Avatar CLIP Report

Ran Marashi

Michael Radzeiovsky

Eitan Levi

Table of Contents

Abstract:

The goal of this project is to create a VR scene using avatars that will be generated using the repository at https://github.com/hongfz16/AvatarCLIP. The scene will contain numerous different, diverse looking avatars that were generated using this repository. The different avatars will have basic motion capabilities and will be rigged such that they will be able to interact with their environment in a way that physically makes sense (For example, they won't be able to move through each other or through other objects as if they were not there). Unity will be used to create and design the scene. Different kinds of GPUs will be used for the generation, and their capabilities will be compared. We will also focus on the limitations and shortcomings of the implementation, on the installation process and on the integration with Unity.

Note: It doesn't seem like the repository is maintained any longer, as the repository wasn't updated in several months and new issues that are raised are not handled by the developers.

Installation Process:

Although clear instructions are provided in the readme file we still experienced several difficulties.

First of all, using Windows (at least on Windows 10) to do the generation is not possible. This is due to the neural renderer module which doesn't seem to support Windows. This was confirmed by using a computer with sufficient hardware with dual boot: When trying to install the required files on Windows error messages were generated. When we switched to a Linux environment (Ubuntu 22.04) the installation was completed smoothly. This was the result using a few other computers as well and no method of overcoming the errors generated on Windows could be found.

If it is important for one to use a Windows based environment for the generation, it may be possible to use another neural renderer to see whether the generation can be completed successfully. Thankfully the resources which were provided to us (At first the Lambda computational cluster and afterwards the Newton cluster) are not Windows based so we could proceed once they were provided.

From now on everything written in the "Installation Process" section is relevant to those environments only.

Using older GPUs (GPUs that support CUDA10.x):

This was the case for us at first because we had access only to the Lambda faculty server which has older GPUs like GTX 1080.

In this case no changes are necessary. Simply follow the instructions in the readme file.

Using newer GPUs (GPUs that support CUDA11.x):

In this case there is an important change needs to be made which was not mentioned in the readme file.

In the requirements.txt file change the line:

git+https://github.com/mingyuan-zhang/neural_renderer.git

To:

git+https://github.com/adambielski/neural_renderer

Editing the line enables CUDA11.x support which is not originally provided. Without editing this line the following error will appear:

```
Traceback (most recent call last):
  File "main.py", line 21, in <module>
    from models.utils import lookat, random_eye, random_at, render_one_batch, batch_rodrigues
  File "/home/ranmarashi/AvatarCLIP/AvatarGen/AppearanceGen/models/utils.py", line 3, in <module>
    import neural_renderer as nr
  File "/home/ranmarashi/anaconda3/envs/AvatarCLIP/lib/python3.7/site-packages/neural_renderer/__init__.py", line 3, in <module>
    from .load_obj import load_obj
  File "/home/ranmarashi/anaconda3/envs/AvatarCLIP/lib/python3.7/site-packages/neural_renderer/load_obj.py", line 8, in <module>
import neural_renderer.cuda.load_textures as load_textures_cuda
ImportError: libcudart.so.10.1: cannot open shared object file: No such file or directory
```

The generation process (Not including coarse shape generation):

The generation process also varies greatly depending on the GPU hardware available.

This is due to the fact the a very powerful GPU is required to produce high quality results. In the repository it says that the authors of the article were using an NVIDIA V100 32GB GPU.

Generation using cheaper GPUs is certainly possible, but it is clear that the quality of the generations is affected. We will expand on this in the results section.

Each avatar generation is based on a single .conf file (unless coarse shape generation is used and then the process is slightly different). In the .conf file it is determined what avatar will be generated and with what parameters. Stronger GPUs will produce results of higher quality because they support values of parameters that weaker GPUs cannot.

Because the generation is a long process it is recommended to experiment with different values as early as possible to make sure that the quality of the avatars generated is as high as possible. We will expand on this in a few paragraphs.

