Documentation

[Github](#) [Wiki](#) [Unity Thread](#) [UMA Discord](#) [Code Documentation](#)

Contents

About UMA

Unity Multipurpose Avatar, is an open avatar creation framework, it provides both base code and example content to create avatars. Using the UMA pack, it's possible to customize the code and content for your own projects, and share or sell your creations through Unity Asset Store.

UMA is designed to support multiplayer games, so it provides code to pack all necessary UMA data to share the same avatar between clients and server. It may be necessary to implement a custom solution depending on your needs to optimize and reduce serialized data.

Minimum Requirements

UMA 2.11 requires Unity 2019.4 or above. Alpha or Beta versions of Unity may or may not work, but are not supported until the release version.
UMA 2.11 requires .Net 4.X API compatibility level and .Net 4.X equivalent scripting runtime version in player settings.
If using blendshapes, GPU skinning should be disabled.

Performance and Memory Usage

UMA framework provides a set of high-resolution content, flexible enough for generating a crowd with tons of random avatars or high quality customized avatars for cutscenes. Source textures are provided for generating final atlas resolution of up to 4096x4096. Depending on the amount of extra content being imported to the project, it might be necessary to handle memory management or reduce texture resolution.

Every UMA avatar created has its own unique mesh and Atlas texture, requiring extra memory. The standard atlas resolution of 2048x2048 is recommended for creating a small number of avatars, for games creating on a huge number of avatars, using lower atlas resolution or sharing mesh and atlas data will be necessary.

UMA was initially planned to provide 50 avatars on screen, but the latest version can easily handle a hundred of unique avatars.

When using a large number of slots and overlays, with large textures, it is recommended to use Addressables, so UMA can load and unload data from memory.

How Does it Work?

Creating 3d characters is a time-consuming process that requires a number of different knowledge areas. Usually, each character is created based on a unique mesh and rig and is individually skinned and textured.

When developing games that might require a huge number of avatars, it's expected to develop a solution to handle avatar creation in an efficient way. Usually they start from a set of base meshes and follow standards to be able to share body parts and content. Each project ends up with a different solution, and it's hard to share content or code between them.

UMA is meant to provide an open and flexible solution, which makes it possible to share content and code between different projects, resulting in a powerful tool for the entire community.

UMA has two main goals: Sharing content across avatars that uses same base mesh and optimizing created avatars, while providing the ability to change avatar shape in real-time.

To achieve that, a special rig structure was created that handles bone deformation in a strategic way that makes it possible to deform UMA shape based on changes on bones position, scale and rotation.

UMA example avatars are based on two base meshes, a male and a female. Each of them can share clothing and accessories and can be used as a base on the creation of new meshes for different races.

Because clothes and accessories are skinned to UMA base meshes, they receive the same influence of UMA body shape, so any mesh deforms to any UMA body.

This way, if you have an armor or dress, it can be shared between all male or female avatars, even if they have very different body shapes.

An overlay system was implemented that makes it possible to combine many different textures when generating the final atlas. Those extra textures can be used to create even more variation to each avatar, and can be used for clothes, details and many other possibilities.

UMA optimization occurs in many steps: Each UMA avatar can have a unique texture atlas providing all necessary texture data, this makes it possible to have each UMA generating a single draw call, in the case of a single material being used.

All UMA parts are baked together into one final mesh, which reduces the calculations involved in processing. Joen Joensen from Unity team implemented an advanced skinned mesh combiner to accomplish this.

How is Content Created?

See the Content Creation Section

What’s new in 2.11

Minimum supported version is now 2019.4

Generation

- Added support for more than 4 boneweights. Can now use as many boneweights as you like.
- Reworked the SkinnedMeshCombiner to use the new boneweights.
- New "no coroutines" method of generation results in faster build
- Skeleton removal during build is now done in build process to fix issues where skeleton is not available during some frames.
- New texture atlas fitting methods:

    - Reduce (with a variable reduction amount)
    - Best Fit Square (Expands the texture to the best fit square, and then reduces them to fit the atlas). This takes a bit more work, but wastes the least space.
- Broke out fit methods on generator
- Added property to sharpen reduced textures by forcing it to use a higher mip. Produces significantly more detailed output on textures that are fit.

Recipes

- Added ability to transform (scale and rotate) an overlay in a recipe (Not on first overlay - only additional overlays)
- Added ability to change color channel length on shared colors in recipe editor.
- Added Property Blocks to Overlay Colors (shared colors). Now you can set floats, vectors, ints, colors, matrixes, etc. on the materials.

ExpressionPlayer

- Added Pointing, Peace Sign, Rude sign, and grasp to Expression Player. Note: the arms/hands must be procedurally driven for this to look right (VR, etc).
- Fixed some errors where the Expression Player could generate errors during startup

DynamicCharacterAvatar

- UMA characters are now visible at edit time (can be disabled)
- Made an edit time DNA editor, combined with colors and wardrobe under "customization". • Added filters to colors in DCA customizer section
- Added UpdateBounds() function.
- Added load/save preset in the editor.
- Default colors are now white/black for the various colors on the race.
- New events: WardrobeAdded and WardrobeRemoved.
- Added "Always Rebuild Skeleton" option in the advanced settings. This will help you keep your rig clean and tidy, for a slight performance cost.

UMAMountedItem

- New object to make mounting simpler. Add this to an object you want to make mountable. Specify the bone you want to attach it to, and any offset/rotation. Will automatically attach and process, and survive rebuilds.

OverlayData

- Made it possible to have Overlays with no texture channels (these overlays can be modified with material property blocks).

SlotData

- New: Wildcard slots. This is a type of utility slots that can be added to recipes, which allows you to place overlays onto slots (or other overlays) based on tag. So you can place an overlay onto a "Torso" without knowing the actual slot ID, just the tag. This allows you to reuse recipes for slots that have the same UV coordinates.

Slotbuilder

- Made blender slot rotation fixup optional.
- Reworked slot builder to emphasize the automated generation from FBX
- Can now specify bones to keep when building slots. (for mount points, etc)

Global Library

- Functions (and menu items) to backup and restore the Global Library.
- Made the Build Preprocessor optional (off by default)

Misc

- Set proportions on default female race to a bit less heroic.
- Fixed some possible warnings during overlay generation.
- Fixed race loading in editor so it does not need to instantiate the race to get slots
- Lot of minor issues where assumptions were made in code that occasionally were not true.
- URP conversion option will convert hair to use Speedtree8 shader
- Forearm Twist slot script will now let you set the names of the bones
- Reworked the bone cleaner to make it more robust
- Added flipping, highlighting to bone pose editor
- Added "find UMA" to bone pose editor, so you don't have to select it
- Bone Pose Editor will now remember the selected UMA when possible
- Bone Pose Editor will now allow mirroring on X,Y,Z axis. Default UMA (blender export) is on Y axis.
- Provided sane colors for additive channel when an overlay did not have all channel colors defined (issue with core samples)
- Tightened up avatars, and made the new avatars the default
- Numerous bug fixes on edge cases
- Better resource cleanup

DCARendererManager

- Added option EnableRenderers to turn on/off the renderer manager for toggling between first and third person.

Quick start

Example Scenes

View the example scenes. There are quite a few located in UMA\Examples. A good first one to examine is “UMA DCS Demo - Simple Setup”.

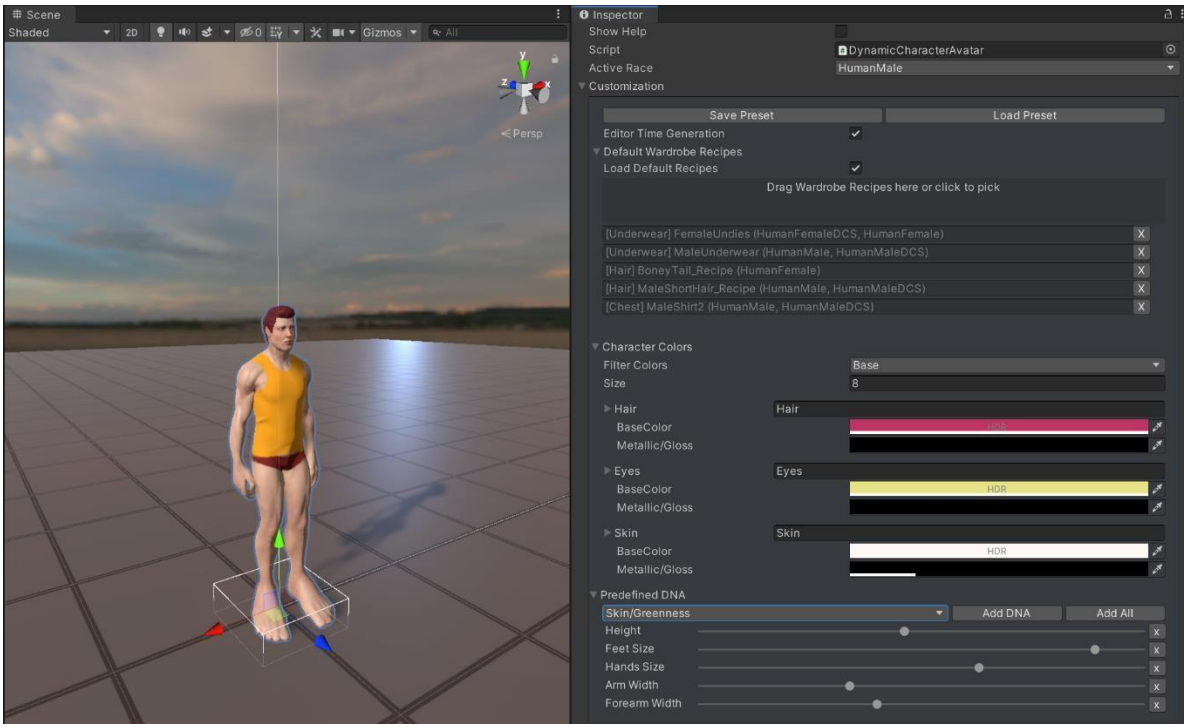
New Scene

1) Open the ***UMA/Getting Started*** folder - this folder contains various prefabs that can be used to get started quickly.

2) Find the prefab "UMA_GLIB" and drag this into the scene. This is the object that contains the UMA context, generator and mesh combiner objects. The context in this prefab goes directly to the UMA Asset Index to access items.

3) Next, find the prefab "UMADynamicCharacterAvatar" and drag this into the scene. This will be an UMA avatar. It contains the script "DynamicCharacterAvatar".

4) Set it's position to wherever you want and make sure there is an object or terrain underneath the avatar object. Also, make sure it is in view of the game Camera. It should generate and display the character, and allow you to make changes to it in the inspector, in the customization section:



5) From here you could change to other races or add new wardrobe items, update colors, add DNA and change it, etc. Note that only wardrobe items that match the current characters race will be loaded, but you can still add other items for other races.

Basic Concepts

UMAGlobalContext and UMAContext

The UMAGlobalContext contains the methods to access the assets needed to generate UMA characters. It accesses these items straight from the Global Library. A prefab that has been setup for most use cases has been provided in the **UMA/Getting Started** folder (UMA_GLIB).

There is another context (UMAContext) that accesses assets from a set of libraries. It loads items from various libraries that store references to the items in the scene. This context is kept for compatibility with older code. It is recommended when using UMA 2.10 to use the Global Context.

