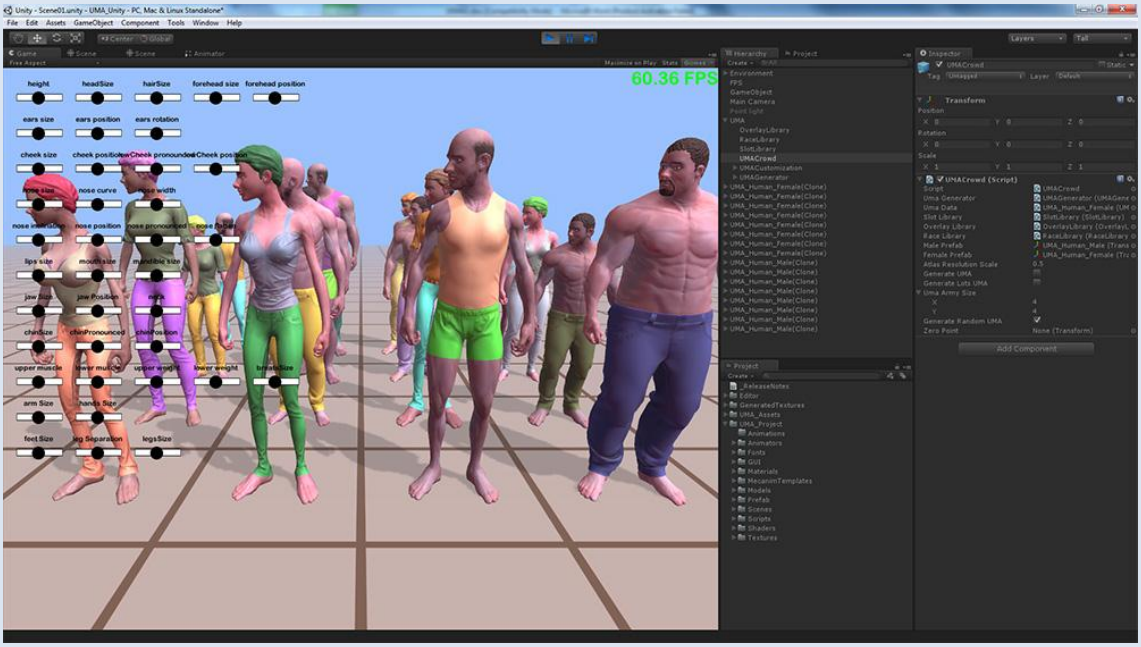
UMA - Unity Multipurpose Avatar

UMA version 1.0.1.0R

Unity 4.3



Avatars generated by UMA framework

Overview

What is UMA?

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.

Close on detailed avatar

UMA works both on Unity PRO and free licences, but benefits from PRO features for accelerating UMA creation steps.

This project has been updated for Unity 4.3, as it will require access to the Mecanim avatar API on following releases.

UMA can be integrated and used on mobile, standalone and web player builds and included shaders require Shader Model 2 or superior. It has not been tested on console builds. If you´re targeting old devices, it ´s recommended to reduce texture resolution and take other optimizations measures to avoid memory issues.

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 it ´s 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.

100 Avatars in real time

Free and PRO license differences for UMA

UMA results on PRO and Free licenses are meant to be as similar as possible, however calculating a UMA atlas can be handled differently. For PRO it ´s possible to create the atlas using RenderTextures, so texture calculation is very fast. For Free license, it requires processing each pixel individually, so it takes considerably more time and processing power. It ´s possible to split this calculation over as many frames as necessary to avoid drastically lowering frame rate, but higher resolution textures will require a considerable amount of time to be processed. The gameObject UMAGenerator provides an option for using PRO or Free, I´ve left this as a way PRO users testing performance of content on indie projects, for example. The Free license users need to uncheck the *usePRO* checkbox on UMAGenerator.

How does UMA 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 realtime.

To achieve that, I´ve created a special rig structure 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 mehes, they receive the same influece 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.

I´ve also implemented an overlay system 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.

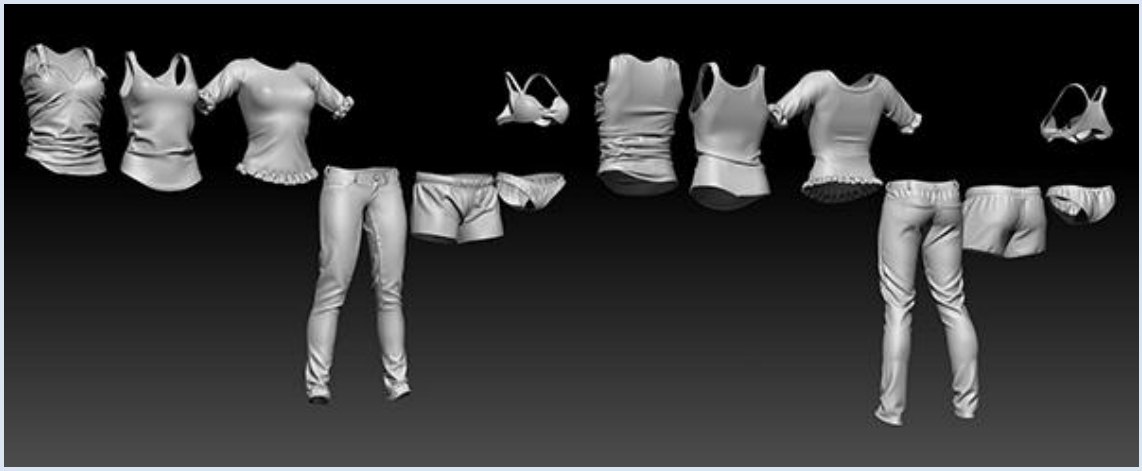
UMA optimization occurs in many steps: Each UMA avatar can have an unique texture atlas providing all necessary texture data, this makes it possible to have each UMA generating a single drawcall, in the case of a single material being used.

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.

UMA avatars generated based on same content and different shapes

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

How content is created?