Another important detail is that a single generation process doesn't actually create a single avatar. In the default settings an almost complete avatar (a mesh stored as a .ply file) is created every 500 iterations where only the validate_mesh command is left to be executed in order to complete the generation. A picture which displays the current state of the avatar is generated every 100 iterations. These default settings may be changed by the user in the suitable .conf file.

Using the generated pictures it is possible to get an idea of how the avatar will look like. The pictures are stored in the **validation_extra_fine** folder (located at AvatarCLIP/AvatarGen/AppearanceGen/exp/smpl/examples/<avatar>/validations_extra_fine). This is highly beneficial for a few reasons. First of all as we mentioned the generation is a long process and we would like to make it as quick as possible to save time and resources (this is even more important when working in a cluster shared with other people like we did). It is very possible that before the generation process is completely finished we will see that the avatar is already at sufficient quality allowing to terminate the current

generation and move on to the next one using the latest mesh that was generated.

Being able to have a look at the avatar prior to the completion of the generation is also beneficial because it allows us to spot buggy generations sooner, saving time and computational resources. We will expand on the possible bugs in the results section.

Last note regarding the .conf files: In some cases, like the number of iterations for example, a limit on the value of the parameter is enforced inside the python code. For example in the original setting of the .conf file we can see that the default value for end_iter (the number of iterations) is end_iter=100000. However, when generating the avatar we'll actually see that the generations suddenly halts after the 30010th iteration. Trying to understand what happened we found the following lines:

```
if iter_i == 30010:
    break
```

These lines cause the generation to conclude after at most 30010. Several similar lines are present for other parameters.

We recommend to leave these limit as is. Again, if we focus on the number of iterations, we noticed that when allowing the generation to continue past the 30,000th iteration the avatar has started to look worse (while using the default values for other parameters). This combined with the fact that more iterations require more time to complete made us stick with the original number of iterations (which as we'll mention later still may be too large at times).

<u>Older GPUs:</u>

If an older, weaker GPU is used it is most likely that the following error message will appear straight way, stating that CUDA has ran out of memory (specific numbers are depending on the exact GPU and hardware):

RuntimeError: CUDA out of memory. Tried to allocate 978.00 MiB (GPU 0; 15.90 GiB total capacity; 14.22 GiB already allocated; 167.88 MiB free; 14.99 GiB reserved in total by PyTorch)

Although this message can appear for several reasons, it is most likely that the reason is that the parameters that were used for the generation demand too much from the GPU.

In order to still be able to generate avatars it is required to edit the .conf file that was used.

These are the changes that should be done (marked in red):

dataset {
  data_dir = ./data/zero_beta_tpose_render
}
train {
  learning_rate = 5e-4
  learning_rate_alpha = 0.05
  end_iter = 30000
  batch_size = 512
max_ray_num = 7000 is the default for the scaled down network but it is highly recommended that you will use the **largest value possible** that is not larger than 112*112 (the original default). Make sure to be slightly below the limit to avoid generations being terminated in the middle. This both wastes resources and gives an unusable result.
  validate_resolution_level = 1
  warm_up_end = 500
  anneal_end = 0
  use_white_bkgd = False
  save_freq = 1000
  val_freq = 100
  val_mesh_freq = 500
  report_freq = 100
  igr_weight = 0.1
  mask_weight = 1.0
  clip_weight = 1.0
  pretrain = ./pretrained_models/zero_beta_stand_pose.pth
  add_no_texture = True
  texture_cast_light = True
  use_face_prompt = True
  use_back_prompt = True
  use_silhouettes = True
 head_height = 0.7 (Decreasing this value causes a severe deterioration in the avatar quality. It should be as close as possible to 1. Giving values larger than 1 seems to cause runtime errors).
}

```
clip {
    prompt = a 3D rendering of a boxer in unreal engine
    face_prompt = a 3D rendering of the face of a boxer in unreal engine
    back_prompt = a 3D rendering of the back of a boxer in unreal engine
}
model {
    nerf {
        D = 4,
        d_in = 4,
        d_in_view = 3,
        W = 256,
        multires = 10,
        multires_view = 4,
        output_ch = 4,
        skips=[4],
        use_viewdirs=True
    }
    sdf_network {
        d_out = 129
        d_in = 3
        d_hidden = 128
        n_layers = 3
        skip_in = [3]
        multires = 6
        bias = 0.5
        scale = 1.0
        geometric_init = True
        weight_norm = True
    }
    variance_network {
        init_val = 0.3
    }
```

```
  rendering_network {
    d_feature = 128
    mode = no_view_dir
    d_in = 6
    d_out = 3
    d_hidden = 128
    n_layers = 1
    weight_norm = True
    multires_view = 0
    squeeze_out = True
    extra_color = True
  }
  neus_renderer {
    n_samples = 32
    n_importance = 32
    n_outside = 0
    up_sample_steps = 4     # 1 for simple coarse-to-fine sampling
    perturb = 1.0
    extra_color = True
  }
}
```