A Context of some type is required in the scene for the UMA system to work correctly. You can create this one time, and set the flag “Don’t Destroy On Load” to keep it resident – doing this requires you to structure your game in a specific way and is beyond the scope of this document. See https://docs.unity3d.com/ScriptReference/Object.DontDestroyOnLoad.html for more information.

UMAGenerator

The UMAGenerator component is included on the UMAContext in the prefab. This component is used to tune the generation of UMA Avatars.

FitAtlas	This value should be set true unless you have provided your own Texture Merge prefab. When this is set, textures are reduced as required to make all textures fit into the atlas.
Convert Render Textures	Checking this will convert the generated textures from RenderTextures to Standard Texture2D. This will slow the generation process quite a bit, but will also allow you to programmatically change the texture bits. Setting this is not recommended unless you specifically require this functionality.
Convert Mip Maps	Checking this will generate mipmaps. This is recommended.
Atlas Resolution	This will set the maximum size of any generated texture atlas.
Default Overlay Asset	If an overlay is not found during generation, this will substitute.
Initial Scale Factor	This is the scale factor applied to overlays when they are added to the atlas. Setting a scale factor of 2 will result in all overlays starting with their size divided by 2. This is useful for improving texture RAM usage on lower end systems. This should normally be set to 1, and increased to improve RAM usage.
Fast Generation	If this is checked, UMA will generate the skeleton and mesh for a queued UMA in one frame. If this is not checked, they will be generated in separate frames.
Iteration Count	How many iterations of the generation loop to process in one frame. If Fast Generation is enabled, then this is equivalent to how many UMA’s to generate per frame.
Garbage Collection Rate	How many iterations before collecting garbage and freeing memory.
Process All Pending	If you check this, all UMAs will be generated on the next frame.

Texture Merge prefab	If you provide your own texture merge module, it should be included on this prefab. This is an advanced option.
Mesh Combiner	If you provide your own mesh combiner, it should be included on this prefab. This is an advanced option.
Edit Time Atlas Resolution	This is the atlas size that is used during editing. You can make this lower so that generation is faster during editing.
Edit Time Scale Factor	Like the other edit-time option, this is used to tune the edit time generation, and is used in place of Initial Scale Factor during edit time.
Sharper Fit Textures	When textures do not fit into an atlas, they will be resized to fit. Checking this box will tell the generator to use a higher mip map to scale down to the size. If this is not checked, the default is to let the driver decide which mipmap to use, usually resulting in a fit texture that is slightly blurrier.
Atlas Overflow Fit Method	**Best Fit Square** – Using this method, when the atlas is overbooked, the generator will create the largest possible square to contain the the atlas textures, and then reduce the textures to fit in the atlas exactly. This results in the least amount of wasted space and the maximum amount of resolution. This is the default for UMA 2.11. **Decrease Resolution** – Using this method, the system will attempt to reduce the texture sizes on a percentage basis to fit in the atlas. The percentage size to reduce for each fit pass can be specified (previous versions of UMA used this method, with a hardcoded value of 0.5).

Dynamic Character Avatar

The DynamicCharacterAvatar is the highest level component to utilize UMA and the wardrobe system. It allows you to easily select races, set events, character colors, and default wardrobe recipes. A default prefab named ***UMADynamicCharacterAvatar*** is located in the ***UMA/Getting Started*** folder.

Wardrobe System

The Wardrobe system introduces the concept of “Wardrobe Recipes”. These containers allow you to select compatible races that they are to be used on, the wardrobe slot (not to be confused with regular slots) as well as other configuration options. Finally, you can add the various slots and overlays that represent this wardrobe recipe. In essence, this encapsulates the idea of an “item” or “clothing” that can used as one object. Each wardrobe item will contain one or more slots, and one or more overlays for each slot. They can even contain slots that have been used on the base race recipe, or on other wardrobe slots. When this happens, the extra slots are merged out, and the overlays are added to the existing slot. For example, if you want to have a tattoo on the face, you would add a face slot, and a tattoo overlay – even though

there is already a face slot (with the face overlay) on the base race. When the character is built, the extra face slot will be removed, and the tattoo overlay will be added after the face overlay.

Events

CharacterBegun

CharacterBegun is fired at the start of the generation process, when the generator picks the UMA out of the queue.

CharacterCreated

CharacterCreated is fired after the UMA has been completely generated. It is only fired on the first build.

CharacterUpdated

CharacterUpdated is fired after the UMA has been completely generated. It is fired on every build of the UMA.

CharacterDestroyed

CharacterDestroyed is fired when the UMA is destroyed, and the components of the UMA are being cleaned up.

CharacterDNAUpdated

CharacterDNAUpdated is fired after the DNA has been applied to the character body.

RecipeUpdated

RecipeUpdated is fired during the Build Process. This can happened during the call to BuildCharacter() call, or if you are using Addressables, after the assets have been loaded, and the UMA is ready to construct. At this point, the UMAData.UmaRecipe is valid, and can be modified if needed.

AnimatorStateSaved

During the generation process, the rig and avatar can be recreated. When this happens, the Animator state is saved, and then restored so there is not skip in animation. This event fires when the animator state is saved in case you need to save some additional data to be restored to the new animator.

AnimatorStateRestored

AnimatorStateRestored is called during the reset of the animator. See AnimatorStateSaved above for more information.

WardrobeAdded

WardrobeAdded is called when a new Wardrobe Item is equipped on the character. This event will pass the UMAData, and the UMAWardrobeRecipe that was added. You can get the icon and the wardrobe slot from the recipe.

WardrobeRemoved

WardrobeRemoved is called when a Wardrobe Item is removed from a character. This event will pass the UMAData and the UMAWardrobeRecipe that was removed. You can get the icon and the wardrobe slot from the recipe.

Click here for more information on Wardrobe recipes

Changing the avatar in code

To set a wardrobe recipe in script, all you need is the function:

DynamicCharacterAvatar.SetSlot(UMATextRecipe "wardrobe recipe")

This will attempt to apply the named recipe (looks for compatible race and appropriate slot). After setting the wardrobe recipe, the changes will not take affect until you call:

DynamicCharacterAvatar.BuildCharacter();

This is so the user can make several changes before attempting to rebuild the UMA.

Finally, to clear a wardrobe recipe, simply call;

DynamicCharacterAvatar.ClearSlot(string "wardrobe slot name")

The "wardrobe slot name" is the location that the recipe is set to, for example "chest", "head", or "hands".

To set a character color on your avatar use;

DynamicCharacterAvatar.SetColor(string ColorName, Color colorValue)

This version of SetColor will only set the color that is multipled by the albedo. To have more control over how the color is set, you can call SetRawColor:

DynamicCharacterAvatar.SetRawColor(string ColorName, OverlayColorData colors);

When using this, you must construct an OverlayColorData that contains a channel (Both multiplied and additive) for each texture in the overlay.

As with wardrobe recipes, colors are cached, so for changes to take affect you will need to call;

DynamicCharacterAvatar.UpdateColors(bool triggerDirty)

*note-set triggerDirty to true for immediate results, otherwise the colors will be set on the next "buildCharacter" call

Saving and loading from a string

You can save the DynamicCharacterAvatar to a string using an AvatarDefinition.

```
compressedString = Avatar.GetAvatarDefinition(true).ToCompressedString("|");
```

Then load him from that string:

```
AvatarDefinition adf = AvatarDefinition.FromCompressedString(compressedString, '|');
Avatar.LoadAvatarDefinition(adf);
Avatar.BuildCharacter(false); // don't restore old DNA...
```

Preloading DNA

You can preload DNA on a DynamicCharacterAvatar (DCA) by creating a UMAPredefinedDNA object, and adding it to your DCA after you instantiate it, but before you build it. This is not a Unity Monobehavior or ScriptableObject – it's a straight C# class that holds string/float pairs for the DNA. This DNA is loaded into the character during BuildCharacter().

To use this class, simply create a new instance, fill it with DNA and Values (using UMAPrefinedDNA.AddDNA(string,value), and then assign it to the DCA.predefinedDNA field after instantiation.

Random Generation

UMARandomAvatar

The UMARandomAvatar is a component to generate a Random DynamicCharacterAvatar. This component can also be used to test the generation of Avatars by generating an array of Avatars centered around the game object.

To create a Random Avatar, create a new Game Object, and place the UMARandomAvatar component on it.

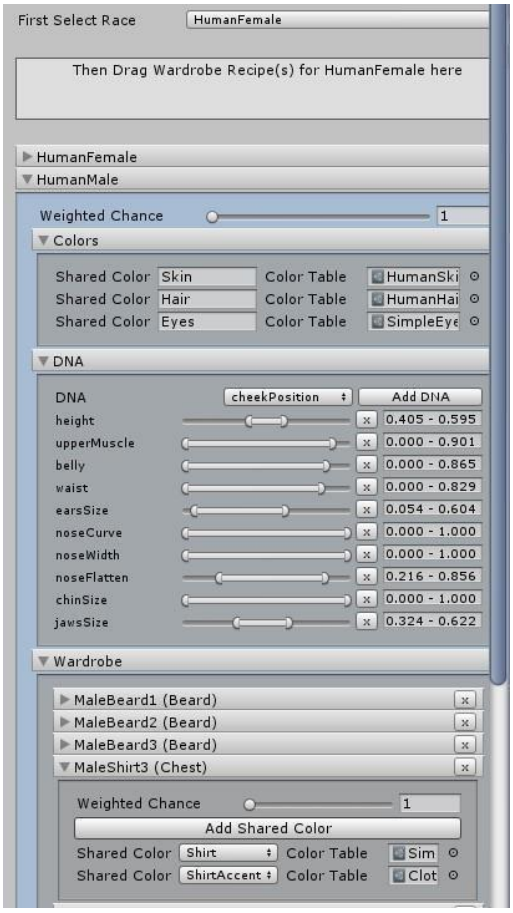
The UMARandomAvatar uses an object called a UMARandomizer to choose how to generate the object. See the documentation for this object below.

You can have as many UMARandomizers on a Random Avatar as you like – one will be randomly chosen. This allows you to specify different sets of random data to fine tune the generation – for example, make sure darker skin colors are matched with darker hair colors.

UMARandomizer

This is a scriptable object that defines the random properties for a character. A RandomAvatar can contain many UMARandomizers, but only one is chosen and used to generate a character. The UMARandomizer allows you to setup random data for specific races (for example, human male, human female). To use the randomizer, select the race you want to work on at the top in the inspector, and then drop all of the wardrobe assets for that race on the drop area. This will create the random race instance, and allow you to set the colors, chances, DNA, etc for that race.

Chances on each item default to 1. (there is a 1 in N chance of selecting that item – with N being the total number of “chances” for each similar item). For example, if you had 12 items, and each item had 1 chance, then the chance to select an item is 1 out of 12. If you set an item to 5 chances, then every item except that item would now have a 1 in 17 chance, while it would have a 5 in 17 chance of being selected.