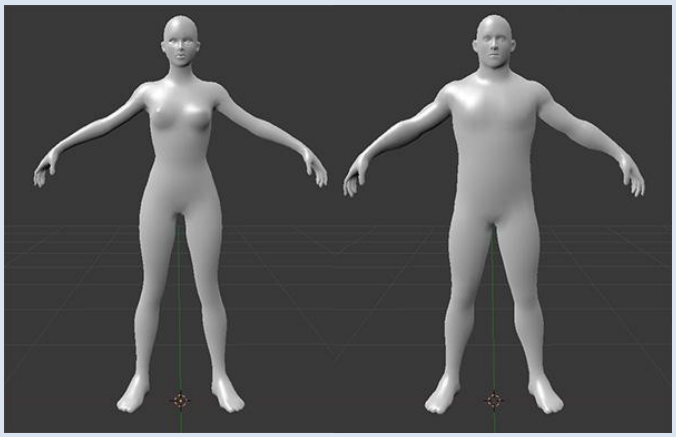
Set of High poly meshes based on UMA final clothes.

Creating content for UMA doesn´t require much extra knowledge beyond the usual asset creation pipeline. There are three main type of content: Base meshes, Slots and Overlays.

Base Meshes

UMA provides two starting base meshes, a male and a female. It ´s possible to create completely different and unique base meshes and still take advantage of UMA. As all content uses base meshes as reference, if you create a minotaur base mesh and are able to keep the same data from the original male torso base mesh, all clothes based on that male torso can be shared with the new Minotaur mesh.

For a completely different race, such as horses, you will need unique content. This could then be shared with other similar races, like unicorns and even dragons. Male and female Base meshes were created considering the average volume those bodies would be able to reach using the adjustments in the UMA rig.



Uma Female and Male base meshes.

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.

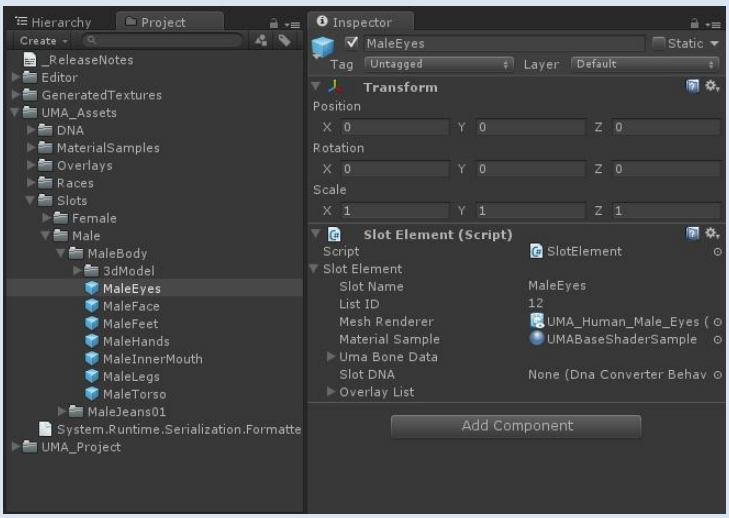
For example the base meshes provided are normally split into several pieces, such as head, torso and legs... and then implemented as slots which can be combined in many different ways.

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 Bumped Specular 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 wouldn´t be visible. To handle this, it´s important that the vertices along mesh seams share the same position and normal values to avoid lightning artifacts.



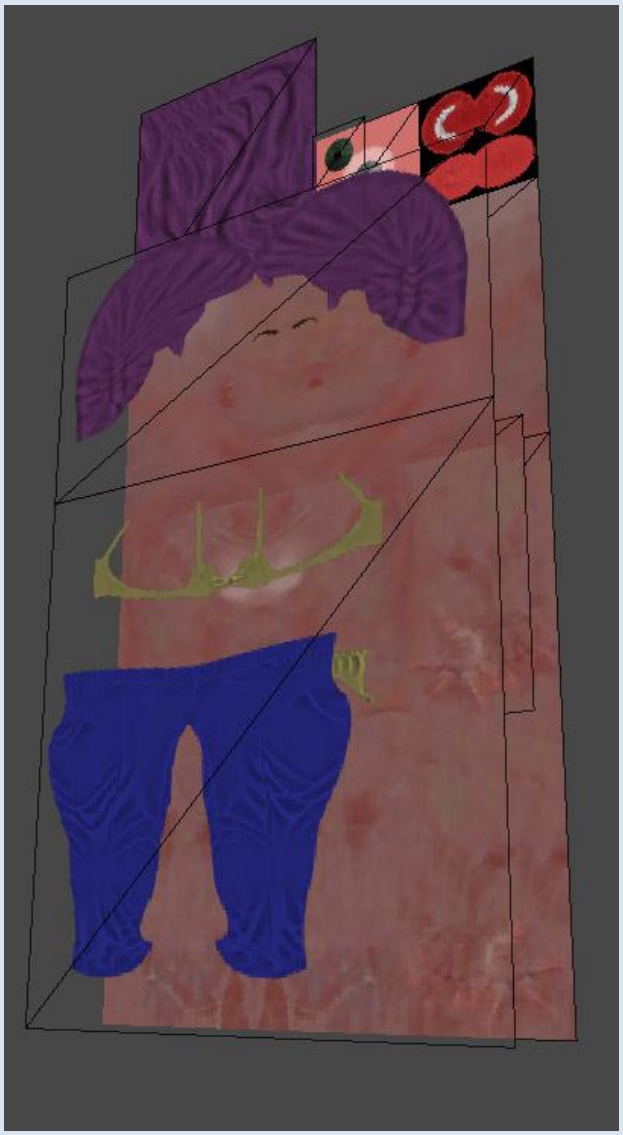
Example of Male slots.

To handle that, we provide a tool for importing meshes that recalculate the normal and tangent data based on a reference mesh. UMA MaterialBuilder will be explained in following pages.

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.

UMA standard shader requires two textures provided by overlays, one texture for Diffuse color (RGB) + overlay mask (A) and one texture for Normal map(GA), Specular (R) and Gloss (B). These non-standard textures let us compress a lot of information in only two textures, reducing final memory usage.