Adjusting the parameters' values causes us to face two serious problems. The first problem is of course that the quality of the results cannot match the quality of the results that are presented in the project article and in the project website. However this is quite reasonable as better hardware should be able to carry more computations that require more memory, otherwise it wouldn't really be better.

The second problem is much more severe. We **cannot** fully utilize the hardware that we do have in order to produce higher quality results. Even though parameters such as head_height and max_ray_num may be adjusted by us so that we can get the most out of hardware there are other parameters that we have far less control over. This unfortunately includes the most important parameters which are the parameters that define the scale of the network. Parameters such as the number of layers in the network are discrete and they affect other parameters which are powers of 2. We'll notice that many parameters are actually halved in their values and trying to increase

them without increasing other parameters, which we cannot increase using cheaper hardware, will result in pytorch mismatches.

In the repository it says that you may pause the generation after 10,000 iterations. We highly suggest that you don't do so. The avatar body may look fine at that phase, but the face usually seems completely unfinished and even blurry. Hand structure (especially at the palms) also is very lacking at that phase.

If you wish to pause the generation we recommend doing so somewhere in between the 20,000th and the 25,000th iteration using the default values.

The time needed for generation varied between 4 and 10 hours using a GTX1080 with a 12GB memory.

Newer GPUs:

In this case there is no need to edit the .conf files.

In this case pausing after 10,000 iterations is usually acceptable, as avatars already contain their main features as opposed to the case of the older GPUs. In some cases it may actually better to pause after 10,000 iterations than to continue (This will be discussed shortly).

The time needed for generation varied between 4 and 10 hours using an A40 with a 48GB memory.

Coarse Shape Generation:

In this case we only used newer GPUs. The main reason for that was that it was not possible to scale down the parameters such that the older GPUs would be capable of generating coarse shapes.

Trying to execute a coarse shape generation with the default parameters resulted in us getting an out of memory errors very quickly. This was not surprising, so we tried to scale down the parameters similar to what we did in the previous section. Unfortunately this was not successful as the generation still could not be completed due to out of memory errors. This, along with the fact that this generation is more generic and will be used to create numerous different avatars so it needs to be of sufficient quality, and the fact that the authors did not provide in this case a scaled down .conf file to be used for the generation, led us to not use older GPUs for this purpose as we felt it was not fitting.

This type of generation was more resource demanding. The default value of the number of iterations to generate a coarse shape was 300,000 which in our case required 40-50 hours of calculation. Similar to the previous case it is possible to see during the calculation how the coarse shape is looking and interrupt it if the quality of the result seems sufficient. In this case the relevant pictures are stored in AvatarCLIP/AvatarGen/AppearanceGen/exp/smpl/base_model/<base model name>/**validations_fine**.

Here is an example (In this example we generated an obese man):

The right part of the picture shows how the coarse shape "should look" which is based on the output of render.py, and the left part shows how it is currently looking. When the quality of the left part seems sufficient one can interrupt the generation.

