Documentation

Github Wiki Code Documentation

Contents

What is 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.

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 amount 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.

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 an unique mesh and rig and is individually skinned and textured.

When developing games that might require a huge amount 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 an 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.

On UMA latest version, the old CombineInstance() code is still available in case there is no intention on using extra bones that would be dynamically included on avatar. The legacy code requires less processing time, but provides limited results.

How is Content Created?

See the Content Creation Section

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.

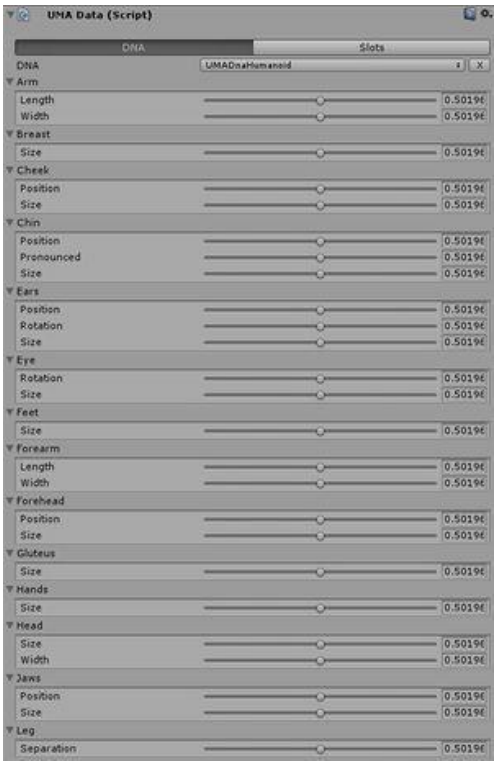
2) Find the prefab "UMA_DCS" and drag this into the scene. This is the object that contains all your libraries, context, generator and mesh combiner objects.

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. Hit play and you should see the default male get created.

5) While playing, you can select the avatar object and see the dna in the "UMAData" component that now exists on it. You can adjust the values to see immediate changes.

6) From here you could change to other races or add new wardrobe items along with their necessary slots and overlays.



Basic Concepts

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 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.
Click here for more information on Wardrobe recipes

Setting in script

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)

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

Libraries and Global Library

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

In version 2.5, a “Global Library” was added. This library is not scene specific, so items added to this library are available across all scenes. The global library is accessed by the “Dynamic” versions of the libraries - DynamicOverlayLibrary, DynamicSlotLibrary, etc. To access 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), whether it is broken, and whether it has a build reference. 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. (Note: the Dynamic Libraries can also find items stored in Asset Bundles - See below).

Note that all items in the scene libraries and global library contain a reference to the item, and that will cause those items to be included in your build. The global library allows you to clear the references (for performance reasons). Once removed, however, the references must be re-added when you build your application or the different UMA assets will not be included. This is done using the “Add Build References” button in the library. If any items are not in the build, you will see a notification of this in the library.

Race

A "race" in UMA is nothing more than a specific TPose and set of DNA converters. 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.

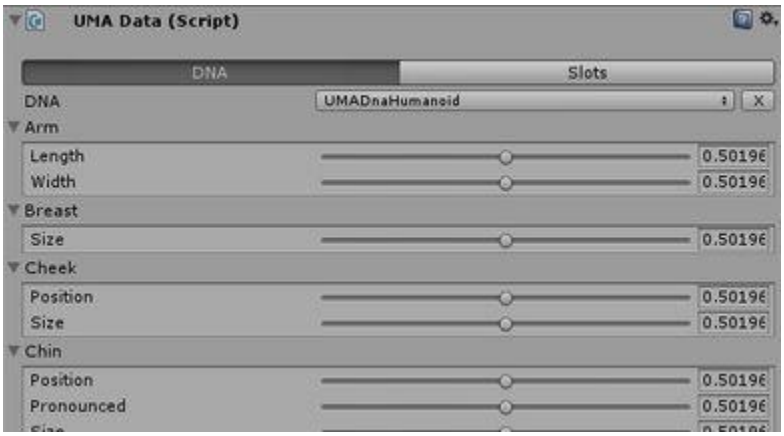
The term "race" can be confusing in UMA. It is used for avatars that share the same base mesh and DNA. For example, if you have a dwarf that is created from the human base mesh simply by changing the dna to be shorter and stockier, then it will still be the same "race" as the human, in UMA terms. This 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.

These individual pieces can 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

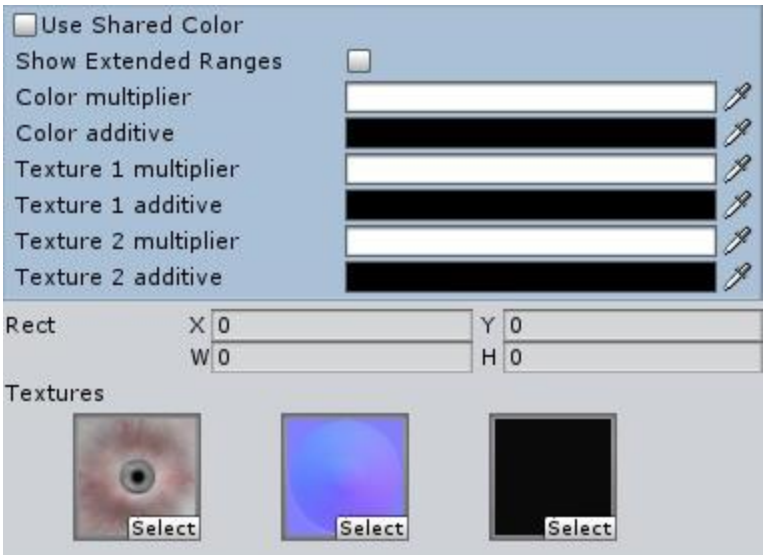
Overlays

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.