Example of overlay composition

Together, Joen Joensen and I have worked on “UMA material builder”, a tool that receives 3 standard textures for Diffuse(RGB) + mask(A), Normal map(RGB) and specular(RGB) + gloss(A), and compacts the data into the two textures described above. In the process, the specular color is reduced to one channel of data, the resulting average color of the 3 channels provided.

Following Pedro Toledo's post “Brief Considerations About Materials” http://www.manufato.com/?p=902, Joen and I worked together to reach two shader that handle specular color based on diffuse color reference resulting in Dielectric and conductor materials.

I´ve already managed to integrate some of the most common Unity shaders into UMA and the adjusts required are quite simple. Even more advanced shaders can be integrated and used, keep in mind it´s possible to provide extra textures on each

overlay, this way, it´s possible to include displacement maps, Sub surface scattering masks or any necessary data.

Asset creation pipeline

UMA project provides an UMA content creator pack, it´s a zipped folder with all necessary base meshes and textures for creating your own content. It provides:

- base diffuse, specular and normal map textures

- UV layout reference images

- Male and female base meshes in .OBJ

- Rigged and Skinned base meshes in .FBX file format

- .ZTL files for Zbrush users

- An open .Blend file for blender users

I´ve spent a long time recording and producing video tutorials (http://www.youtube.com/user/fernandoribeirogames) to cover all the basic process of content creation, but the knowledge for actually working on any 3d software is prerequisite.

The overall knowledge for generating UMA content is 3d modeling, rigging, skinning and texturing. It´s also possible to work on existing content already available. For example, if you have an tshirt slot, with the right texture work it´s possible to provide an chain mail overlay without extra knowledge of modeling, rigging or skinning.

Texturing and UV mapping

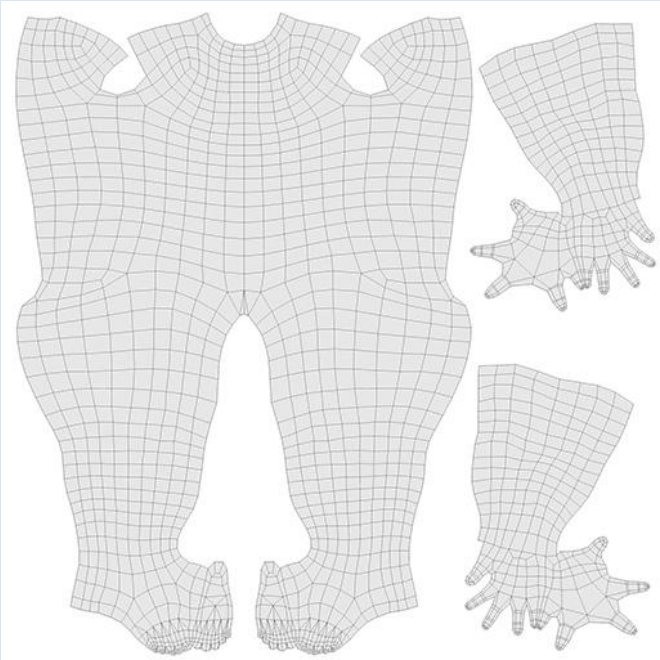
Both UMA male and female base textures have a specific resolution. Below is the list of those base texture sizes in pixels:

- Head : 2048x1024

- Body : 2048x2048

- InnerMouth : 512x512

- Eyes : 512x512



Male body UV layout

Those values are specific to the provided base meshes as a standard for anyone creating content for them. If you plan following a different standard, it´s possible to use any different resolution.

When creating new content for UMA, if you´re aware it will be covering an area of UMA body texture, it´s possible to use that UV area for the new content texture, if it offers enough space. This example can be seen on Female tshirt and hair, both of them save atlas space, having those textures as overlays for body base texture instead of being base textures themselves.

Also, any overlay texture and it´s covered base textures don´t need to keep the same size. For example, FemaleUnderwear01 overlay covers only the left half of the base, so it´s possible to have that overlay with half the width of body base texture.

The Rect provided together with the overlay elements is responsible for keeping information of the positioning adjust of the cropped overlay, relative to base texture coordinates.

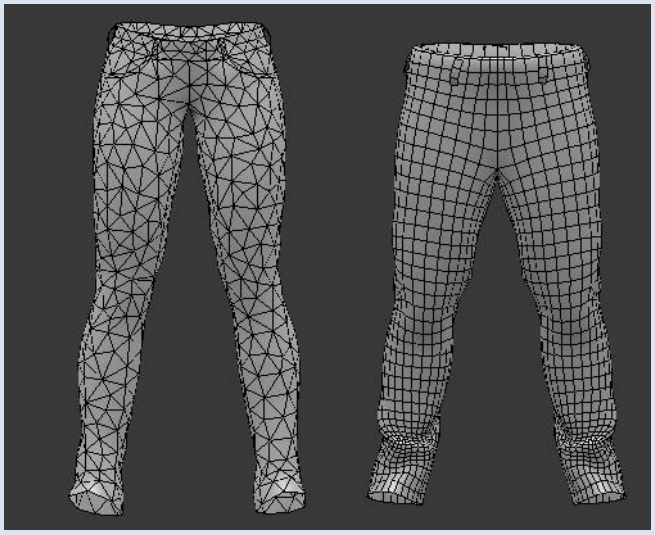
It´s possible to provide overlays that will receive color adjusts at atlas creation. In those cases, usually the predominant color is white or gray in those areas, to have a neutral influence over final color.

Above image illustrates the use of a cropped overlay in junction with Eye base texture to generate iris color variation

In other hand, UMA 1.0.1.0R don´t support combining textures with different original size: Even cropping an overlay removing the unused masked area, the original overlay size should be the same of the base texture. So if you provide a cropped overlay for eyebrows, it requires a Rect data relative to the 2048x1024 head reference texture. Providing a non cropped overlay with smaller or bigger resolution than the base overlay might generate error messages.

