemOriginal’ methods that return a reference to the original item in the database without copying it. In the editor, changes while using this reference are permanent. In game changes to these references remain until game reload. **Only** use this if you know what you are doing!

# The ItemDatabaseEditorWindow- User Class

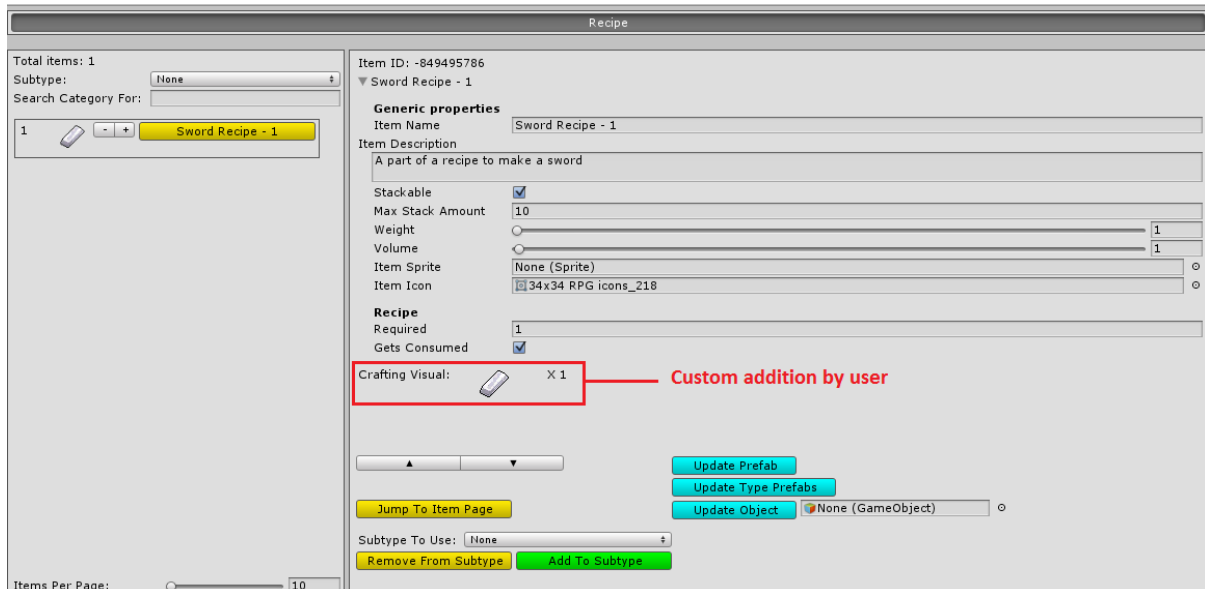
The ‘ItemDatabaseEditorWindowUser.cs’ script file is a new file introduced in V2.5 of the VID.

It contains the second part of the *partial* class ‘ItemDatabaseEditorWindow’, which is the class that handles drawing the database editor window which you frequently use. It is **automatically generated** when you first open the database editor window.

This new class contains two methods, the ‘AfterItemDrawing’ and ‘AfterSubtypeButtonsDrawing’ methods, both of which are **called automatically** by the editor window when an item is being shown and are passed the shown item as a parameter. These methods can be used to **directly customize the item section of the editor window**.

The ‘**AfterItemDrawing**’ method is called right **after** an item is shown on the database and **before** all the buttons under it. So adding editor code in that method will add directly above those buttons.

The example below showcases one possible use case, where visuals for a ‘Recipe’ type are custom added to the window.



And the code for this:

```
public partial class ItemDatabaseEditorWindow : EditorWindow
{
    void AfterItemDrawing(ItemBase shownItem)
    {
        Texture2D iconPreview = AssetPreview.GetAssetPreview(shownItem.itemIcon); //Cache for efficiency

        EditorGUILayout.BeginHorizontal();
        GUILayout.Space(22); //Push to the right
        GUILayout.Label("Icon: ", GUILayout.Width(32));
        GUILayout.Label(iconPreview, GUILayout.Width(64), GUILayout.Height(64)); //You can adjust image size here
        EditorGUILayout.EndHorizontal();

        EditorGUILayout.BeginHorizontal();
        GUILayout.Label("Crafting Visual: ", GUILayout.Width(110));
        GUILayout.Label(iconPreview, GUILayout.Width(64), GUILayout.Height(64));
        GUILayout.Label("X " + ((NewType)shownItem).Count); //Show count
        EditorGUILayout.EndHorizontal();
    }
}
```

The ‘**AfterSubtypeButtonsDrawing**’ works exactly the same, except that it draws under all the other buttons in the editor window.