UMAMountedItem

Sometimes you need to mount an item to a character (like a weapon, etc), Typically, you would turn of "Rebuild Skeleton", and you would then just attach that item to the bone. But sometimes, you need to keep "Rebuild Skeleton" turned on – this can be the case if you need to change races, and the races have different rigs, or if you are equipping and unequipping items that add bones (for example, hair, skirts, etc). When this is the case, you can use UMAMountedItem to mount the item to the bone. UMAMountedItem is applied to a prefab on an empty GameObject, and the mounted item is added as a child of that gameobject. This prefab should be added as a child of the GameObject that has the DynamicCharacterAvatar component.

Example:

```
GameObject go = GameObject.Instantiate(swordPrefab, avatar.gameObject.transform);
go.name = InstantiatedItemName;
go.SetActive(true);
```

UMAMountedItem fields

BoneName	String	This is the name of the bone that you want the item mounted to.
ID	String	This is a unique ID for the Mount Point. It's used to add a mount point as a child game object of the bone specified. This should be unique for each mount point.
Position	Vector	A position offset for this object. This should only be used for fine tuning – generally, you should adjust the child object with the item game object for best results.
Orientation	Quaternion	An rotation offset for this object. This should only be used for fine tuning – generally, you should adjust the child object with the item game object for best results.
IgnoreTag	String	This tag should contain the name of the ignore tag (usually "UMAIgnore"). This will be placed on the gameobject.
SetScale	Bool	When true, the object inherits the parent's object scale.
MountOnStart	Bool	When true, the item is mounted on Start.

Libraries and Global Library

The different UMA components are assembled at runtime. To find these, UMA can use several different methods.

UMAGlobalContext and Global Library

The new method in 2.10 is to use the UMAGlobalContext, and access the Global Library directly. It is recommended to upgrade to this method in 2.10 and above. However, the older versions are still in UMA for backwards compatibility.

UMAContext and Scene Libraries

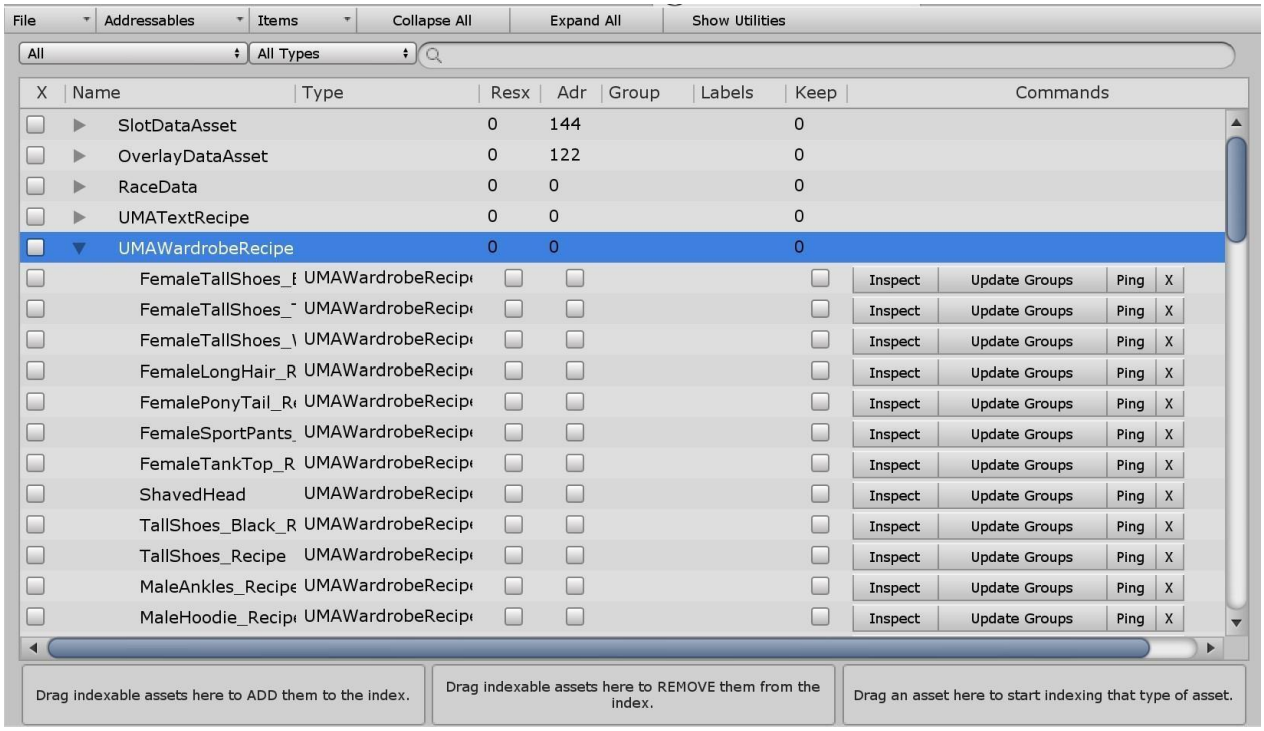
The original methods from 1.0 are using the scene-based libraries - OverlayLibrary, SlotLibrary, etc. These libraries are only valid for the scene they are in. To add items to these libraries, select them in the scene hierarchy, and drop items onto drop pad for the specific library components in the inspector.

UMAContext and Dynamic Libraries

These Dynamic Libraries were introduced in 2.5, and are versions of the scene libraries that can load items from resources or the global library, and cache them in the scene.

To view the global library, use the menu item UMA/Global Library Window to open the Library window, and drag/drop your folders containing the assets onto the drop pad. The library window shows all the indexed items by type (you can expand a type by clicking on it). You can inspect an item by clicking the ***Inspect*** button. You can select the item in the project window by clicking on the ***Ping*** button. You can filter the window by
If you need to access the asset, you can click the name of the asset, and it will be selected in the project. In addition, you can filter the global library by using the filter above the indexed types.

When using Addressables, the ***Adr*** column shows the if the item is addressable. The ***Group*** item shows the group it was added to, and the ***Labels*** column shows the labels that were added to the item. Items are labelled by the recipes they are in, so they can loaded by label in a logical manner. The ***Keep*** flag can be toggled on to tell the system not to delete this item when clearing orphaned items. The ***Update Groups*** button is used to update the addressable groups with the recipes contents. You should use this if you have changed or added items in the recipe and do not want to regenerate.



The three drop pads on the bottom are used to add and remove items from the menu, and to add types to the index. To add or remove items, drop the items (or folders) onto the pad, and they will be scanned for indexable items and added. If you want to include your own items in the index, drop any item on the last pad, and a new type will be indexed.

Menu Items

File/Rebuild From Project: The project is scanned, and every indexable item is added to the index.

File/Repair and Remove Invalid Items: Rebuilds the index using the GUID and/or location of the asset, updates the items with the correct flags, etc. This is helpful if you have moved assets around and/or updated them outside of Unity. If an item cannot be found, it is removed from the index.

File/Toggle Utility Pannel: This enables or disables the utility panel, which has some utilities for updating the items en-masse.

File/Empty Index: Removes everything from the index. This should be used only if you plan to recreate the library from scratch.

Addressables/Generate/Single Group (fast): This adds everything in the library into a single addressables group that is setup to “pack separately”. This option will produce an asset bundle for each item. Use the options in the UMA preferences to change how this group packs and labels items.

Addressables/Generate/Generate Groups (Optimized): This will generate a group for each recipe. Any item that is in one or more recipe will move into the shared group, which is packed separately. Using this option will produce the minimum number of Asset Bundles and IO, but takes longer to calculate.

Addressables/Delete Empty Groups: This will remove any empty groups from the addressables system.

Addressables/Remove orphaned slots: Use this only after building groups. This will remove any slots that were not added to addressables, were not marked “keep”, or were not forced to be included in resources (in the recipe).

Addressables/Remove orphaned overlays: Use this only after building groups. This will remove any overlays that were not added to addressables, were not marked “keep”, or were not forced to be included in resources (in the recipe).

The ***items*** menu contains various functions for selecting, deselecting and removing items.

Race

A "Race" in UMA is a way to specify the information that is specific to a single archetype. This includes TPoses, DNA Converters, Slots, Expressions, and wardrobe slots. For example there are RaceData entries for Male Humans and Female Humans, because they have slightly different ***TPoses*** and gender specific DNA converters, despite sharing the same DNA types.

The term "race" can be confusing in UMA. It can be used for avatars that share the same base mesh and DNA. For example, if you have a dwarf that is created from the human base mesh simply by changing the dna to be shorter and stockier, then it will still be the same "race" as the human, in UMA terms. This is completely abstract from any game level concept of character races. This is the reason why a human male and a human female are different "races" in UMA, because they have different base meshes. A Race can also define a “Base Recipe” and a set of Wardrobe Slots. The Base Recipe defines how your avatar looks without any wardrobe items applied. Wardrobe slots are used on the avatar to hold a single specific wardrobe recipe. Click here for more information on Races

DNA

The Dna of a UMA avatar are the possible changes or deformations that can be made to customize an avatar. While these are meant to be accessed programmatically, These individual items can also be accessed and changed at runtime to see immediate results on the "UMAData" component.

Click here for more information on DNA

Slots

All UMA content that provides a mesh is a slot. Slots are basically containers holding all necessary data to be combined with the rest of an UMA avatar.

For example the base meshes provided are normally split into several pieces, such as head, torso and legs... and then implemented as slots which can be combined in many different ways. An UMA avatar is in fact, the combination of many different slots, some of them carrying body parts, others providing clothing or accessories. Lots of UMA variation can be created simply by combining different slots for each avatar.

Slots also have a material sample, which is usually then combined with all other slots that share same material. Female eyelashes for example, have a unique transparent material that can be shared with transparent hair. It's necessary to set a material sample for all slots, as those are used to consider how meshes will be combined. In many cases, the same material sample can be used for all slots. UMA standard avatar material uses a similar version of Unity´s Standard Shader, but UMA project provides many other options.

The big difference between body parts and other content is that body parts need to be combined in a way that the seams won't be visible. To handle this, it's important that the vertices along mesh seams share the same position and normal values to avoid lighting artifacts.

To handle that, we provide a tool for importing meshes that recalculate the normal and tangent data based on a reference mesh.

Click here for more information on Slots

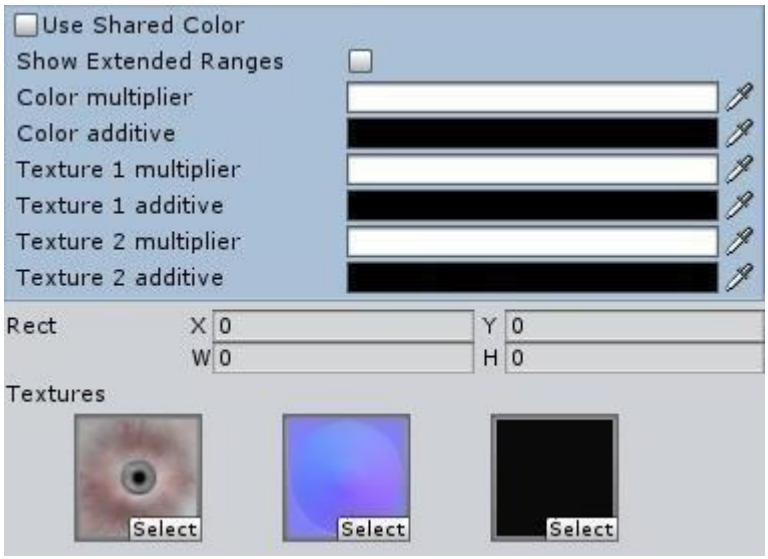
Overlays

Each slot requires at least one overlay set but usually receives a list of them. Overlays carries all the necessary textures to generate the final material(s) and might have extra information on how they are mapped. The first overlay in the list provides the base textures, and all other overlays included are combined with the first one, in sequence, generating the final atlas.