Click here for more information on Overlays

UMA Material

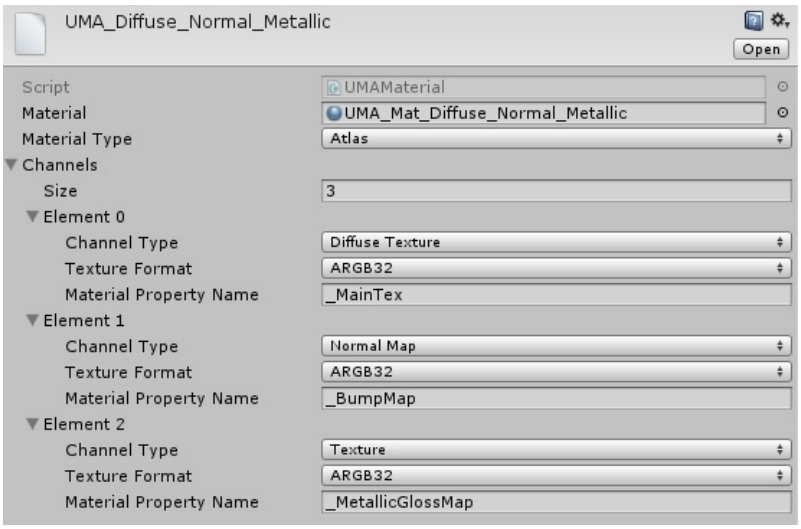
The UMA Material asset object is used by Slots and Overlays. This asset is a wrapper for Unity Materials so that the UMA system can properly group identical materials to be atlassed and merged. An UMAMaterial can be created for any type of material.

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.



[Click here for more information on UMA Materials]

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 creatings slots, overlays, and a recipes 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

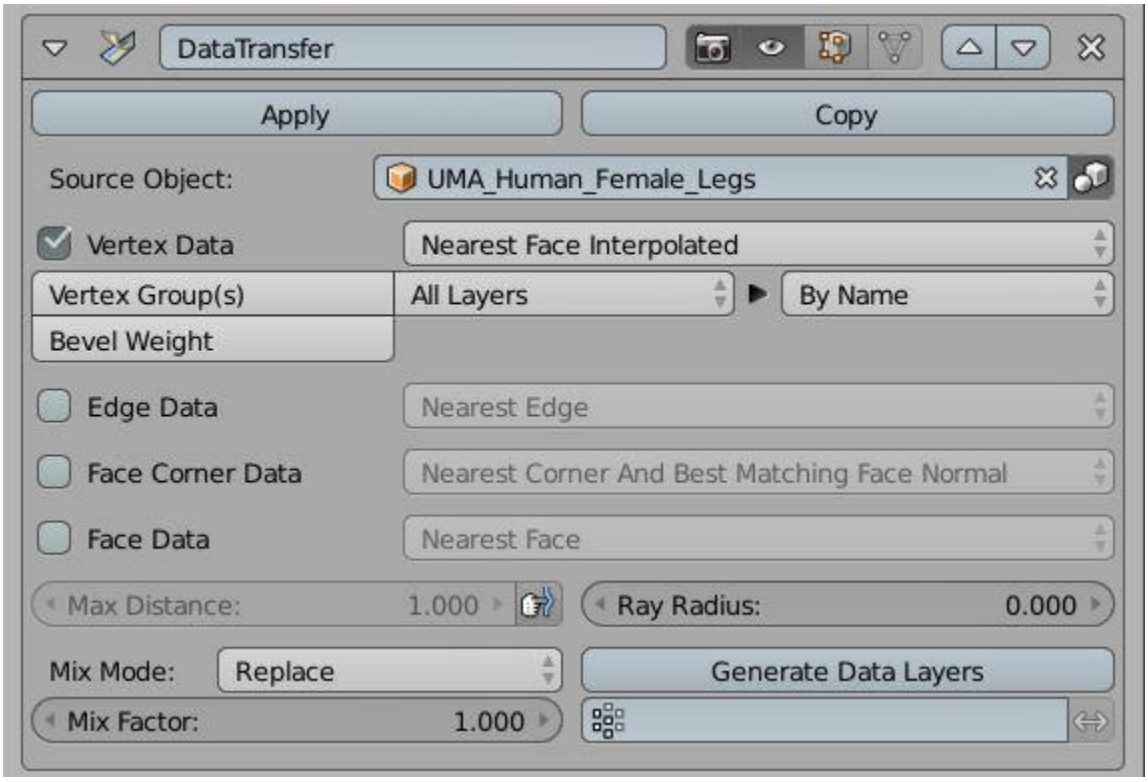
First, download the blends from the content pack repo here:

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).
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.



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.

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 the “Slot” from the mesh. To use the slot builder, you’ll fill out the fields and press the “Create slot button.