3d modeling

It´s possible to integrate any 3d mesh into an UMA avatar, it´s important to follow the same optimization guidelines usually used for traditional characters and clothes, as topology and vertex count. I´ve included meshes with both uniform and non uniform polygon placement, it´s important to keep in mind when each case can be used. The same way, I´ve worked both on meshes mostly quad based, and meshes entirely based on tris before exporting process. All content can be integrated despite those differences, but stretched polygons might cause poor lightning results, especially visible when not using normal maps.



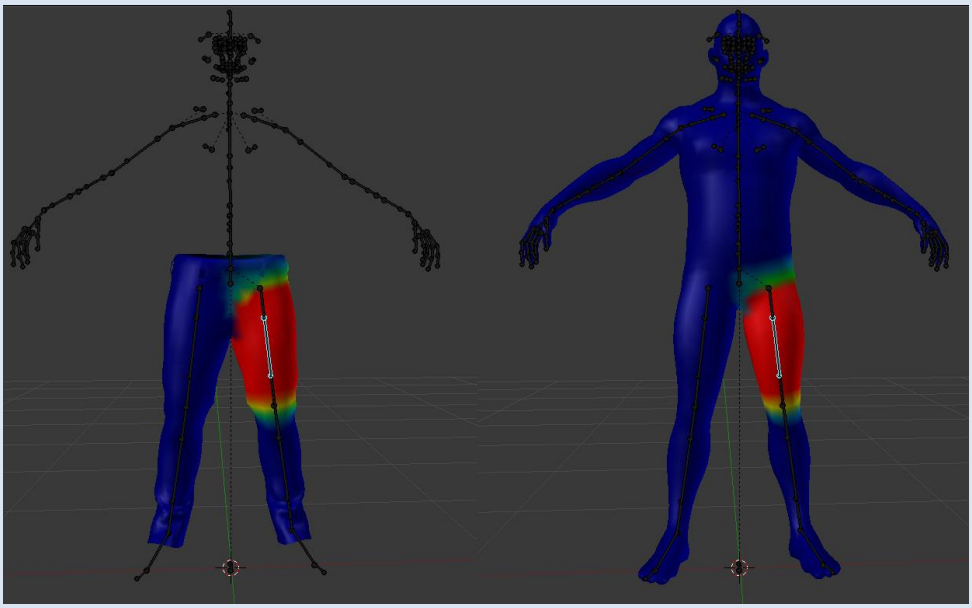
Female and Male jeans completely different topology

Rigging and skinning

This is where all the UMA magic happens. All shape variations we can achieve on UMA avatars are a consequence of changing bone scales, positions and rotations. A mesh correctly following those changes depends on rigging and skinning entirely and an incorrect skinning process might lead to issues such as a clothing pieces not following the same shape variation as the body.

I´ve provided base meshes with rig and skin data because most 3d software has specific tools that allow users to transfer bone weight data between different meshes avoiding most of the time consuming process involved. I´ve shown this being done in Blender at some video tutorials (http://www.youtube.com/watch?v=ImD6APS0xek), but the same can be done with XSI Gator or other specific solutions.

For dresses or armor, simply projecting those values might not be enough, and skinning knowledge may be necessary for best results following body variations. What is important to keep in mind if you´re creating your own rig is to keep a pair of parent and children bones anywhere you need that specific area to receive non-uniform changes. This is the case on most of the UMA rig, for example the arms. You´re able to change the arm scale both in uniform and non-uniform ways, but having an non-uniform scale deforms all child bones too, so those changes need to be applied to the child bones working in pair. It´s also important to keep in mind it ´s always these child bones that carry the skinning data, as it will always change directly, and under the influence of the parent bone.



Male jeans skinning and male base mesh skinning. Skinning Data has been projected with Blender Transfer weights tool

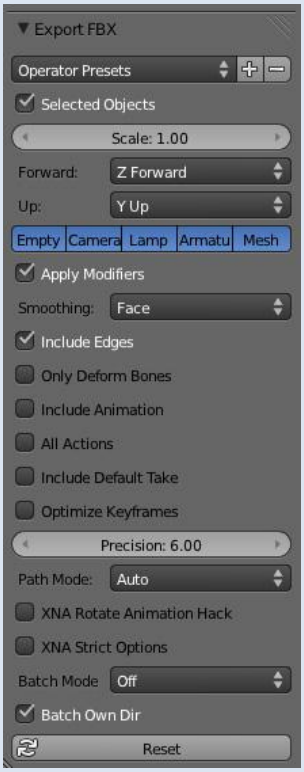
3D software

Blender is UMA standard content creation software, because it´s open source and accessible for all developers. Being the standard means I´ll provide most of the 3d tutorials on this specific application but keep in mind most of the available softwares have a set of tools to handle what I´m doing in blender.

Initially, I planned including here the standard import and export setting for each 3d software I manage testing the integration with UMA, but as you´re going to notice, importing setting mostly depends where those files where generated, so it´s hard having an standard. For export setting, I´ll explain the specific setting for exporting content for UMA base mesh.

Blender

Importing files: Blender has a huge limitation: it can´t import fbx files (*Update: On Blender latest version, fbx import is already being integrated*), so it will be harder sharing skinned and rigged models from other 3d application to Blender. Importing obj files is a straightforward process, and blender provides settings for definning forward axis and up axis, covering different coordinate systems. It´s usually a good idea checking "*Keep Vert Order*", if you need to keep the index of mesh vertices, this is specially important working with zbrush and other sculpting tools.

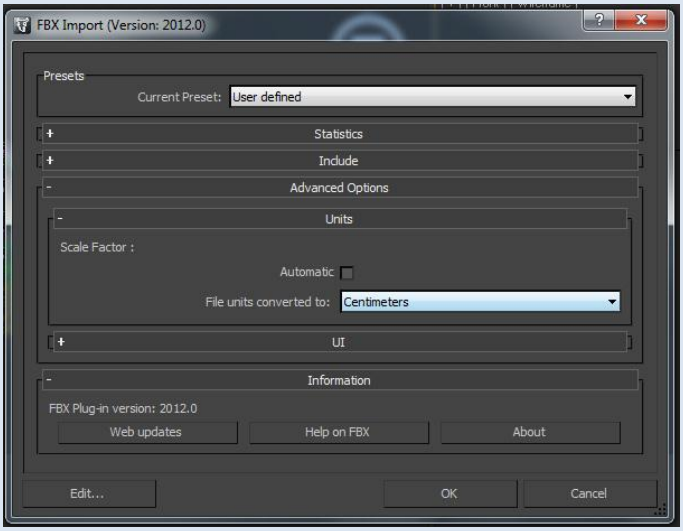


Exporting files: Exporting fbx files of rigged and skinned content to unity requires having "*forward axis*" as "*Z forward*" and "*Up Axis*" as "*Y Up*".