# Dealing With Item Subtypes

When a subtype is created for a main type for the first time, **an enum is created in the ‘VIDItemLists.cs’ script** to store all the subtype names for that main type. These enums are always named like this: name of main type + “Subtypes”.

Therefore, if the main type is called ‘MeleeWeapons’, then the name of the resulting enum will be “MeleeWeaponsSubtypes”.

Now every time a new subtype is added to a type a new entry is added to its enum. **This entry is the same name as the added subtype.**

In addition to this enum entry, **all subtypes of all the main types** are added to the **subtype list** in the ‘VIDItemLists’ script.

```
public List<ItemSubtype> subtypes = new List<ItemSubtype>();
```

Each item subtype entry has three things: a name, a main type and a list of ints that stores the IDs of all the items in the subtype.

The main purpose of these enums is so that the subtype name can be retrieved easily so that it can be used in the overload of the ‘GetRandomItemCopy’ method. This new overload takes in a subtype name as a string and returns a random item from that subtype.

This is where the enums come in. Instead of remembering every subtype name and risking making spelling mistakes, you can easily select the subtype you want from the enum and then use ‘.ToString()’ to get the name of the subtype as a string like so:

```
void Start()
{
    //Get a random item from the swords subtype
    ItemMelee randomSword = (ItemMelee)ItemSystemUtility.GetRandomItemCopy(MeleeWeaponSubtypes.Swords.ToString());
}
```

# Item Names Enums

Item Names Enums is a new feature introduced in V3. Previously when you wanted to load an item you did something like this:

```
ItemMelee equippedWeapon;  
  
// Use this for initialization  
void Start()  
{  
    equippedWeapon = ItemSystemUtility.GetItemCopy<ItemMelee>("Short Sword", ItemType.MeleeWeapon);  
}  
  
// Update is called once per frame  
void Update()  
{  
}
```

This had a lot of problems. You had to remember the name of the item, a change to the name would require a change in code, any spelling mistake would be hard to debug and you had to constantly switch between your code editor and the database to know what items you had at your disposal.

Now the new Item Names Enums feature aims to remedy all of these problems, and more!

Two new buttons are introduced in the editor that generate an enum per item type. Each enum contains all the items in that type. The entry is the name of the item (assuming it is valid, no name collisions, with no spaces, and starting with a letter), and **the number assigned to each entry is its ID.**

This is very powerful. The major effect of this is that it overhauls how you load items. Since now you can use code like this:

```
public MeleeWeaponItems weaponToLoad;
ItemMelee equippedWeapon;

// Use this for initialization
void Start()
{
    equippedWeapon = ItemSystemUtility.GetItemCopy<ItemMelee>((int)weaponToLoad, ItemType.MeleeWeapon);
}

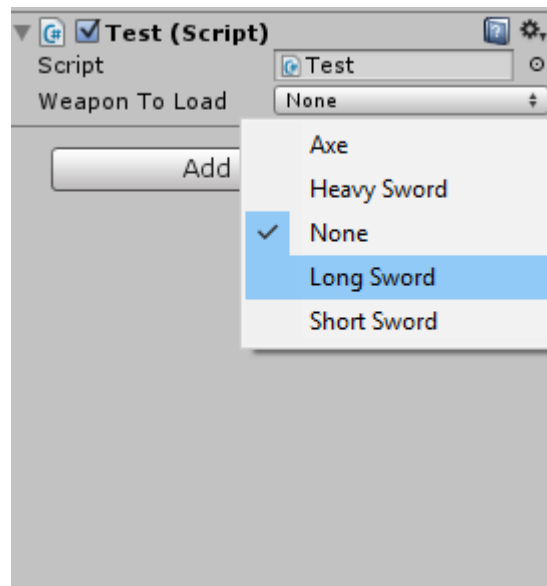
// Update is called once per frame
void Update()
{
}
```

This might not seem that different, but it is. First you will notice the “MeleeWeaponItems” enum type variable. Item type name enums are named like this:

“YourItemTypeName” + “Items”

The ID of the item is used to get an instance of the item from the database.

This now shows a dropdown in the editor like this:



This means you can easily change what items are loaded without changing any code!

In addition, this removes the need to remember names and/or change them in code every time they are edited in the database.

These enums can also be used while coding to get the IDs of any item, in case they are needed.



# Using a Different Field to Display Icons

The ‘itemIcon’ variable is for the icon that will normally be displayed in something like an inventory, while ‘itemSprite’ is what will be displayed in game (in case of 2D. You can replace it with a mesh for 3D).