Colors

Each overlay can modify its color multiplicatively or additively. This can be used to tint overlays, such as having a skin texture that can be tinted to different skin tones.

Overlays can be layered on top of each other to selectively color parts of the final output.



Shared Colors

Shared Colors can be set as a common color to be used in an overlay set to use it. This way the same color can be used across overlays. For example, being able to set the same color on all skin textures (eg, body and face, etc…). They are set by creating a Shared Color at the top of your recipe. Then in the overlay, toggle “Use Shared Color” and select the shared color channel you set at the top of your recipe.

In 2.11, Shared Colors can now contain other shader properties. Float, Color, Vector, and int properties can be set directly in the inspector. Array, Matrix, Compute Buffer, and Texture properties should be set in code, using the “PropertyBlock” member on the OverlayColorData.

Click here for more information on Overlays

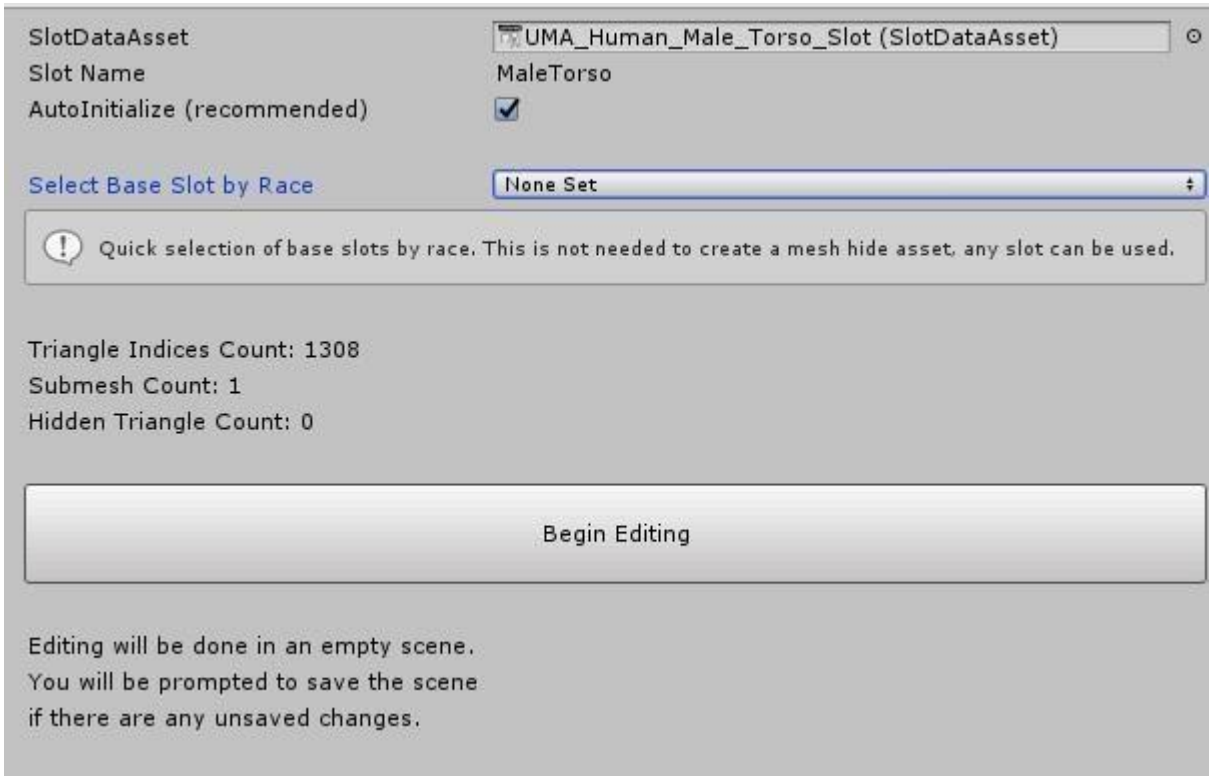
Mesh Hide Asset

The Mesh Hide Asset object is assigned to a Wardrobe recipe to hide portions of the geometry of **other** objects (either parts of the base race slots, or parts of other wardrobe items). This object is used to fix “poke through” when the item is equipped.

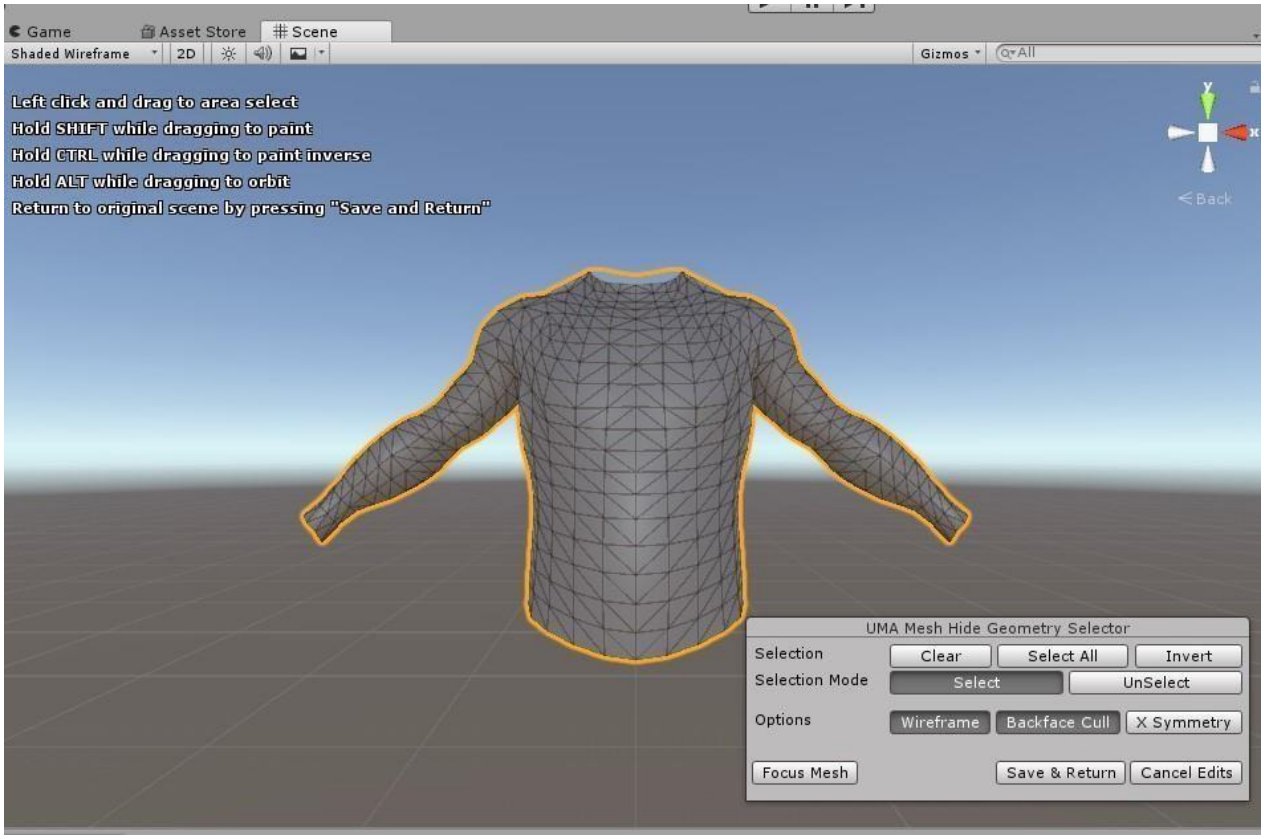
Note: Wardrobe items that completely cover base slots or other wardrobe items should use the functionality in the recipe to hide or suppress the hidden slots instead, as that is more efficient. Mesh Hide Assets are used when the slots are not completely obscured.

To create a Mesh Hide Asset, right click in the project, and select “Create ./ UMA / Misc / Mesh Hide Asset” from the popup menu.
In the inspector, select the SlotdataAsset that you want to be partially obscured. You can either use the select from the field menu, drop a SlotDataAsset on the field, or you can select the race and base race slot below the field.

After selecting the slot, the “Begin Editing” button will be enabled. Press this to load the slot into an empty scene for editing:

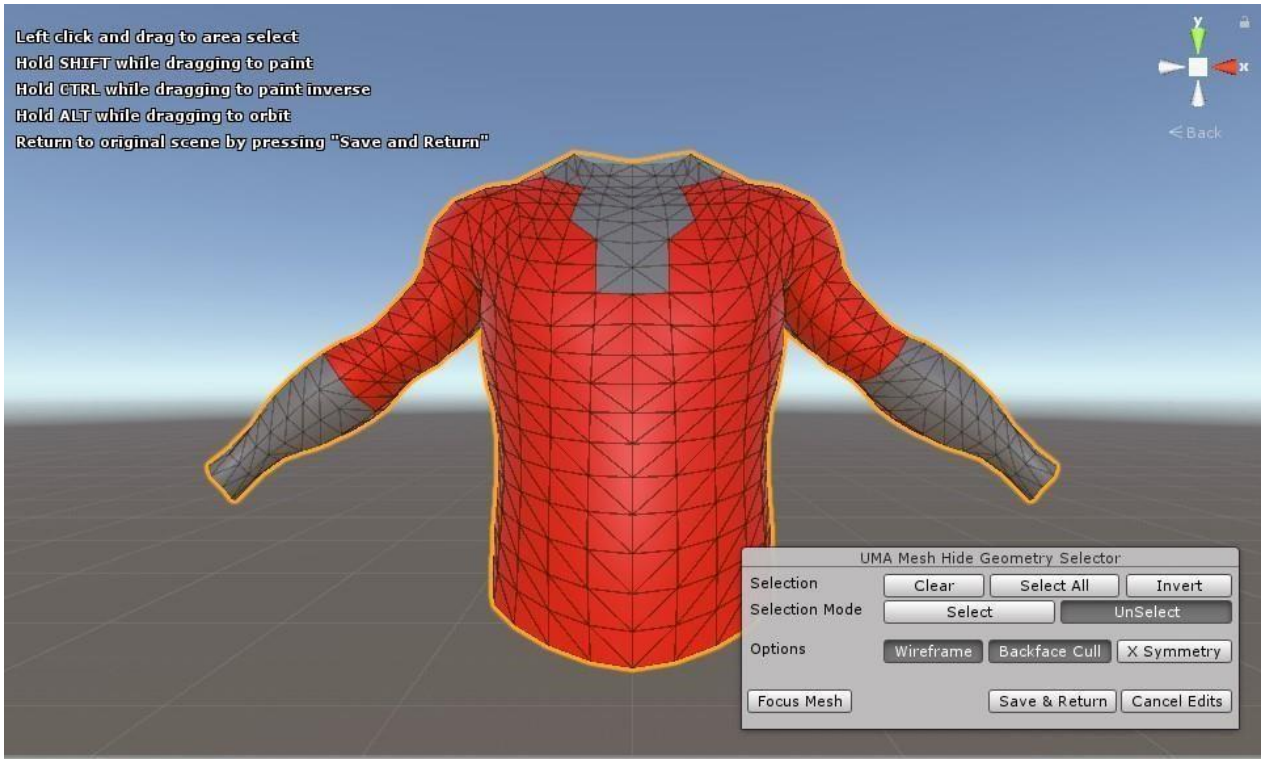


A new scene will open, and will look similar to the following. This scene allows you to select or unselect specific polygons:



Selected polygons are red. These are the polygons that will be hidden. Unselected polygons are gray. These polygons will remain and be visible.