Once the generation of the coarse shape has concluded it can be used to create avatars similar to the previous section. The generation process is almost

identical except it uses a different template. One important thing to note is that when generating an avatar using a coarse shape it is required in its .conf file to provide a checkpoint from the generation of the coarse shape. Checkpoint are created by default after every 10,000 iterations (This can be changed in the .conf file of the coarse shape itself), so if the coarse shape generation was interrupted you cannot use the default value which points at the checkpoint that is generated after the 300,000th iteration.

Avatar2FBX:

As we mentioned earlier the output of the generation is in the .ply file format. The .ply format is a simple format for describing an object as a polygonal model. Unfortunately .ply objects are not currently supported by the Unity engine.

Therefore, the repository includes code that can be used in order to convert the .ply format to an .fbx format. .fbx files, unlike .ply files are widely supported in different softwares including Unity.

The readme file includes simple instructions for the conversion. Unfortunately, the conversion is quite problematic. The main issue is that after the conversion is finished the avatars are completely colorless. That is, they are completely white. This makes this module unsuitable for our needs.

It is important to notice that the .ply avatars do contain color as is confirmed by loading those files in to softwares that support them (Blender for example). This confirms that the problem is with the Avatar2FBX part of the repository, and not with the generation itself.

An explanation of how we dealt with this issue is provided later.

Results:

Regular avatars (Not using coarse shape generation):

Note: The results that are discussed here are the results of generations that were done with the parameter values that were mentioned above (default for newer GPUs and with the changes we mentioned for older GPUs) and were allowed the full 30,000 iterations.

We would like to stress that we have generated many avatars but we didn't want to overload this document with dozens of images. For that reason there are many examples we couldn't show and we chose examples that we feel that best represented the general qualities and limitations in each case.

Older GPUs:

Expectedly in this case the results are not as impressive as the results that are presented in the paper and in the project website.

Below are some of the higher quality avatars:

Result for the input "Jesus Christ":

Result for the input "Demon":

Result for the input "Farmer":

These avatars are looking quite good at first glance. However most of the time the results were not that good looking. Even when looking at these results which are truly some of the best of the bunch some problems and limitations are quite noticeable. More iterations do not fix those issues (this is not surprising because the learning rate at the end is already very small). Increasing the learning rate will not fix these issues as well, as will be discussed later.

First of all, their feet are not looking good but rather quite unfinished and lacking in detail. The faces could use a little work as in some cases they seem quite blurry and lifeless. This is especially evident when looking at the farmer. Some color seems to be missing in some parts. Transitions are not exactly sharp – try to find out where the shirt of farmer ends and where the arm begins. The structure of the palms looks bad at times: fingers are missing and instead we can spot some lumps. The farmer seems to be wearing some sort of a hat but we can barely tell. Something about the legs and the body of the demon is not quite right.

The reason we focus on there shortcomings is that with the good examples of the older GPUs it is much easier to show problems that are still relevant even with better hardware with a small amount of examples (whereas had we looked at newer examples we'd need many more examples to show the same problems, and with worse examples it would be hard to even point at these specific problems). When we used better hardware it was much less likely to face any of these problems, however, it still remained a possibility. By that we mean for example that if when using older GPUs the palms were buggy in 50% of the generations, with the newer GPUs the percentage decreased to 10% which is still something that should be noted. In short better hardware will make those problems less prevalent but won't fix them completely.

We'll focus on these problems and more in the limitations section.

We would also like to present more typical results when using older GPUs (Results for Lincoln, Chef, Detective and Waiter):