It´s very important when exporting content being sure both it´s position, rotation and scale are normalized, this means you need to bake all those into vertices position. This can be made with "*Ctrl+A*" or "*Object/Apply*".

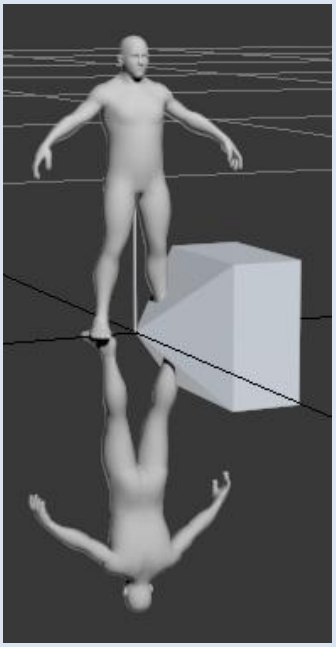
You usually don´t need including animations when exporting clothes and acessories, but it´s really important that the rig is exported along with the mesh for your content. Including the rig might raise the disk space required by that file, so it´s usually a good idea to keep a bundle of meshes on the same fbx file whenever possible.

3ds max



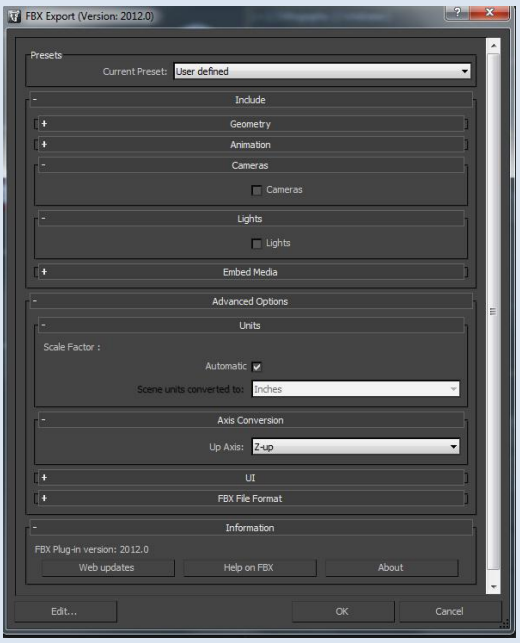
Importing files: I´ll focus on importing files from the "*Content creator pack*", as those were exported from blender, there´s a small change required when importing base mesh fbx files to 3ds max, you need to set "*units*" to "*Centimeters*" to have the correct size.

As you can see on the following image, the fbx file (upper mesh) has the correct rotation and is fully rigged and skinned. The Obj (lower mesh) file requires manual rotation adjusts after importing process.



3ds max provides modifier "*skin wrap*" as a solution for copying skinning data between UMA base mesh and your own content.

Exporting files: The same warning I´ve provided for blender users apply here too: It´s very important when exporting content being sure both it´s position, rotation and scale are normalized. It´s possible to adjust mesh pivot using the "*Affect Pivot Only*" button. There´s a specific adjusts required when creating your own content, the Z rotation value of the mesh should be set to 180 degrees, this is usually done entering *Affect Pivot Only* mode and rotating the pivot itself.



When exporting the fbx files, you need to have both your content mesh and rig selected. It´s usually a good idea not including the cameras and lights on the fbx files. For exporting, units can be kept on "*automatic*" and "*Up Axis*" need to be set to "*Z up*".

Mecanim

Animation

UMA is integrated with an humanoid mecanim avatar, opening a huge list of animations that can be retargeted to any UMA avatar. Both Male and Female UMA prefabs had their mecanim avatar settings adjusted. UMA rig provides enough bones do handle facial animation, the same techniques required for facial animation on any mecanim avatars apply to UMA.

Slots that provide extra bones not linked to humanoid mecanim avatar require extra adjusts to properly receive animation, as animation data won´t be retargeted to those bones.