For example, look at the following picture. The forearms and neck will be visible, and all other polygons will be hidden:



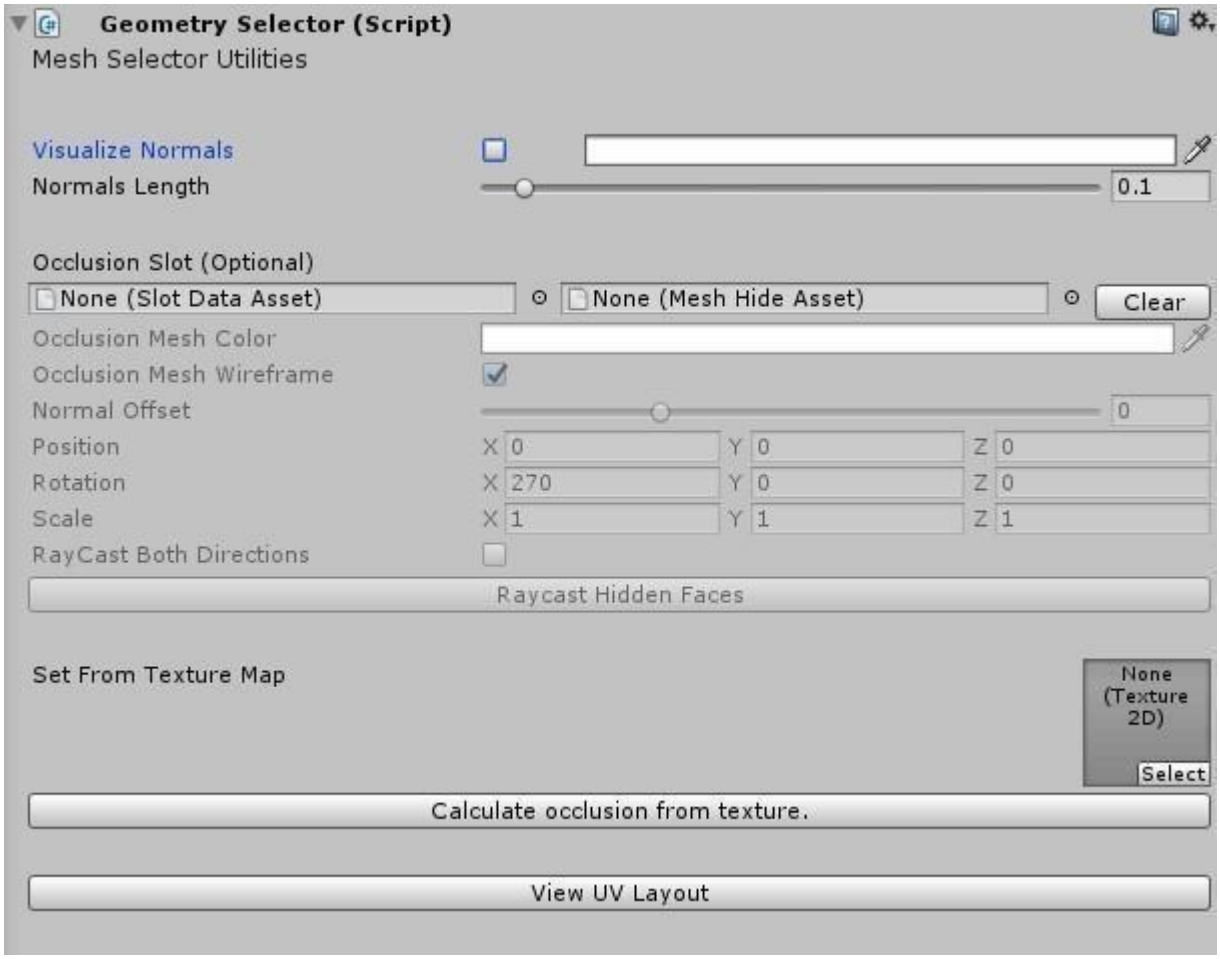
Pressing “Clear” will deselect all of the polygons. Pressing “Select All” will select every polygon. Invert will invert the selection status of all polygons (selected polygons will be unselected, etc.)

You can click on the mesh to select or deselect a polygon. You can drag a bounding box around polygons, and it will select or unselect based on the Selection Mode. You can hold down the SHIFT key to “paint polygons freehand” using the current Selection Mode.

X Symmetry is an experimental drawing mode that works well when you have X mirrored meshes. It attempt to automatically select polygons on the opposite side left/right when selecting or painting polygons.

Press “Save & Return” to finish editing the Mesh Hide Asset and return to the previous scene.

You can choose to automatically select occluded polygons by selecting a an occlusion slot in the inspector. Press “Raycast hidden faces” to detect and select occluded polygons. Sometimes the occlusion slot can be rotated or offset incorrectly. You can adjust the transform of the occlusion slot in the fields provided:



Bone Poses and Physiques

The new Physique slot is intended to allow you to generate and apply your own physiques. To create a Bone Pose Set, right click in your project, and select ***UMA/DNA/Physique Bone Pose Set***. Be sure to name your new set something unique (for example: “Trollface”). This gives you the option to create a wardrobe recipe as well as add the generated assets to the global library. After generating, you will need to define the Bone Pose by selecting it in the project, and editing it in the inspector. To edit the pose, you must be running a scene. Select anything with a UMAData (for example, a DynamicCharacterAvatar), and drop it into the ***Source UMA*** field.

UMA Material

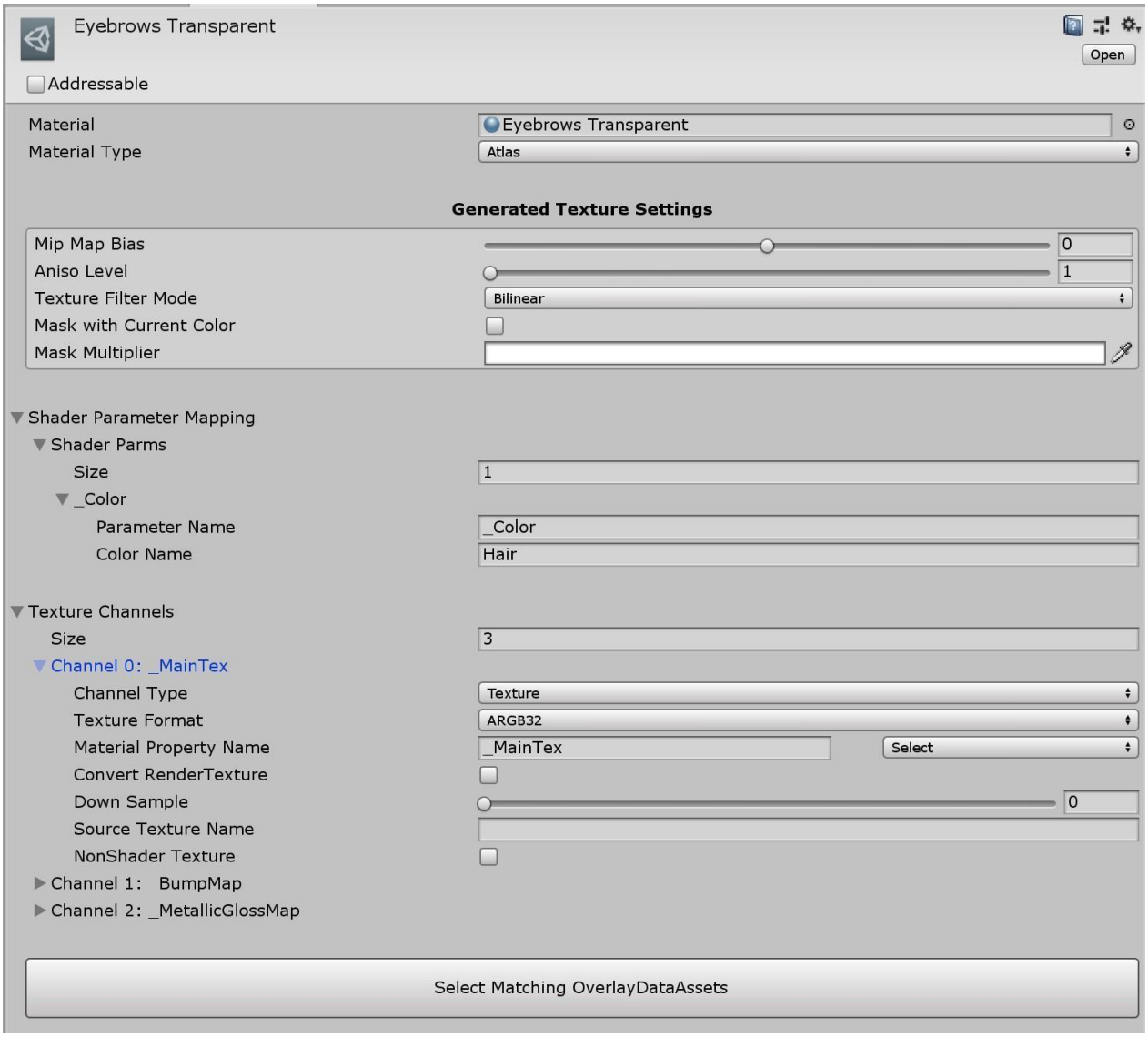
The UMA Material asset object is used by Slots and Overlays. This asset is a wrapper for Unity Materials so that the UMA system can properly group identical materials to be atlassed and merged. An UMAMaterial can be created for any type of material.

There are several prebuilt UMAMaterials for most of the commonly used material and shaders, located in UMA/Content/UMA_Core/HumanShared/Materials.

For example, in the "standard" folder;

"UMA_Diffuse_Normal_Metallic" - Unity Standard Shader that accepts albedo, normal, and metallic/roughness texture.

"UMA_Diffuse_Normal_Metallic_Occlusion" - Unity Standard Shader that accepts albedo, normal, metallic/roughness, and ambient occlusion texture.



Material: The template material used to generate the end material.

Material Type: When the Material type “Atlas” is selected, The UMAMaterial is used to determine what items go together on an atlas, and to specify the unity material that is used to generate the material for the atlas.

Generated Texture Settings: You can adjust the various setting for the generated textures. Mask with current color specifies what color is used as the base color for alpha and cutout

albedo textures. This can give a better result when generating hair textures, etc. Mask Multiplier is used to darken the masked texture to better match the source textures.

Shader Parameter Mapping:

This is used to map shared colors to shader parameters in the generated materials. The shared color must be present in the recipe. The current value of the shared color is used during the build process to update the material. This can be used, for example, to set color parameters for skin and hair shaders.