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.

Slot Mesh: 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.

UMAMaterial: This is the material you want your slot 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 a folder to contain your new slot (and overlay/recipe, if selected). The new folder for your slot will be named the same as the element name (see below).

Root Bone: Leave this as "Global" in 99.9% of cases. The only time to change this would be if you had a different skeletal hierarchy, and it was compatible with UMA.

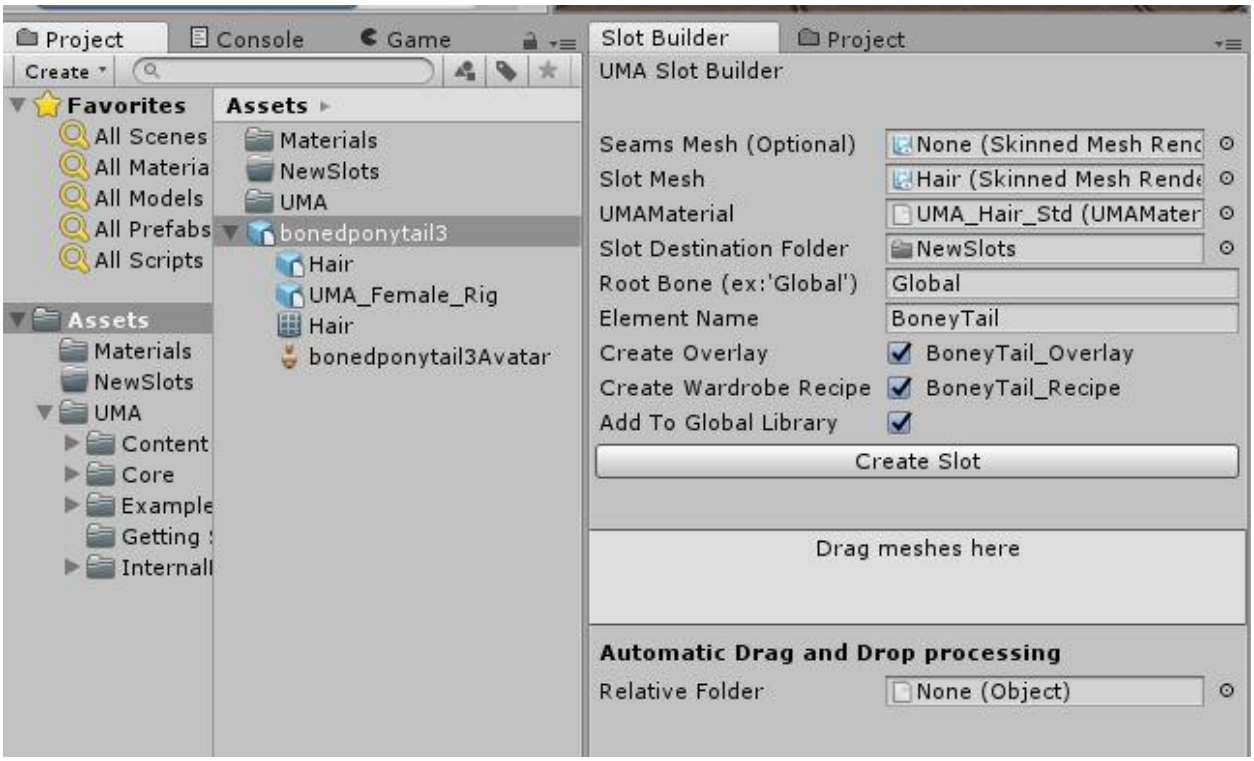
Element Name: This is the "name" of your slot. It's also used as the base for naming all the new files that are generated.

Create Overlay: Check this if you want an empty overlay created for your slot. 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***.

Create Wardrobe Recipe: Check this to create an empty wardrobe recipe for your slot. 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.

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.



UMA AssetBundles

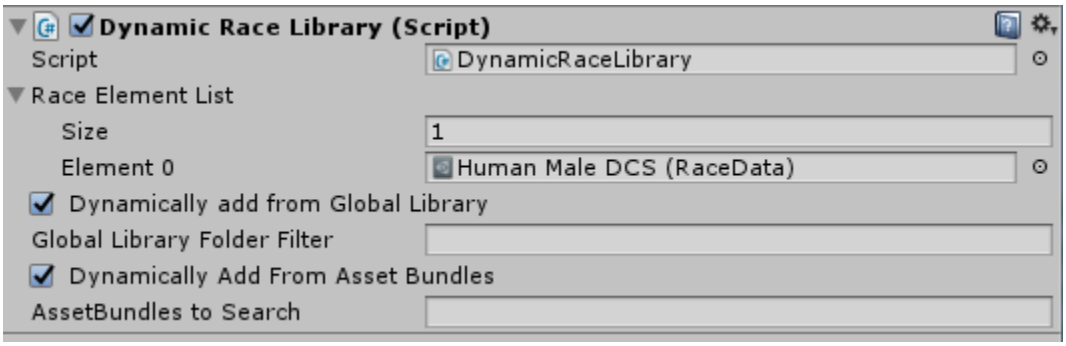
Overview

As of UMA 2.5 your UMA content can be assigned to and retrieved from AssetBundles. The DynamicCharacterSystem was built from the ground up with AssetBundles in mind and as a result, loading recipes or races from assetBundles is very similar to loading them from the Global Library, in that its is not required that you do anything more than make the DynamicCharacterAvatar request a given recipe in order for it to be downloaded and applied to the Avatar.