Example of bone driven facial animation

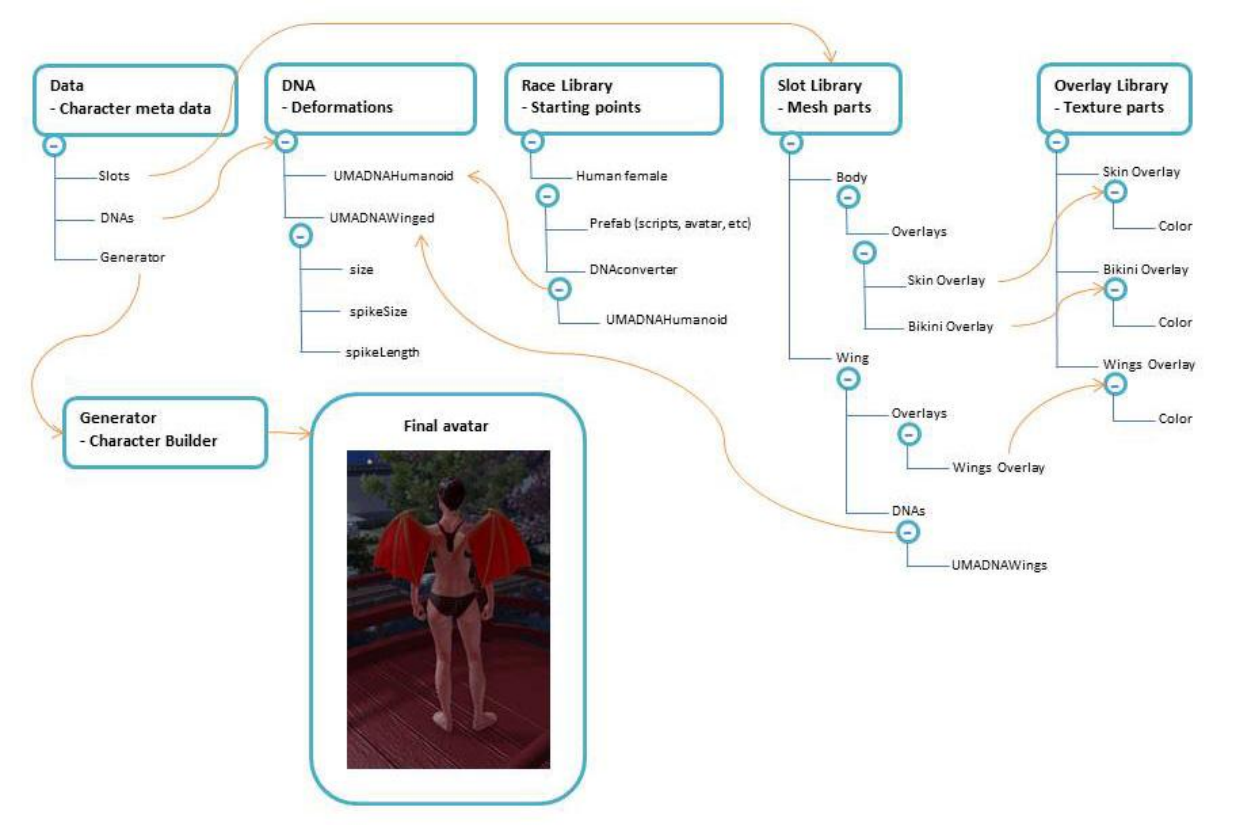
Mecanim avatar creation API

Unity 4.2 provides an Mecanim avatar creation API, this is necessary to recreate mecanim avatars based on changes UMA rig might have received. Before unity 4.2, this was not possible and I had to use LateUpdate to keep bone changes, consuming extra processing time and possibly breaking Mecanim IK depending on the kind of changes bones received. UMA latest version already integrates Mecanim avatar creation and provides full suport to IK and Root motion thanks to Joen Joensen and Mecanim team help.

UMA Components

Overview

Joen provided a diagram for the beta group showing the relationship between the various parts of the UMA framework. I´ve included a new version here to help explaining how avatars are created.



Libraries

All avatars created need to have access to a set of necessary data: Races, Slots and Overlays. Each library provides a list of each of those within an dictionary, being the key an string with the name of the element, and the value the element data itself.

For projects with big amount of content as mmo games, you probably don't want everything loaded in memory and loading/unloading assets might become necessary, this case requires implementing custom libraries.

- race library

Each *RaceData* on the race library provides a prefab with all shared scripts, avatar data and etc, one or more DNAConvertes and a list of the name of bones that require being updated.

As you´ve noticed, I´ve separated male and female as two different race datas. This is because each race provides an specific base mesh and rig/skinning data, and usually require specific content. Using unique base meshes for male and female gave me the possibility of reaching a better silhouette and topology for each of them.

DNAConverters are responsible translating values like “height” and “armSize” to bone deformation and shape changes. It´s possible to have two different races with the same values to “height” and “armSize”, but each of them converting those values in different ways.

- Slot library

Each slot provides a skinnedMeshRenderer and a material sample, as explained earlier. If you´re planning on inluding extra bones on any Slot, it will be necessary to use Joen´s “UMA material builder” for generating the *slotData*, as it is responsible for identifying if there´s any extra bone and listing all their transforms.

The first overlay in the *slotData´*s *overlayList* provides the base textures, and all other overlays included are combined with the first one, in sequence, generating the final atlas.

- Overlay library

Overlay library gives access to all available *overlayData* , This library might use a huge amount of memory depending of the project, as it keeps direct reference to texture files. By default all UMA textures are provided with read/write tag checked, but this is only necessary if you´re using free license.

Each overlay have an *color* that can be used to tint it´s first texture on *textureList*. Usually this value is set by code, at the moment the overlay is being included on the slot.

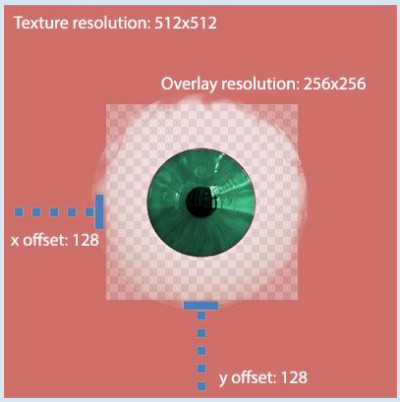
The *Color32* arrays channelMask and channelAdditiveMask gives exceptional control over adjusting texture channels. Each color here will change the overlay texture from *overlayList* sharing the same index. With this, it´s possible to adjust each texture channel of each overlay of an atlas.

For example, if we consider reducing the gloss value of a hair texture, we could set an channelMask second color (index 1, relative to second texture, where normal map, specular and gloss are kept) channel “b” to a lower value.

Functions public void SetColor(int overlay, Color32 color) and public void SetAdditive(int overlay, Color32 color) on overlayData are responsible for setting those values. We could have something like hairOverlay. SetColor(1 , new Color32(255,255,50,255)); to reduce gloss value on above mentioned example.

The rect value is specially useful if you´re providing an overlay that covers a small area of the base texture. Both male and female eyebrows cover only a small part of the head base texture, and it would be a waste of memory having a huge texture with most of it´s area completely masked.

The x value is the horizontal offset of the cropped overlay, y value is the vertical offset, and width/height are relative to cropped overlay size.



cropped overlay being combined to base texture

I´ve managed to include on Overlay library a set of buttons to adjust all overlay textures in a single click, this is specially usefull to reduce the resolution of all textures or set their compression.