Click here for more information on UMA Materials

UMA Simple LOD

UMA comes with a Simple LOD solution. This solution can do both texture and slot level LOD. Parameters for the UMASimpleLOD script are:

LOD Distance	The distance to step to another LOD. The default is 5.
Swap Slots	Check this to use Slot level LOD also.
LOD Offset	Used to offset the LOD chosen by distance for slots. Subtracted from the calculated value. For example, if this is set to 1, then LOD level 1 for slots will not be chosen until actual distance is 2*LOD Distance.
MAX LOD	The lowest available LOD to look for if an LOD slot is not found.
Max Reduction	The maximum scale reduction for the Atlas textures. (The default, 8, means the texture can be reduced in half 8 times)

Texture LOD is automatic. Each time the avatar is a multiple of the LOD distance, the atlas texture is reduced in half. For example, at LOD level 3, The texture is reduce to 1/8th the maximum size.

Slot LOD happens only if the “Swap Slots” option is checked. This requires that you pregenerate slots for the various LOD levels, reducing polygons for each level. The number of LOD levels are variable. If an LOD level is not found, it will use the previous level.
The slots are found in the library by name, identified by the LOD level assigned. For example, if you had a slot named “vest” and you wanted two LOD levels for it, you would create two additional slots, named “vest_LOD1” and “vest_LOD2”. These should be in the global library, or an asset bundle.

Texture LOD and Slot LOD happen at mid-level. The entire character is not regenerated from base race and wardrobe slots. Instead, the existing recipe is updated, and the character is reset to generate. This is faster than generating the character from scratch.

Low Level access

DynamicAvatar

It is possible to use UMA at a lower level, using a DynamicAvatar. A DynamicAvatar does not have a base race recipe, wardrobe, or shared colors. This avatar is used to construct a character straight from a UMATextRecipe. To change the Avatar, you can access the low level slots and overlays on the UMAData, or you can simply create a new avatar and provide a new recipe.

Text Recipes

Text Recipes for the DynamicAvatar are created by right clicking in the project, and selecting Create/UMA/Core/Text Recipe. This will create a UMATextRecipe. You set the “race” on the slots tab, after which you can press the “Add DNA” button to copy the DNA to the avatar to adjust. Since this is for the DynamicAvatar, you will have to add all the parts of the race yourself – they are not copied from the base race. Add the slots and overlays, and set the colors. Once you have added DNA, you can switch to the DNA tab, and modify the DNA for the character also.

Place the recipe in the DynamicAvatars “UMA Recipe” field. You can also add additional utility recipes – such as a capsule collider recipe, and attach a runtime animator controller.

Content Creation

Animations

UMA does nothing special with animations so any generic or humanoid animations can be used with the appropriate rig.

Clothes

Making new clothes for a race involves taking a mesh, skinning it to the target character’s skeleton, then importing it in to unity and creating slots, overlays, and a wardrobe recipe for it. The first step will be very dependent on the modeling software you use.

-Common across any modelers though, you’ll want to import your target race or character mesh into a scene.
-Deform your cloth to fit how you want it to fit to your race in its neutral position.
-Add a skin modifier to your cloth and skin it to your race’s skeleton. A skin wrap is a good tool here.
-Finally, export the whole skeleton and the cloth mesh as an FBX to import in to unity.

Races (Base Mesh)

A whole new race can range from simple to complex depending the features to support.

Without supporting runtime bone deformation for a race, then a skinned mesh with a valid humanoid skeleton is almost all that is needed.

Both a “unified” version and a “Separated” version of the race are recommended (though not needed). The unified version is a single connected mesh. The separated version is with cut, separate mesh for individual body parts that can be set later in uma to be individually hidden or not. For example, separate mesh for head, torso, hands, legs, feet, etc…

The unified version is then used when building slots as a “seam” mesh, which means the normals from it will be used instead of from the individual cut up meshes. The cut up meshes tend to distort the edges of the mesh and then will not look correct when lined up with their neighbors.

To create a valid new race, you will need to create a RaceData asset, a base recipe for that race, and a T Pose asset.
The base recipe is a standard TextRecipe of all the race’s default slots and overlays. This will be added to the corresponding field on the RaceData.
The T Pose asset is extracted from an FBX of this race with it’s full skeleton. After configuring the Mecanim Avatar that is standard in Unity, then you can use the UMA dropdown function “Extract T Pose”. This will create a new T Pose asset for your race that you can add to the corresponding field.

Click here for more information on RaceData
Click here for more information on TextRecipes
Click here for more information on TPose assets

Using Blender to create content

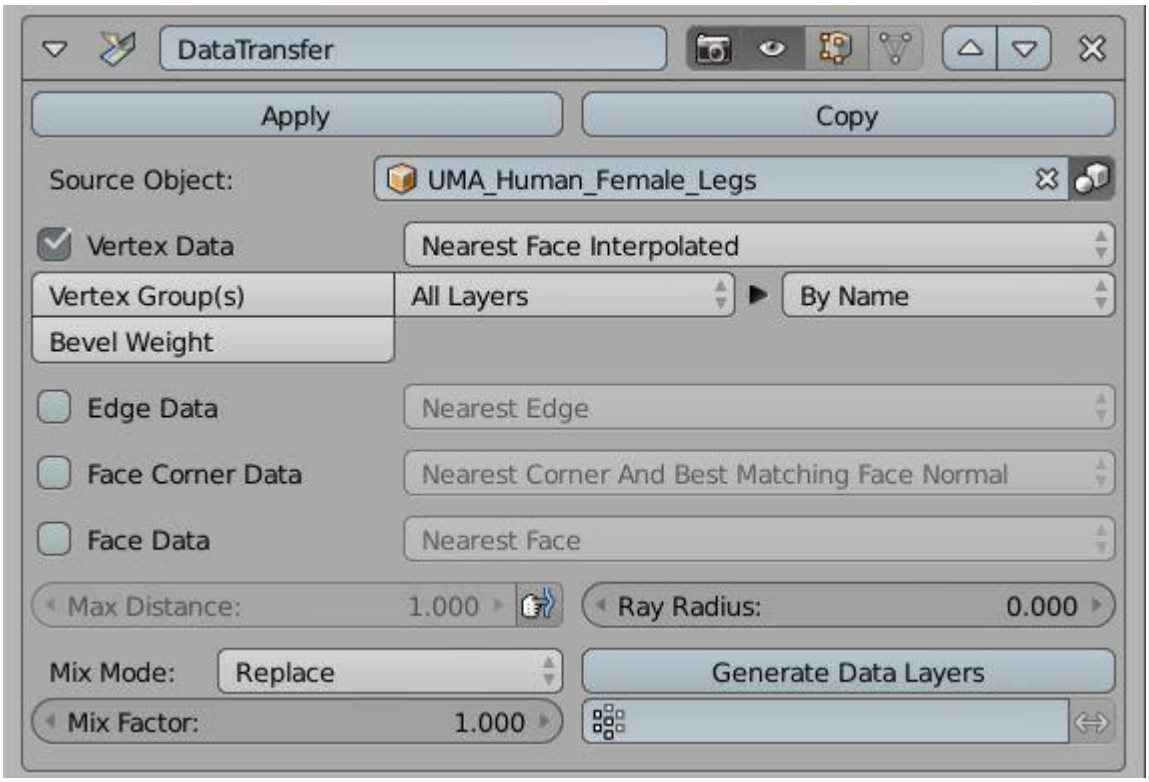
First, download the blends from the content pack repo here:

https://github.com/umasteeringgroup/content-pack/tree/master/ContentPack_1.1.0.1/Blend
UMA_blender.blend contains the unified models (both male and female). It is recommended to use the Blender 2.8 specific blends for Blender 2.8+

UMA_Blender_Separated.blend contains the models that are separated into their slots.

Depending on what you want, either one will work for creating clothing.

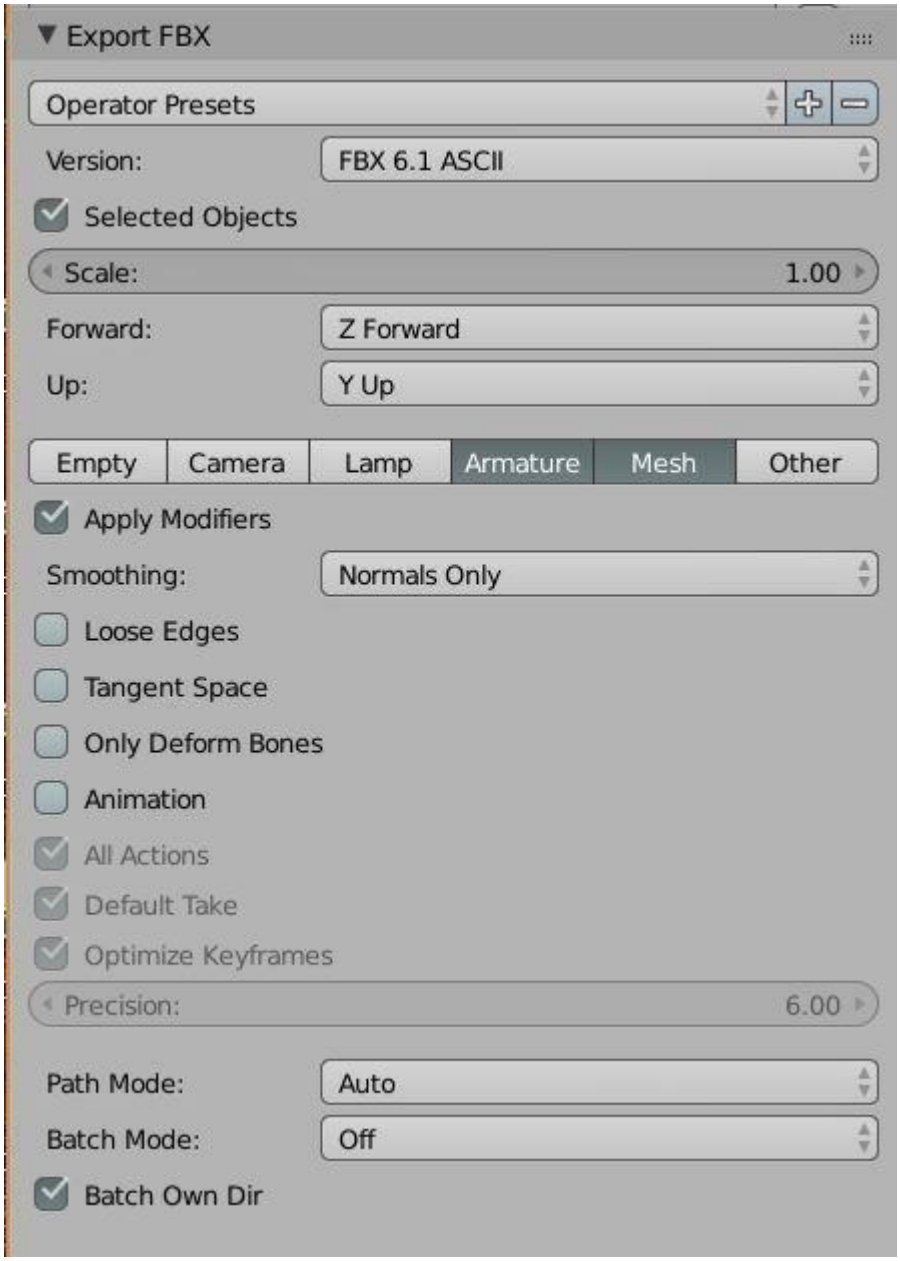
When rigging your clothing, it's simplest to fit it to the specific model, then copy the rigging information using a "Data Transfer" modifier. After fitting the model, add the modifier, select the source model (the original UMA mesh), select Vertex (you want to copy per vertex data), and "Nearest Face Interpolated". The select "Vertex Groups". Make sure Mix Mode is set to "Replace" and press the "Generate Data Layers" button. Then press "**Apply**". (if you don't press Apply - your weighting will be lost). Your clothing should now be rigged. Of course you may need to tune the rigging using Weight Painting.