What Happens When An Avatar Requests an Asset

A DynamicCharacterAvatar requests assets when it is first built or when its Wardrobe or Race changes. The latter happens when you make requests to ‘ChangeRace’, ‘SetSlot’ or one of the Load methods. ‘ChangeRace’ and ‘SetSlot’ have overload methods that take a string parameter (the raceName or name of the recipe file respectively) so that you can request loading of assets from assetBundles (where you may not have the actual asset available to send as a param)
In either case the DynamicCharacterAvatar makes a requests to the DynamicLibraries (Race/Slot/Overlay/DynamicCharacterSystem) for the assets it requires to build itself.

If the DynamicLibrary already contains the requested asset it simply returns it. Otherwise it asks ‘DynamicAssetLoader’ to find it. You can determine whether a given Dynamic library tells DynamicAssetLoader to search AssetBundles as well as the ‘Global Library’ on the settings for the Library itself. For example on the DynamicRaceLibrary:

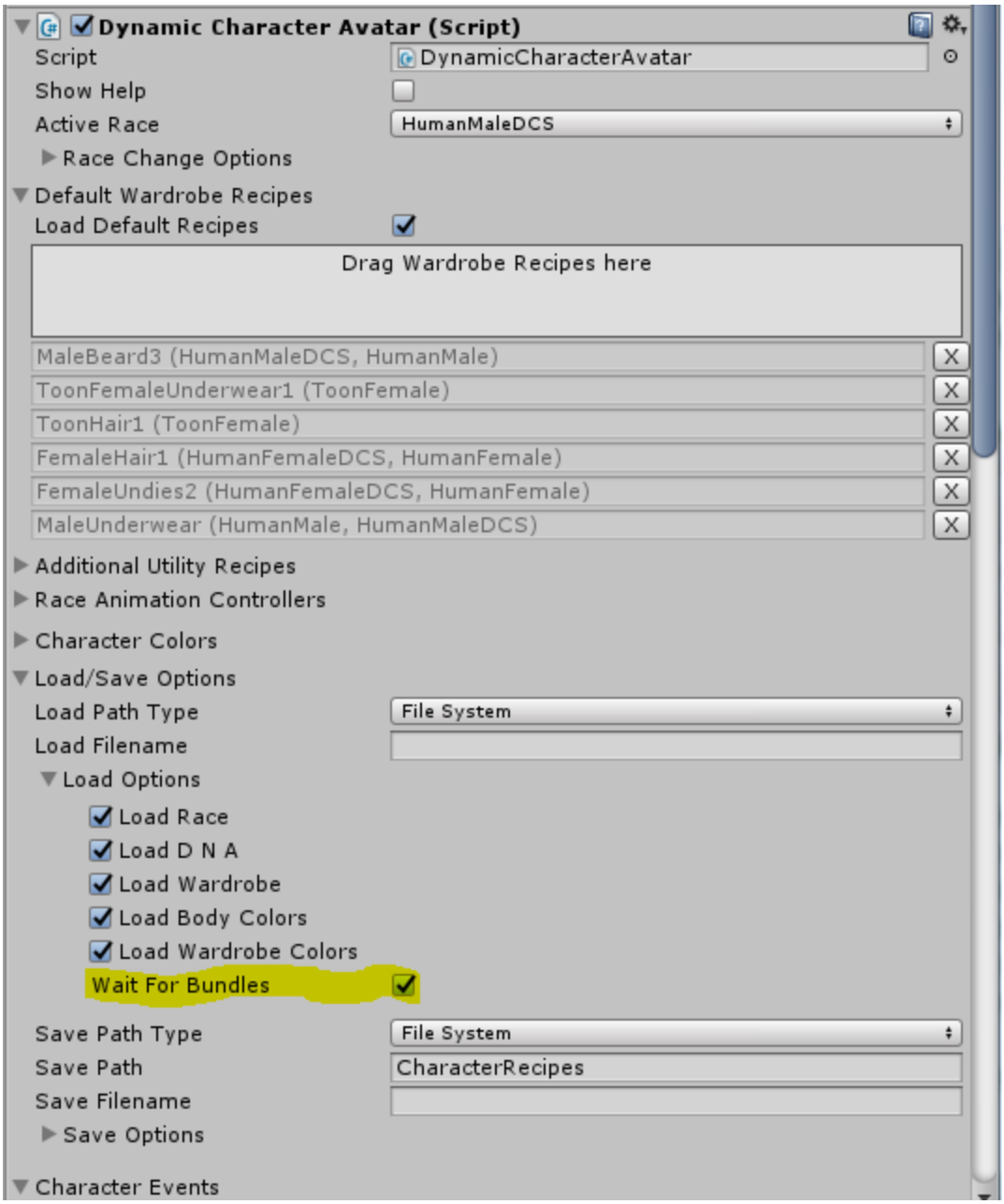


You can optionally filter the bundles that get searched by entering the assetBundle names to search seperated by commas. The filter will search for bundles that start with the assetBundle name, so if for example you have bundles called ‘uma/humanmale/textures’ and uma/humanmale/slots, typing in ‘uma/humanmale’ will cause both bundles to be searched. Leaving the field blank will cause all bundles to be searched. Searching is fast because we use an AssetBundleIndex to index which assets are in which bundles (more on this later)

If the asset is in the GlobalLibrary DynamicAssetLoader will return it to the library, which will in turn return it to the DynamicCharacterAvatar.