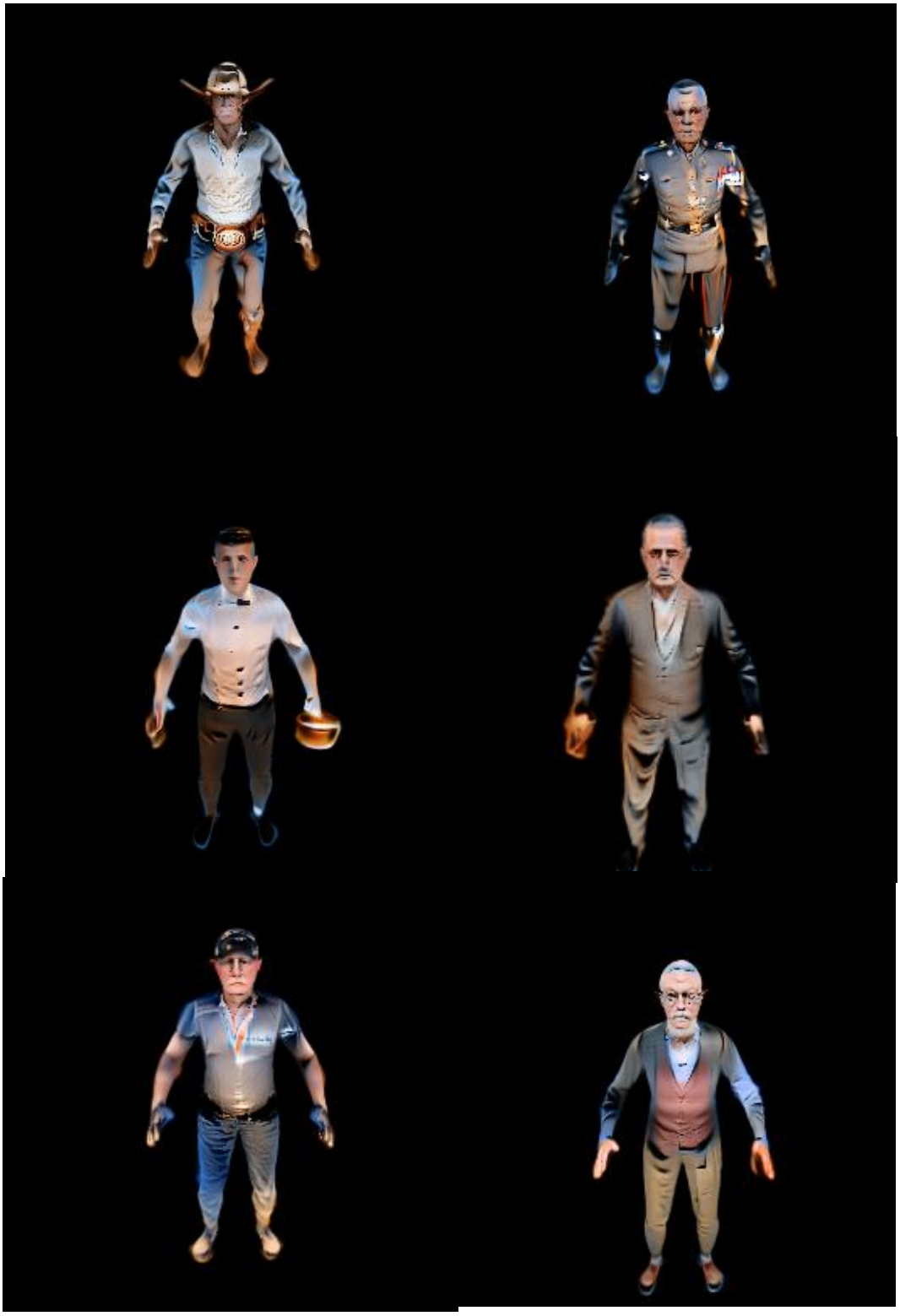
We won’t focus too much on these examples other than displaying them because here the main problem is the lack of general depth which is very evident and not really relevant with the newer GPUs.

We would like to stress that older GPUs are far from being useless but it would be wise to use them to generate more specific types of avatars. If the facial features of the avatar are not that important, or if the avatar has some unique colors that make it easily recognizable, or if the avatar has some other unique characteristics such as facial structure – using older hardware may be sufficient as the result is still likely to be recognizable which will make it more usable. This is more relevant for fictional creatures in many cases such as superheroes, aliens and monsters.