Export Settings for Blender 2.79

You'll export your clothing to FBX using the built-in fbx exporter. Make sure your clothing has been rigged, and has an armature modifier (with the source set to the correct rig). Do NOT apply the armature.

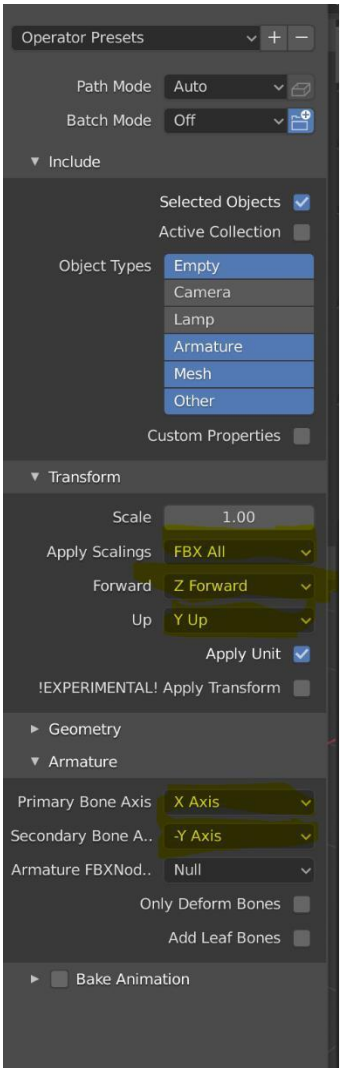
Then export to an FBX with "Forward" set to "Z Forward" and "up" set to "Y Up". Only the armature and mesh need to be exported.



Import the FBX into Unity, and open the Slot Builder.

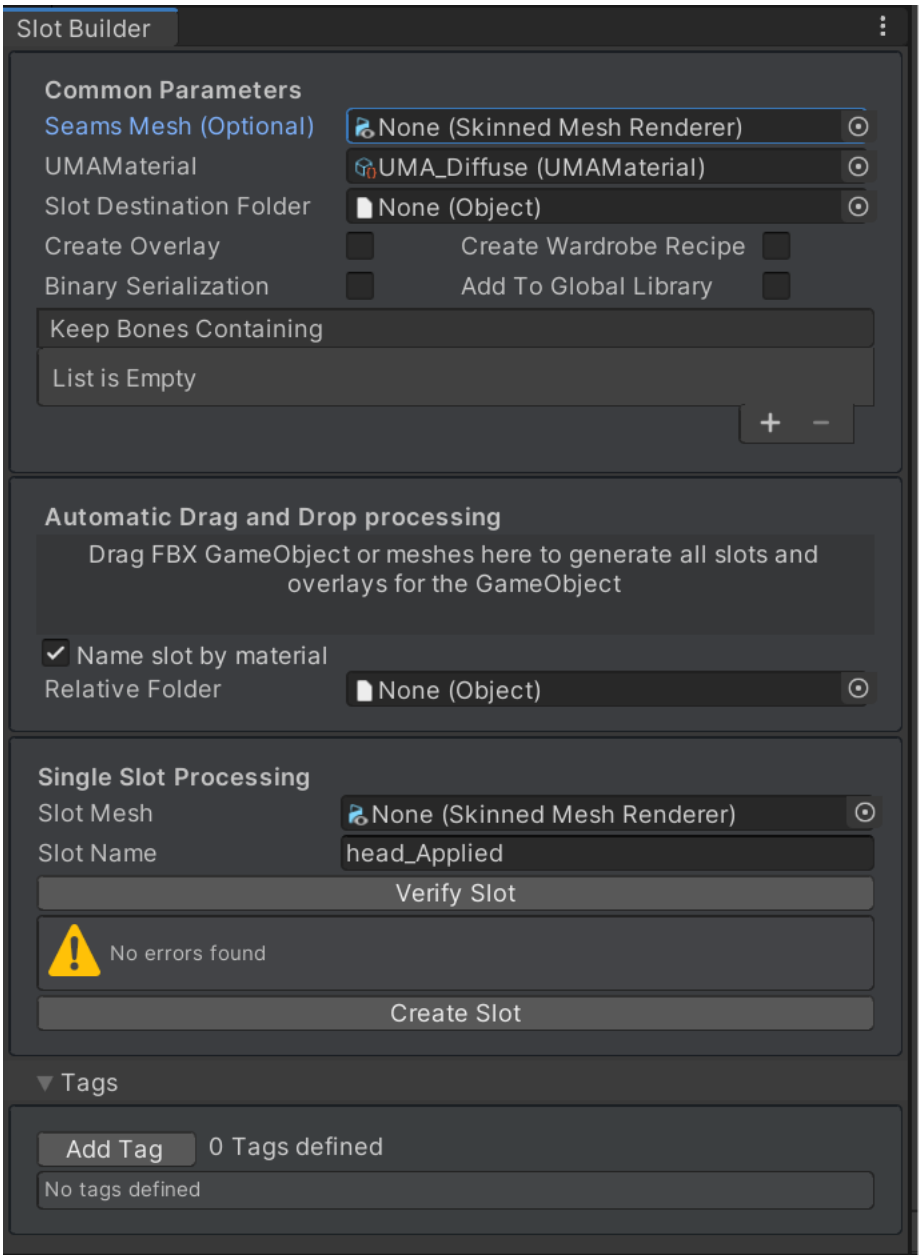
Export Settings for Blender 2.9+

The following settings are used for Blender 2.9+. Be sure you have selected both the armature and the mesh before selecting to export the FBX.



When using these export settings, you should not have to update the scale parameter in Unity.

Using the Slot Builder



Open the slot builder using the UMA/Slot Builder menu item and dock it somewhere. The Slot Builder is used to create “Slots” from the mesh (Slots are mesh parts that are used to construct the Skinned Mesh Renderer). You can either use the slot builder in “automatic” mode, or you can build slots one at a time.

Automatic Mode

To use in Automatic mode, fill out the “common parameters”, Check if you want the slots named by the material or not (if not, they will be named by the mesh), and then drag/drop the FBX

model onto the pad in the "Automatic Drag and Drop Processing" section. This will create slots (and optionally, overlays and wardrobe items) for each skinned mesh in the FBX, each in a separate folder under the "Slot Destination Folder".

Name Slot By Material	Bool	If this is selected, the slot will be named by the material, instead of by the mesh name.
Relative Folder	Folder	When slots are named by material, this folder should contain the location of the materials used.

Single Processing Mode

To use the slot builder, you'll fill out the common fields, drop the Skinned Mesh (under the FBX object) onto the "Slot Mesh" field, Set a Slot Name, and press the "Create slot" button. A folder with the slot name will be created under the "Slot Destination Folder" named after the slot, and the slot and other items will be placed in that folder.

There are two buttons used in Single Processing mode:

Create Slot: Press this button to create the slot and any optional items. The folder will be created, the items generated and optionally added to the library. Depending on what you've selected and the size, it could take several seconds to complete.

Verify Slot: This is used to verify that all UV coordinates are in the correct range (0-1). Textures that are atlassed cannot use tiled textures.

Fields

Slot Mesh	Skinned Mesh Renderer	This is the actual mesh data. Expand your fbx in the project view, and you'll see multiple components below it. Select your clothing mesh, and put it in the Slot Mesh field. You may see a warning that it's too small - if so, go to the import options for you fbx, and set the scale to 100 and apply it, and then drop it on there again.
Slot Name	String	This is the name you want to give the slot. It is used when creating the folder, the slot, the overlay and wardrobe item, if selected.

Common fields

Seams Mesh: The “seams mesh” is used when you want to fix the normals when you’ve split your mesh into multiple pieces. That’s not used that often, so ignore it for now.

UMAMaterial: This is the material you want your slot(s) to use. Press the selector button (the small circle to the right), and select the material. Most items use the “UMA_Diffuse_Normal_Metallic” material, But you can select any one you want that you need. Slots that share a material have their textures built into the same texture atlas.

Slot Destination Folder: This is the location where UMA will generate folders to contain your new slot(s) and overlay(s)/recipe(s), if selected. The new folder for your slot will be named the same as the generated slot name.

Create Overlay*:* Check this if you want an empty overlay created for your slot(s). Once created, you would need to select it in the project view and add your textures and material to it. The overlay will be named ***{element name}_Overlay***.

Binary Serialization: Check this to force the system to use Binary Serialization to create the slot. This is much faster to load, but not so friendly for some source control systems.

Create Wardrobe Recipe: Check this to create an empty wardrobe recipe for your slot(s). Once created, you will need to add your slot(s) and overlay(s), and do the normal setup.

Add to Global Library: Check this if you want all of your new items added to the global library. If you forget this or something goes wrong, you can just drop the entire folder onto the global library to add them.

Keep Bones: Normally, UMA will discard bones from a slot that aren’t in use by that slot. This speeds up building, and makes the slot a little smaller. However, sometimes you need to place bones into the rig for the slot that are used for physics, or for helpers (mount points, etc). To keep UMA from discarding these bones during the slot building process, add them to the “Keep Bones Containing” list. The system does a string.contains to see if the bone should be kept. For example, if you end all your helper bones with the text “_Helper”, then you can add “_Helper” to the “Keep Bones Containing” list to make UMA retain those bones when building the slots.

UMA Addressables

Overview

As of UMA 2.10, raw asset bundles are no longer supported automatically. If you want to use raw asset bundles, you should download the bundle, read all the items from the bundle, and add them to the Asset Index using ***UMAAssetIndexer.Instance.ProcessNewItem(theItem);*** Items have to be processed before they can be used in any character.

Instead of raw asset bundles, UMA 2.10 uses the Addressables system to manage asset bundles. Using this system, it can manage the load and unloading of assets as needed.

What Happens When An Avatar is built.

A DynamicCharacterAvatar requests assets when it is built using BuildCharacter(). If the character has “BundleCheck” enabled (default is on if you are using Addressables), then after the recipe is generated and the list of slots and overlays are known, it gathers the labels for the character, and passes them to the addressables system for loading asynchronously, and then returns. Once the assets are loaded, the build process schedules the UMA to be generated, and the previously used items are unloaded, freeing up memory (assuming no other UMA has a reference to them). This saves an enormous amount of memory.