UMA shape

- UMADna

Being able to adjust avatar shape in real time, even if the avatar share both texture atlas and mesh is very powerful. This is possible because we use bone changes to deform both the avatar body mesh and it´s clothes. To archive those bone changes, we need to know both which bones should be adjusted and how this should happend.

UMADna holds the values from changes an avatar can receive, and we provide an *UMADnaHumanoid* class inheriting from *UMADna* as example of how this works. Both male and female avatars uses this *UMADnaHumanoid* to define their shape values.

If you consider the winged humanoid on the diagram image, it might be possible to have a single *UMADnaWinged* being used by both male and female wings. This class would only need to provide the extra values like “size” , “spike size” and “spike length”.

- DnaConverterBehaviour

The *DnaConverterBehaviour* is responsible for converting the Dna values to bone changes, This is where the code for deforming bones is kept. We provide both an *HumanFemaleDNAConverterBehaviour* and *HumanMaleDNAConverterBehaviour* inheriting from *DnaConverterBehaviour and a*s you can see, male and female avatars share the same shape values, but resulting bone changes are specific to gender.

Creating UMA avatar

UMAGenerator

UMAGenerator provides all necessary functions and methods to handle avatar creation. Creating an avatar is done in 3 steps: calculating it´s final mesh and rig, processing it´s texture atlas and updating it´s shape.

umaDirtyList keeps track of all UMA avatars that need to be created or updated, in documentation following pages it will be clear how and avatar is included on list.

Both *usePRO* and *convertRenderTexture* visible as checkboxes on unity editor are very important. If you have an Unity PRO license and usePRO enabled, it ´s possible to create the atlas using RenderTextures, so texture calculation is very fast. For Free license, it requires processing each pixel individually, so it takes considerably more time and processing power. It ´s possible to split this calculation over as many frames

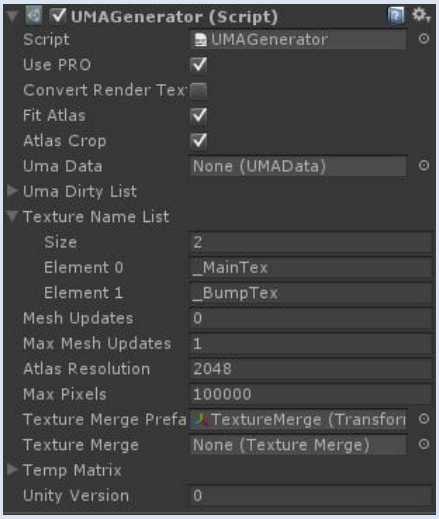
as necessary to avoid drastically lowering frame rate, but higher resolution textures will require a considerable amount of time to be processed.

convertRenderTexture is useful in case you´re working with Unity PRO, but don´t want the final atlas being a RenderTexture instead of Texture2D. This might be the case if you´re planning compressing the atlas. Keep in mind this process is heavy and might bring slowdowns on 2048x2048 atlases. it´s Also known that RenderTexture data won´t be kept on Webplayer after window is resized or on builds when for example accessing task manager, in those cases atlas would have to be recalculated or converted from RenderTexture to Texture2D to avoid the problem.

textureNameList provides a solution if you´re planning using extra textures that should be baked on atlases and applied to your sample materials. Usually we have _MainTex and _BumpTex, and if your sample material shader require both of them, atlas will search on overlays´ *textureList* index 0 and 1 for those two. In case you add a third texture name, for example _DetailTex, and provides a shader requesting this data, overlays using this shader are expected to provide the extra texture on corresponding index.

maxMeshUpdates and *maxPixels* defines how many meshes will be baked and how many pixels can be processed on a single update step. Setting *maxPixels* is only necessary when not using *usePRO*. If you´re targeting mobile or old devices, it´s important to profile the performance and adjust those values. It might also be useful to adapt those values dinamically depending on actual performance for the specific device running the project.

atlasResolution is very important, it defines maximum resolution of the atlases generated. If you need very detailed textures, you might want to increase this to 4096, but keep in mind the memory used for high resolution textures is really big, specially if not compressed. It´s possible to adjust overall overlays resolution to fit smaller atlases, this will be covered on following pages. Setting atlas resolution to small values but not adjusting textures resolution accordingly will generate errors on project, as not all textures will fit atlas.



UMAGenerator

FitAtlas Forces final atlas to fit Atlas resolution in case the resulting atlas size would be larger than maximum size.

AtlasCrop Final atlas unused texture space will be cropped. If you need the resulting atlas being a square texture, this is not recommended.

UMACrowd

UMACrowd is an example of how UMA avatars can be randomly created based on content we have on libraries. Both colors and shape values don´t have any elaborated creation method. In most games you will need a color palette and more control over shape and cloth combinations.

First of all, *UMACrowd* needs to keep track of *SlotLibrary*, *OverlayLibrary*, *RaceLibrary* and *UMAGenerator*. You can consider *UMACrowd* as a chef with access to all necessary ingredients and tools to cook an unique meal. Each UMA avatar, or meal if you wish, is created based on an specific recipe the chef writes based on available ingredients and how they are combined. UMAGenerator is the available tool the chef uses to actually cook the meal, and bake an UMA avatar.

atlasResolutionScale This value changes the overall texture size of all overlays an UMA avatar receives, reducing final atlas size. If you´re planning on having a lot of UMA avatars with unique atlases, it might be necessary considering a lower value here. Usually the values we set here are 1.0f, 0.5f, 0.25f, 0.125f and 0.0625f, as those are also the values for getting the correct mipmap on Free license. As default, it´s set to 0.5f, with atlas resolution of 2048.

generateUMA , visible as a checkbox at Unity Editor, checking it generates one random UMA avatar.