Newer GPUs:

Using better GPUs resulted in significantly better result. This is not surprising as using older GPUs required us to scale down the network which hurt the quality of the avatars greatly With newer GPUs we were able to use the recommended values for all the different parameters.

Below are some typical examples: Cowboy, General, Waiter, Vito Corleone, Truck Driver, Professor).



The difference between the quality of the average avatar using an older GPU and the quality of the average avatar using a newer GPU cannot be overstated. Earlier we provided a generation of a waiter under an older GPU. Now we are able to compare this example with a new, much improved waiter:

Some of the easily noticeable improvements are:

- An improvement in the hair quality. When looking at older GPUs' generations the hair almost always seemed extremely blurry and lacking in texture. The new hair seems much more vibrant and natural.

- An improvement in the quality of many facial features. This includes eyes which are barely visible when looking at the original waiter, ears which are much sharper and more distinctive, nose and mouth. The facial features are the most important attributes because if they are not of high quality the avatar will not look human, which is part of the reason we recommend the older GPUs for avatars that are not really human like superheroes and monsters. Facial features matter less in those cases.

- In case of avatars that are holding objects like the waiter we can see that the object seems much clearer. It also actually seems like the waiter is holding the object rather than the it being attached to his hand in some way, which is the way it is looking in the older picture.

- Clothing is much more colorful and natural looking. This is noticeable in the waiter but an even better example is the professor. We can easily tell that he is

wearing several layers of clothing, each with its own unique color, shape and texture.

- Feet are much more distinctive. Looking at the older picture we can't really tell where the feet start. Shoes are also not recognizable. This is not the case with the newer waiter – we can easily tell he is wearing black shoes and that he actually has feet.

- Looking at the Cowboy we could see that the problem that we originally faced with hats (or other similar objects like a crown) is less noticeable. The cowboy's hat is immediately recognizable and is rich in color and texture (previously when we looked at the farmer we could barely tell he was wearing a hat).

Most of the avatars were certainly good enough to be used in our scene. Unfortunately some generations didn't produce such high quality results. This will be discussed in the limitaions section.

You should change the default value of some parameters depending on exactly what you want to achieve. If you would like the best looking avatars to be generated and you have the required resources to make it possible you shouldn't tune the parameters. However, if you wish to make the generation faster and are willing to sacrifice a little bit the quality of the final avatar we encourage you to increase the learning rate parameter value. For example, we noticed that when the learning rate is doubled a quite good looking, distinctive avatar is generated after as little as 5,000 iterations).

Below is a typical example of the effect that tuning the learning rate achieves (In the left image is the original professor with the default value of 5*10^-4. In the right image is the professor with the value of the learning rate being doubled):

It is not difficult to notice that the original professor is looking slightly better. His clothes are looking more realistic, unlike the new professor which seems like he is wearing a jacket only on one side of his body. The facial features of the original professor are also looking slightly sharper and more distinctive.

However, when comparing the professors after only 5,000 iterations we notice that things are quite different:

In this case the professor that was generated using a larger learning rate is looking significantly closer to being finished.

Allowing for more iterations to take place is usually not beneficial and is very wasteful in resources as usually each iteration takes the same amount of time (the exact duration of each iteration of course depends on the hardware used).

Results for coarse shape generation:

Note: The results that are discussed here are the results of generations that were done with the default parameter values and were allowed the full 30,000 iterations for the avatar generation and the full 300,000 for the coarse shape generation.

In this case the model produced impressive results as well. Below are the outputs of “Obese Professor” and “Obese Cowboy”:

We can see that both of them are still instantly recognizable with a similar level of detail comparing to the original professor and cowboy that were generated using newer GPUs. It is also very clear that they are overweight as they are significantly wider than the previous figures. It is also noticeable that their clothing still sits right on them and is not looking distorted or unnatural.

Limitations:

In this section we'll focus on the limitations on the newer GPUs for the reason that these limitations are more relevant to the limitations of the project rather than the limitations of the hardware as is the case with the older GPUs.