If the library allows it and the requested asset is not in the GlobalLibrary, DynamicAssetLoader will check if it is in any AssetBundles, determine which assetBundle the asset resides in, and make a request to the included AssetBundleManager for the assetBundle to be downloaded. It returns a placeholder asset to the requesting library, and the library returns this to the DynamicCharacterAvatar. It also makes a note, that the requested asset is downloading.

The default behaviour for a DynamicCharacterAvatar when it receives a placeholder asset is to wait for the actual asset to be downloaded before it builds. This is controlled by the ‘’Wait For Bundles’ checkbox in the ‘DynamicCharacterAvatar’s Load options:



If ‘Wait for Bundles’ is off the placeholder asset will be used and the UMA will continue building. The placeholder asset prototypes are located in UMA/InternalDataStore/InGame/Resources, and the idea with these is that you could customize them if you wanted, to show an avatar ‘filling up’ as it gets downloaded or something...

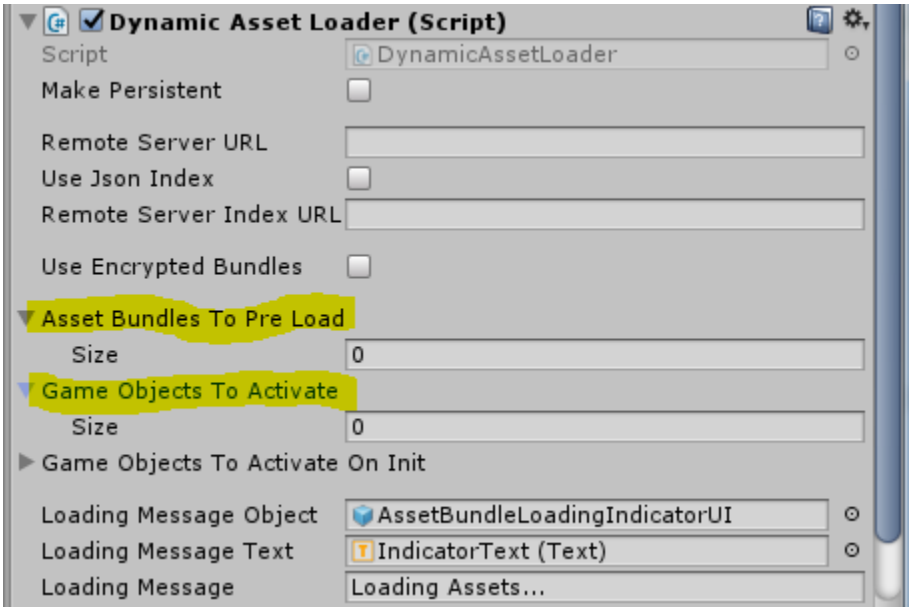
In either case DynamicCharacterAvatar polls DynamicAssetloader each frame to find out if what was requested has downloaded yet. When it has, the build process either continues (if ‘Wait for bundles’ was true) or is performed again.

Because the actual downloading of assetBundles is handled by the AssetBundleManager, any other assetBundles that the requested bundle is dependent on will also be downloaded. DynamicAssetLoader will only consider the assetBundle the requested asset is in to be downloaded, once all the dependencies have been downloaded as well.

Effectively this means that you can use assetBundles with DCA by simply calling the same methods that you would call when using the 'GlobalLibrary' - all the assetBundle loading and waiting is all handled by the system itself in the background.