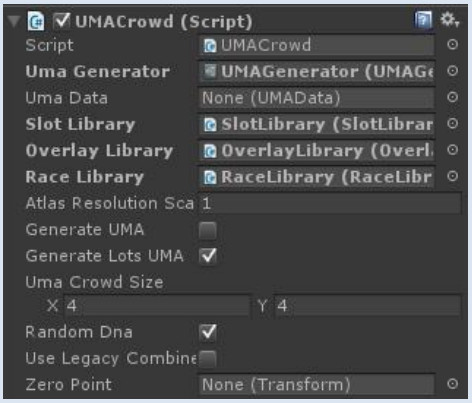
generateLotsUMA, visible as a checkbox at Unity Editor, checking it generates a groupd of random UMA avatar, based on below *Vector2* value.

umaCrowdSize defines the amount of UMA avatars created on column and row. Default values are 4x4, resulting in 16 UMA avatars.

randomDNA visible as a checkbox at Unity Editor, checking it generates random shapes for avatars, otherside all avatars will have neutral shape.

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

zeroPoint Defines the position where UMA avatars are created. If it´s empty, defaulf value is *Vector3.Zero*.



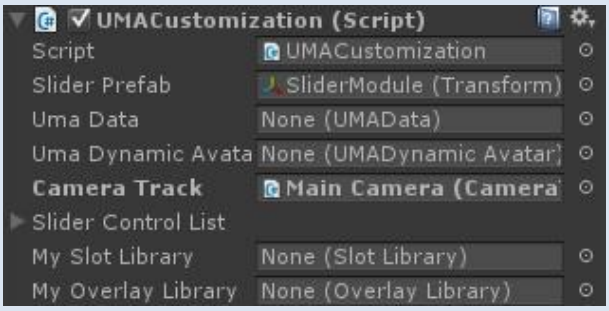
UMACrowd

UMACustomization

UMACustomization exemplify changing avatar shape after it´s creation. Following the same concept, it´s possible to change UMA slots and overlays, triggering them to update with the new content. At the example scene, right clicking on any UMA avatar updates *UMACustomization* sliders with its specific shape, after that it´s possible to use sliders to change any shape value.

UMACustomization generates a rough example of sliders based on GUITextures as reference of how to handle changing UMA shape based on user input, but it´s clear each project will require a custom approach to handle both GUI and user input.

Exchanging UMA cloth and acessories means changing the array of slots used for creating an UMA avatar and including that avatar on *umaDirtyList at UMAGenerator* to get updated.



UMACustomization

UMAData

UMAData provides many functions, methods and classes being used on avatar creation as an standard for sharing information.

Each avatar has it´s own *UMAData*, that´s why manually duplicating avatars on Unity editor might generate errors when trying to access any avatar data, as their *UMAData* has not been properly set.

It´s important to explain the concept of "dirty" at *UMAData*. Both *isShapeDirty*, *isMeshDirty and isTextureDirty* are responsible to keep track of what needs to be updated or recalculated on this UMA avatar.

isShapeDirty means shape data needs to be recalculated. This is usually set when any DNA value received changes that require updating UMA bones.

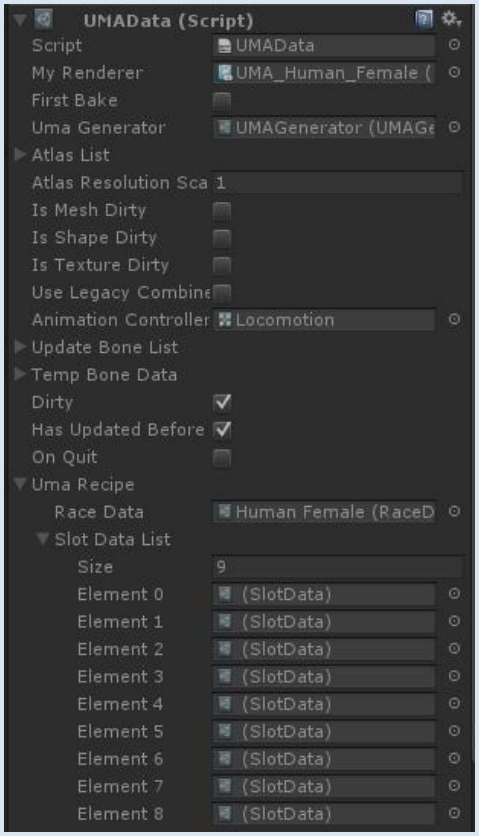
isMeshDirty should be set *true* when UMAData received slot changes and avatar mesh requires being baked again. This also applies if overlays have changed as most likely atlases and in consequence mesh uv will require to be recalculated.

isTextureDirty means atlases require being recalculated. On UMA actual stage, the entire atlases are generated from scratch, but it´s possible to keep track of changes and implement an advanced solution forcing atlas recalculation on specific areas only.

useLegacyCode is set by UMACrowd when creating the avatar, but can be set directly on UMAData.

animationController can be an unique animator for each avatar. This is necessary because when the avatar shape changes and a new mecanim avatar is created, the animation controller need to be set back.

umaPackRecipe was created considering that all necessary data to recreate an UMA avatar might need to be saved or shared on multiplayer games. What *umaPackRecipe* provides is the minimum necessary data for serialization, usually an array with the names of slots, overlays and DNAs, and the extra changes that had been applied to those. All this data can then be unpacked into an *umaRecipe*. In the process, respective libraries provides elements based on serialized names and extra changes are applied to those.



UMAData

Credits

Without Caitlyn Meeks-Ferragallo (Unity Asset Store Manager) and Jay Santos (Unity Field Engineer), UMA probably wouldn´t reach Asset Store any time soon, and even if it did, it wouldn´t have the same quality and flexibility it provides today.

Unity sponsored this project, making it possible to be available at Unity Asset Store for free. In the last months, I´ve received a huge help from Joen Joensen (Unity Software Developer, QA), he both provided feedback for UMA code and included his own scripts to the project.

Development

Fernando Cardoso Emiliano Ribeiro (Huika Game Studio) - UMA Developer

Joen Joensen (Unity Software Developer, QA) - Software architect

3rd party scripts

Aras Pranckevicius (NeARAZ) - HUDFPS

Joen Joensen - WorkerCoroutine, UMATextureImporter, SkinnedMeshCombiner

Sven Magnus - MaxRectsBinPack

A huge thanks for the following beta testers and forum members

Breyer, Cynel, ecurtz, FlorianSchmoldt, janpec, Josef Šíma, SinisterMephisto, Tesla Coil, virror, Whippets.