However, if say you want to get rid of ‘ItemIcon’ and directly use the sprite in ‘ItemSprite’, or generally want to use something else as display, you should change the following code in the ‘DrawSideArea’ method in the ‘ItemDatabaseEditorWindow’ class.

```
//Show icon  
GUILayout.Label(AssetPreview.GetAssetPreview(item.itemIcon), GUILayout.Width(32), GUILayout.Height(32));
```

Simply replace the ‘itemIcon’ variable with whatever variable you want to be shown. (Of course, that variable needs to be something that can actually be drawn)

# Changing Where Assets are Generated

**By default**, the Item Database is generated in the “Visual Item Database/Resources” folder, and the prefab folders (item type folders) are placed in the “Visual Item Database/Item Prefabs/*item type*” folder.

These folder paths are assigned in code at the top of the “ItemDatabaseEditorWindow” class.

```
string objectToFind = string.Empty;  
static readonly string databasePath = @"Assets/Visual Item Database/Resources/ItemDatabase.asset"; //Path of the item database  
readonly string itemPrefabsPath = @"Assets/Visual Item Database/Item Prefabs/"; //Path that contains the category folders
```

By changing these two paths, you can change where the item database is created and where the prefab folders are searched for.

**Note:** do **not** remove “/ItemDatabase.asset” from the database path, because this is needed for the database to function correctly.

**Note2:** The database base **must** be in a ‘Resources’ folder, otherwise it cannot be loaded at runtime. But the ‘Resources’ folder itself can be anywhere you desire.

# Transferring Items from VID 3.1 to VID 3.2

If you have downloaded this version of the VID (V3.2) on top of the older version (V3.1) then you probably have items in V3.1 and you will probably want those, so this section will help you in the task of moving them.

Before moving on, make sure that you have imported everything in the package. **You can skip importing the item type classes and the ‘ItemBase.cs’ script** since they did not change, but **import everything else**.

All you have to do is go to ‘Assets/Visual Item Database’ and click on ‘Update Database Version’. You will get three different buttons one after the other, simply click on each one, **wait for Unity to reload** and then click on the next until the window closes itself and you are done.

Now open the VID like usual and you should see the previously empty database now containing your old items.

Once you are **sure everything is as you want it**, you can go ahead and delete the ‘ItemDatabaseV31’ and ‘ItemDatabaseV31’resources, the ‘ItemDatabaseV31.cs’, ‘VIDItemListV31.cs’ and ‘DatabaseVersionUpdater.cs’ scripts as they are no longer needed.

# Adding New Properties to Items

In case you want to add new properties (variables) to the item scripts, you first have to decide whether you want these properties to appear on every item or only on a single item type.

If you want a property to appear on each item type then you have to add it to the ‘ItemBase’ script, and since all other item types inherit from it, they will automatically get it. Just make sure it is a type unity can serialize, otherwise you will **not be able to see it** in the database editor window.

However, if you want it to appear only on a specific item type then you have to place it in the respective item script. So, say you want to have a variable that stores a sword’s length, you have to put it in the ‘ItemMelee’ script.

This is as far as adding new properties goes, but now updating them remains, which is also very easy.

If you added a generic (general) property, then you have to assign (update) it in the ‘UpdateGenericProperties’ method in the ‘ItemBase’ script.

On the other hand if the property was specific to a type, then you have to assign it in the

‘UpdateUniqueProperties’ method in the script of that type, **not** the empty method in the base script.

# Useful Bits of Information

- **Never** change the *ID* of an **instanced** item at runtime. This is to avoid possible bugs and to be able to get the original item from the database.
- **Never** change the *type* of an **instanced** item. Because you need it to know how to cast the item variable, and because of what is stated in the previous point.
- You do **not** have to make a prefab of an item to make an instance of it at runtime.
- Do **not** increase the limit of the ‘itemsPerPage’ variable too high as it could cause lag with big databases.
- Search is **not** case sensitive and is per type/subtype.
- Do **not** be afraid of making the database big, it has been tested with over 10,000 items, and remained very stable and relatively lag free!
- If you want to make an instance of an item, **never** use the ‘GetItem’ method in the database class since doing so will result in the original database item being changed, which you never want. **Always** use the ‘*GetItemCopy*’ method.
- You can watch short tutorials on the database [here](#).

- Always use the *item names enums* when dealing with items, it is much safer compared to hardcoding item details, like IDs or item names.
- Use *item type groups* when you have a large number of item types to make your work more organized and better focused.
- You can access all the *raw* data of the VID through the *VIDItemListsV3* class. However, **only edit those if you know what you are doing.**

# **Contact Information**

- Check out more at: [bloeys.wordpress.com](http://bloeys.wordpress.com)
- You can send me an email at: [bloeys@outlook.com](mailto:bloeys@outlook.com)
- You can catch me in Twitter at: [@bloeys](https://twitter.com/bloeys)
- Catch me at Reddit by: [/u/bloeys](https://www.reddit.com/user/bloeys)
- In the Unity forums and such, I go by the same name as well ;)