Preloading AssetBundles

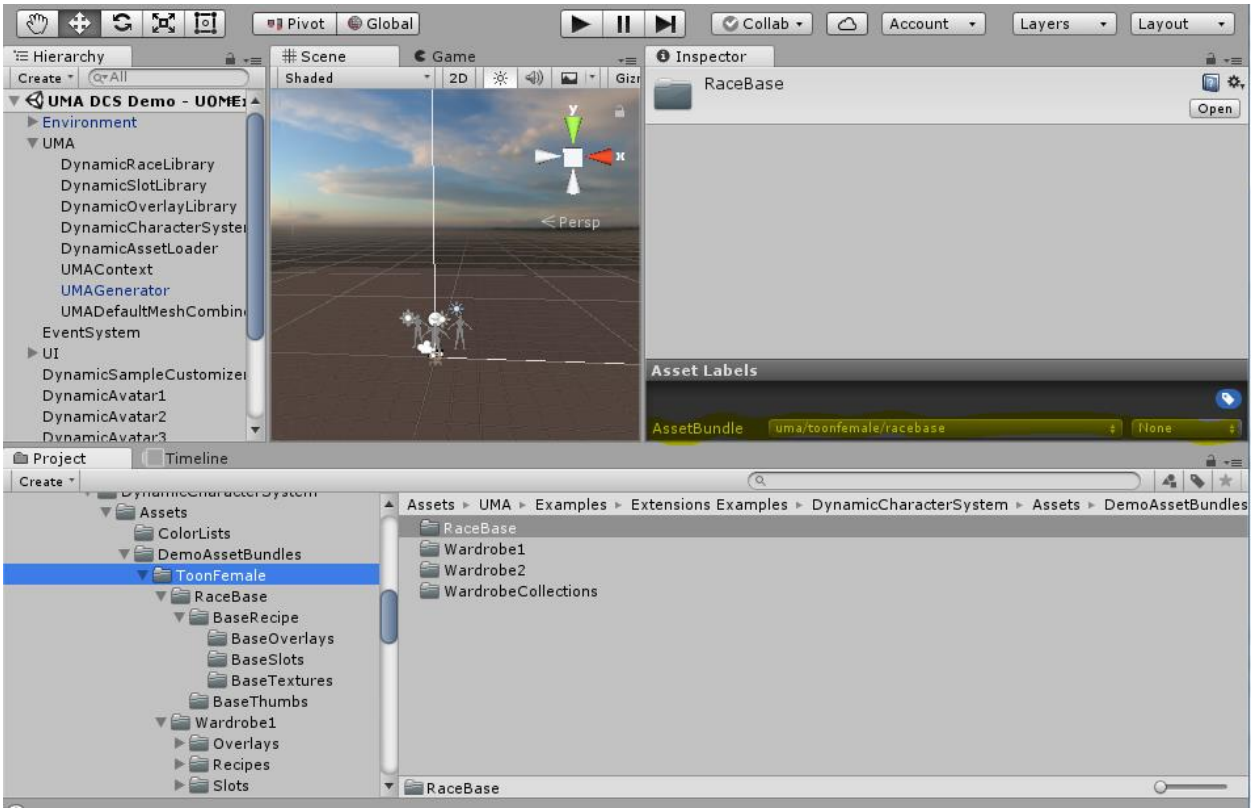
If you wish you can specify assetBundles that DynamicAssetLoader downloads when it starts:



By using this functionality you could force DynamicAssetLoader to download the base race asset bundles for the characters in your game while the game is showing a loading screen for example. Then you can use the 'GameObjectsToActivate' list to perhaps trigger your scene manager to proceed to the 'main menu' or similar.

Assigning Your Content To AssetBundles

Assigning your UMA content to assetbundles is no different to assigning any other content to asset bundles; you simply select the asset(s) or folder you wish to assign to an assetBundle in the project view and give the bundle a name. (See the Unity docs for more information on this).



One thing to be wary of though is that, if any of the assets in an assetBundle require other assets that would not otherwise be included in your game (because they are not referenced by any scripts, not in the Resources folder and are not in the GlobalLibrary), those assets will automatically be added into the assetBundle that contains the asset that requires them. This means they will be duplicated in memory and assets using the duplicates will not be sharing the same asset any more. This is not something we have done, its just how assetBundles work.

So for example if ‘uma/humanmale/overlays’ contains ‘m_faceOverlay’ and that references ‘faceTexture’ and ‘uma/humanfemale/overlays’ contains a ‘f_faceOverlay’ that uses the same ‘faceTexture’ but ‘faceTexture’ itself is not in an assetBundle, two copies of ‘faceTexture’ will be created and one added to each bundle. This will mean that the two overlays no longer share the same texture, your assetbundles will be unnecessarily enlarged and any runtime changes to the texture won’t apply to both avatars.

The solution is to assign ‘faceTexture’ to its own assetBundle, say uma/humanshared/textures’. Now both overlays will use the same texture. Both bundles will have a dependency on ‘uma/humanshared/textures’ and if that bundle has not already been downloaded when one of the overlays is requested, it will be downloaded automatically at the same time.

This is a case where bundle dependencies and the automatic downloading of them, is very helpful. But sometimes, you might find that requesting an asset, causes a bundle to be

downloaded that you were not expecting. This will almost always be because of an inter bundle dependency you were not aware of or did not intend.

Thankfully this is not a complication unique to UMA and Unity actually provides a helpful AssetBundleBrowser tool for checking if your bundles have dependencies you did not expect. You can find more information on this tool here

NOTE: It is recommended that you ***don't*** build your bundles from the AssetBundleBrowser tool but rather use the 'Assets/AssetBundles/Build Asset Bundles' menu option, or the button in the UMA Asset Bundle Manager Settings window (see below) so you can take advantage of some extra features and ensure the AssetBundleIndex is generated correctly.