Setting up the system to use addressables

If you are not already setup to use addressables for your application, you will need to do so now. Open the package manager, and import the addressables package. You must import ***Addressables 1.6*** or greater. Once the package is imported, open the “Addressables groups window”, and it should have a button on it to generate the addressables data. Push the button to generate the data.

Once that is done, you should make sure you have added all of the recipes you want to use to the global library, and then generate the Addressables Groups from the Addressables menu in the global library window. Use the generate single group option during development.

Once the groups are generated, you should switch back to the Addressables Groups window, and build your bundles, and select the playmode. For now, just build your bundles with Build/New Build/Default Build Script. Set the playmode to “use asset database”.
As you get farther in development, you will probably want to change that to “Use Existing Build” so you can test that your groups and bundles work correctly.

Optimizing the usage of addressables

Caching remote items: You can download items from your server using the

Addressables.DownloadDependenciesAsync function. See the “Preloader.cs” script for usage. This will cached the items locally, so they are faster to load.

Preloading items for faster access. You can preload items into memory using UMAAssetIndexer.LoadLabelList({labels}); This will load the items into memory, and add a reference to them in the Global Library. This function returns an AsyncOperationHandle<Object> that can be used to determine when the load is complete, and can also be used to unload the items if needed. If you have default recipes that are always in use, you might want to preload them by passing them in the list of labels.

Disabling BundleCheck. If you have preloaded everything you need, you can skip the bundle check on the character by unchecking “Bundle Check”. This will immediately schedule the character to build instead of delaying while the items are loading. You must ensure that ALL items used by the character have been loaded.

Creating a clean UMA install

To create a clean, production ready UMA install, it’s best to have 2 projects. One project will hold the full UMA install and the other the clean one.

Essential UMA folders

First copy the essential UMA folders from the full install to the clean version. These folders are:

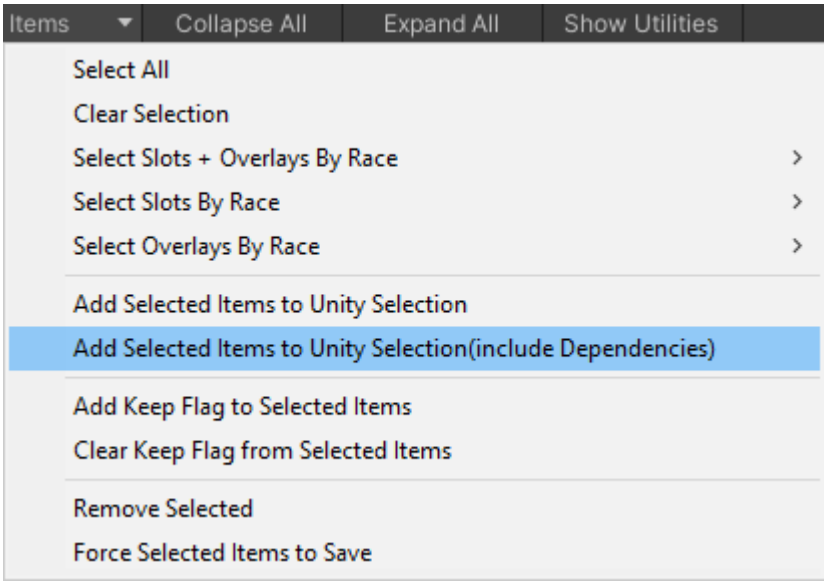
- Core
- Editor
- Getting Started
- Internal Data Store

Export Races and Recipes

Next, go to the full UMA install and export the races and recipes with their dependencies.

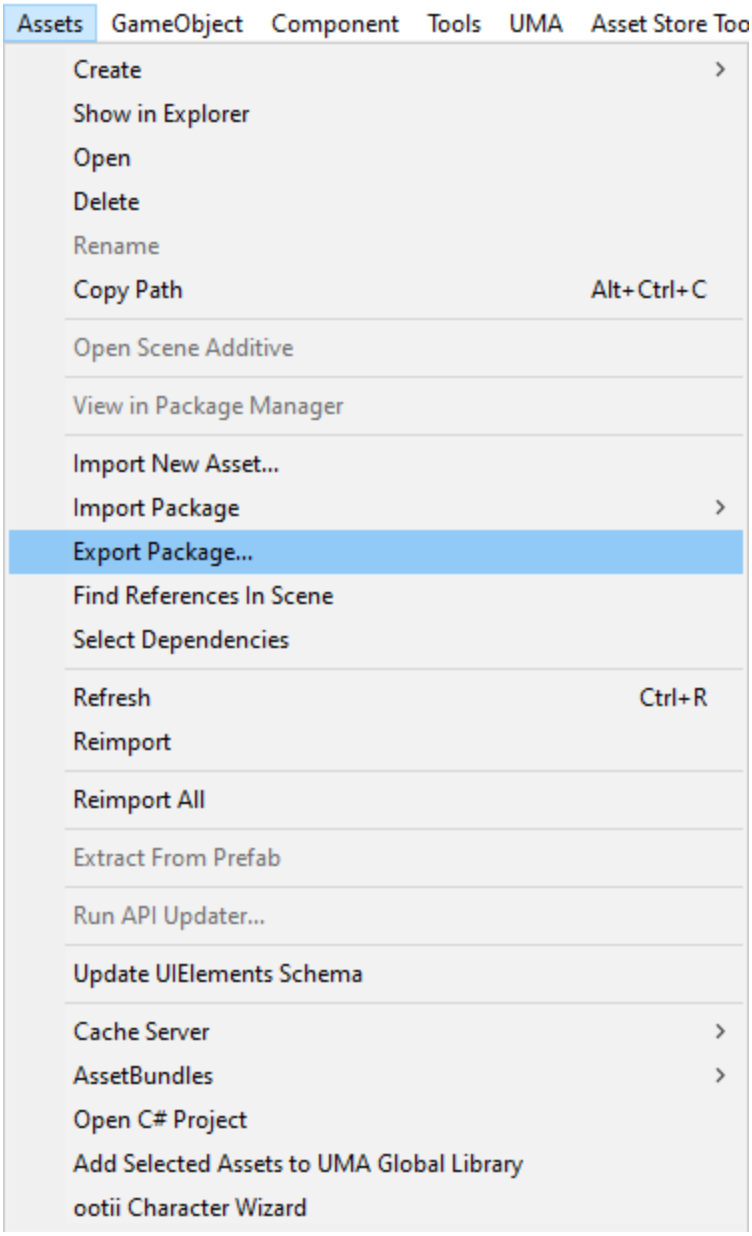
Select the race and other recipes you want to export. For example the HumanMale selected below:

Then go to Items -> Add Selected Items to Unity Selection (include Dependencies).

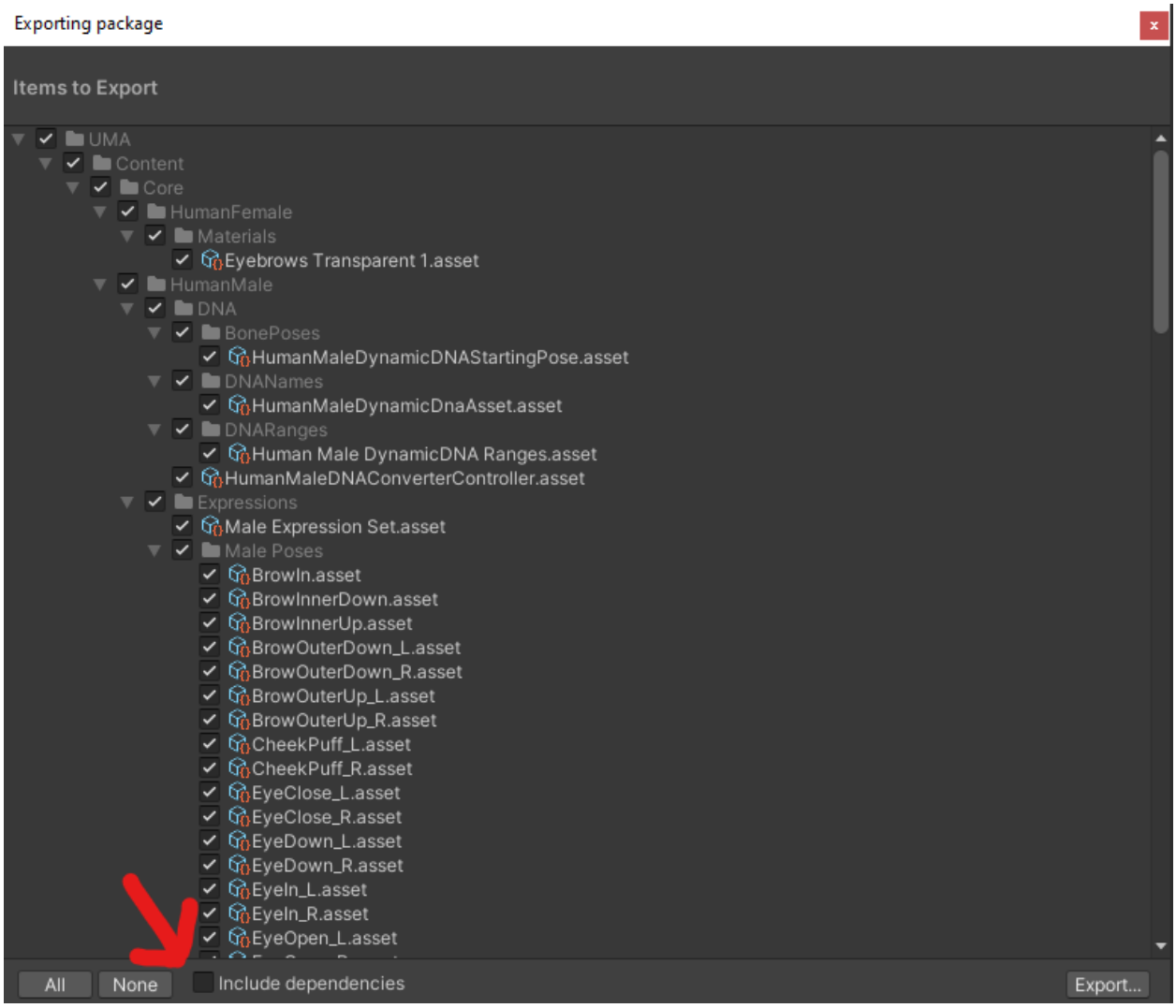


This will select all of the dependencies that the recipe needs to run.

After that just export it as a package.



Make sure to uncheck “Include dependencies” and then export the package.



Export Races and Recipes into Clean Project

The final step is to import the exported recipes into the clean project and then you will have a clean installation of UMA with just the recipes you need and none of the examples and other unnecessary files.

Useful Links

Source Code

[Stable Development Version]

This branch will contain tested but possibly not release ready content.

Tutorials

[Secret Anorak's UMA101]

Secret Anorak's Content Creation
Secret Anorak's Race Creation
Casey's Dynamic Character Creation Tutorial
TheMessyCoder UMA Tutorial
WillB GameArt Tutorials

Discord

Invite to Secret Anorak's Hideout

Content

Will B Game Art
Arteria3D UMA Section
Unity Asset Store
Github Base Content source o3n Web Store o3n assets consist of a custom male and female race along with content compatible with those races. There are also some assets which are ported for the base UMA races. Please note that they cannot be used on base UMA races unless it is stated in the asset description.