Building Your Bundles

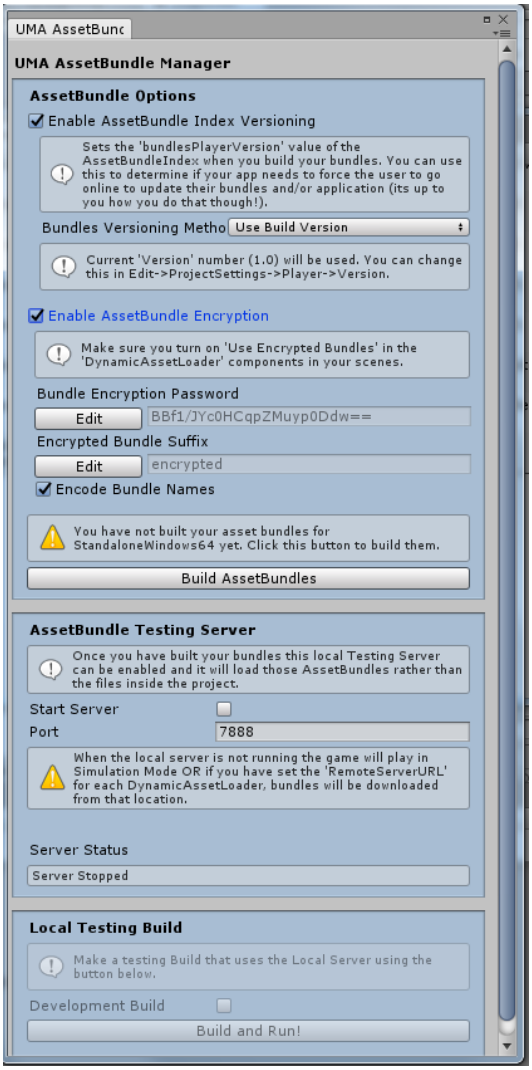
Building your bundles is as simple as going to the 'Assets/AssetBundles/BuildAssetBundles' menu option or clicking the 'Build AssetBundles' button in the UMA Asset Bundle Manager Settings window. Your bundles are built inside the project root (the same folder that *contains* 'Assets') in a folder called 'AssetBundles/[PlatformName]' It is recommended that you leave the live versions here in order that you can use the 'AssetBundle Testing Server' to test your actual bundles (more on this below)

The UMA Asset Bundle Manager Settings Window

UMA provides a modified version of Unity's 'AssetBundleManager' and its associated Build Script. This adds a couple of extra features in addition to those in Unity's version.

First of all, when building your bundles an 'AssetBundleIndex' will be created. This allows the DynamicAssetLoader to determine which asset needs to be loaded and which bundle it resides in. In UMA some assets such as slots and overlays have a 'slotname' or 'overlayname' and it is this name that is referred to in the recipes, rather than the name of the asset itself. But loading an asset, whether from Resources/GlobalLibrary or from an assetBundle requires a filename. The AssetBundleIndex solves this problem by generating a list of names and their associated files and which bundles they are in.

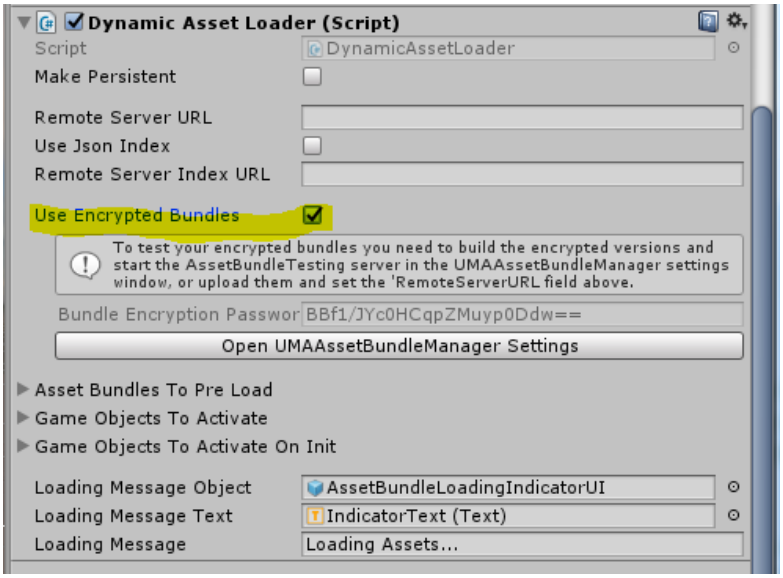
This happens automatically when building using the menu option or the 'Build Asset Bundles' button in the 'UMA Asset Bundle Manager Settings window' and no settings modifications are required.



Secondly, you have the option to use ‘Bundle Index Versioning’ this simply adds a ‘bundlesPlayerVersion’ to the AssetBundleIndex (by default this is set to the ‘Build Version’ as defined in the player settings.) You can use this to determine if the bundles the user has cached are uptodate with the player version of the game.

Lastly, you have the option to encrypt your bundles. This option will encrypt all your assetBundles when they are built (not just UMA ones) and the decryption is handled automatically as the bundles are downloaded.

When using Encryption it is important that every ‘DynamicAssetloader’ in your game also has encryption turned on. This will automatically fill in the encryption field with the bundle encryption key, you can change the encryption key in UMA AssetBundle Manager Settings, but remember to rebuild your bundles when you do this and also be aware that any bundles that used a previous encryption key will no longer be able to be decrypted.



Testing your Asset Bundles

There are two ways to test your assetBundles, ‘Simulation Mode’ or using the ‘Asset Bundle Testing Server’.

‘Simulation Mode’ is the fall back method that is used when no remote URL is set in DynamicAssetloader and the ‘AssetBundle Testing Server’ is turned off in the ‘UMA AssetBundle Manager Settings’ window. In this case assets assigned to assetbundles in your project are discovered and loaded directly *from the project.* So in this case you are not actually using any assetBundles you have built, and DynamicAssetLoader just pretends the files in the project are actually in assetbundles. This is generally fine when you are working on development of your game and dont need to specifically test your bundles (for example you are working on a PC but developing for Android/iOS. In this case using your actual built bundles will result in ‘pnk shaders’ on your characters in the editor, because the shaders in the bundles will be for Android/iOS)

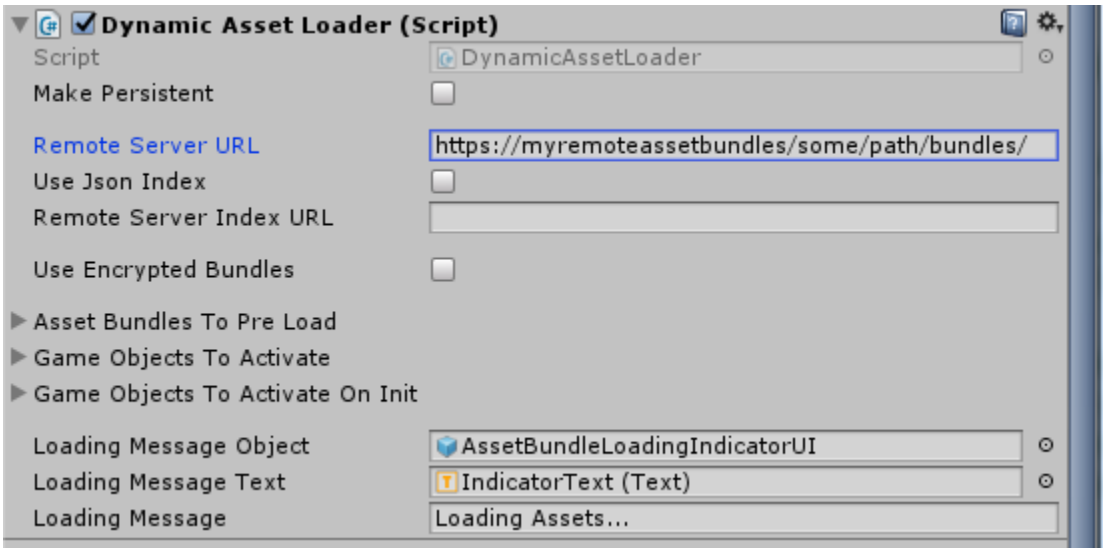
The ‘Asset Bundle Testing Server’ when enabled will create a simple light local server that will allow AssetbundleManager to load the actual assets from the assetBundles you have built (which should be in the project root in ‘AssetBundles/[PlatformName]’) In this case you can test which asset bundles actually get ‘downloaded’ when certain assets are requested and fix any ‘dependency’ or duplicate assets issues. You also use the Asset Bundle Testing Server to test your encrypted bundles.

Using your AssetBundles- Going Live

To actually use your assetBundles you will need to deploy them to a remote server and set the address of this server in the ‘Remote Server URL’ field of each ‘DynamicAssetLoader’ in your game. TIP: You can make the DynamicAssetLoader persistent if you wish and use the same one throughout your game. The DynamicLibraries will find it by themselves.

When you upload your bundles it is important that you keep the folder structure intact as the bundles were actually built. I.e. you need to copy the folder ‘AssetBundles/[PlatformName]’ to https://myremoteassetbundles/some/path/bundles/[PlatformName]’ This is because the AssetBundleManager will append the current platform name to the url, and the AssetBundleIndex expects the assetBundles themselves to be in that folder structure.

Then you need to set this URL in the field in the DynamicAssetLoader, without the platform name but with the trailing slash eg in the example above you would put ‘https://myremoteassetbundles/some/path/bundles/’



When this field is filled in the DynamicAssetLoader will actually download assets from that location when you enter play mode.

The first thing that the DynamicAssetLoader will do as soon as it starts is download the ‘AssetBundleIndex’ from that location, this file is inside the [platformname]index[noextension] assetBundle in the root of the platform folder eg with the above example on Android it would be: https://myremoteassetbundles/some/path/bundles/Android/androidindex. So long as you filled in the URL correctly (https://myremoteassetbundles/some/path/bundles/) the platform and file name will be added automatically.

From there on everything should ‘just work’ when you use the same method calls as you would with assets in the GlobalLibrary (i.e SetSlot, ChangeRace etc)

If you wish you can add an alternative link to a json formatted index, that your server could generate on the fly. This way you can have more control over the folder structure of your bundles, do things like send a user ID and only return certain bundles based on the user and things like that. If you use this feature you can use [PLATFORM] in the url and it will be replaced with the current environment platform. TIP: When you build your assets a json index is also built in the same location as the standard index with the file extension '.json' (i.e. https://myremoteassetbundles/some/path/bundles/Android/androidindex.json) you can check this out to see how your server should format the data for a dynamically generated index

One More Thing…

You can also use DynamicAssetLoader directly to download any specific asset or assets of a certain type, not just UMA assets.

There is an example of this in the UMA/Examples/ExtensionsExamples/DynamicCharacterSystem/Scenes 'UMA DCS Demo - Using Asset Bundles' scene.

Check out the 'UI/WardrobeCollectionLibrary' object in the Hierarchy (and its associated script), that asks DynamicAssetloader' to download all assets of type UMAWardrobeCollection when it starts. The interface in that scene, then shows the collections it has found as download options for the active race (if the race is HumanMale/Werewolf/ToonFemale) when the 'Get More Content' button is pressed.

So by doing this kind of thing you could populate your UI with wardrobe items your user *could* have if they paid some money for example.

Useful Links

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