Note: We would like to stress that most generations ended successfully and the resulting avatars were of high enough quality to be used in the scene. However it was still important to us to focus on a few problems:

Lack of consistency. Different executions of the same avatar with the same parameters may produce significantly different results.

Here are two result of the generation of a "Prisoner":

1st generation of a prisoner (was terminated prior to finish)	2nd generation of a prisoner (was terminated prior to finish)

Lack of symmetry, especially when looking at the avatar's legs. The following pictures are taken from the official project site:

It is quite noticeable that these avatars are not quite symmetric. Another example is the avatar that we generated with the input "Fashion Designer":

Facial features are limited: It is quite evident by looking at the numerous examples we provided, including the avatars that were taken from the project site itself, that the facial features in some cases are lacking. Hair often is lacking the texture or color that would have made it appear more realistic. Eyes at times are barely visible and lack color. Ears, mouth and nose quality also highly vary between different avatars.

The model struggles with avatars that are wearing something on their head:

While the model generally did a good job with avatars that are wearing simple hats (like the cowboy and the truck driver that were shown before) it did struggle with avatars that wear less trivial objects.

This is evident by looking at the examples below:

A larger number of iterations may hurt the final result:

We'd hope that the more we allow the model the better the results would be. Of course it is unreasonable to expect anything close to a linear improvement, but we'd still expect the avatars to be slightly more refined and natural looking with more iterations and computation time especially when not a large amount of iterations were executed. However, this is not always the case.

Here is an example where it is clear that the execution of more iteration has significantly decreased the quality of the result:

A "car mechanic" after the 10,000th

A "car mechanic" after the 30,000th

While some details are looking slightly better the avatar barely seems human anymore which means we spent a lot of iterations and therefore resources for nothing.

The final avatar may look like a "mix" of two different avatars

This is best explained by looking at an example. Walter White is a fictional character that can be associated with the yellow full body suit he was wearing in many instances through out the show as can be seen here:

The final result of the generation was:

Which although recognizable is clearly unnatural as it is clearly a mix of his full body suit and his regular clothing.

Avatars are generated with two faces:

This is quite rare but it’s still a bug that certainly happens, making the generated avatar worthless. It is also an example of why it is important to have

a look at the pictures in validation_extra_fine every few hours, because terminating such generation will allow to save resources.

Here are some examples of this problem (these run were not allowed the full 30,000 iterations which is why they are looking unfinished):

We can see that the avatars seem to grow another face on their body. In these cases we recommend trying similar input as it might solve the problem

Unity:

Unity is a graphic engine developed by Unity Technologies that is used for developing video games for a variety of platforms. Unity combines between C# for creating the logic of the game, the graphics, and sound to create the appropriate environment for the character depending on the developer's vision.

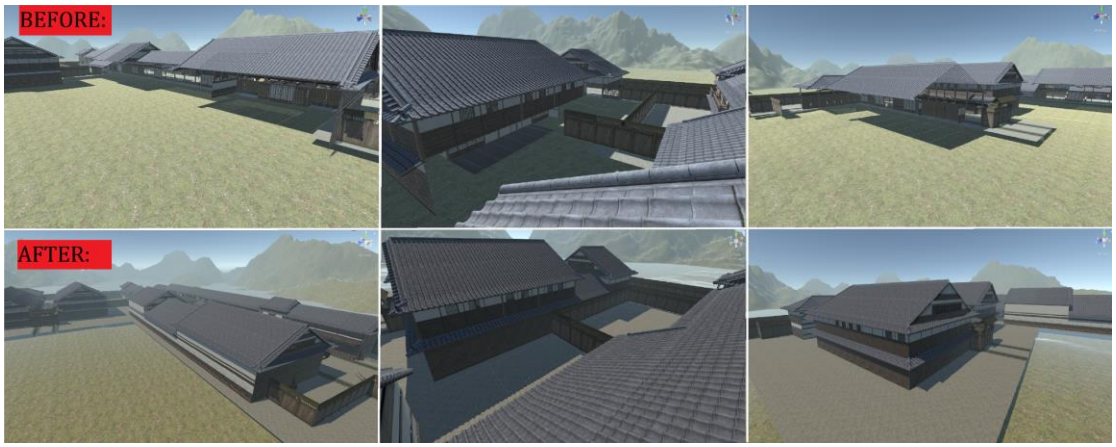
Unity asset store:

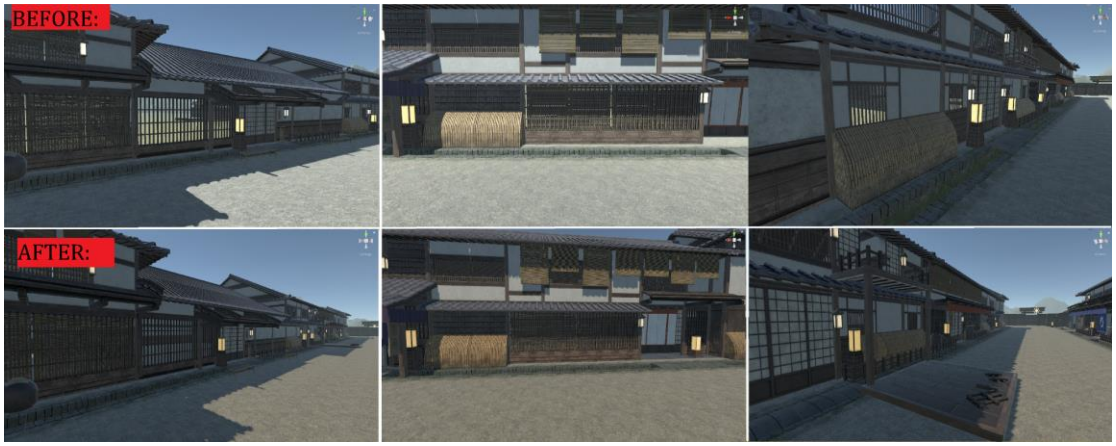
Assets created by users can be developed and sold to other users through the store. We used the store to buy the following environment: https://assetstore.unity.com/packages/3d/environments/historic/japanese-village-kit-33052

In this environment the avatars walk and the scene takes place.

The Japanese village kit:

We bought the environment at a cost of 10 dollars from the Unity store. We were looking for a town with a central street so that the Avatars could walk there and perform the various interactions (we will elaborate on this topic later). Unfortunately, it was only after we had already bought the asset that we discovered that the map was not built completely. The interior of the buildings were empty without any basic furniture and the exterior walls of the houses were not visible to the main street and were transparent. We fixed this by using parts from other houses. In addition, in a certain area of the map there were no buildings at all, so we built an entire house from scratch to complete the map. Here are some pictures before and after some of our changes:





Nav-mesh:

Nav-mesh is the system we use in our project as a solution to movement and collisions between avatars. the idea is to bake the environment and select plots representing where we allow the avatar to walk and where not (for example we don’t want that an avatar will go through a wall, fence and so on). To achieve this while making sure avatars don't bump into each other, Nav-mesh gives us the best solution. After we bake our environment, the avatars can only walk on surfaces we want and in addition the shortest path to the destination is calculated. The destination is random, and when the avatar arrives at the destination the script randomly chooses a new destination (by selecting new legal coordinates). The best part about Nav-mesh is the collision prevention. Every wall or fence or object we define as an obstacle to our Nav-mesh which is static. Our avatars and players are defined as dynamic obstacles so that the Nav-mesh calculates the shortest path while avoiding static obstacles and in real time recalculates the path in case there are dynamic obstacles in the way.

Event:

We have set an event that invokes when an avatar falls from the balcony of one of the buildings. A defined group of avatars will stop their current actions and gather around that fallen avatar. We did this by using the Unity Event System, so that as soon as the trigger of the avatar's fall is activated (the avatar falls from the balcony and crosses a trigger that detects his fall and invokes the event), the pre-marked avatars stop their random walk and gather around him. If the avatar didn’t fall, the other avatars will continue their affairs until and if the event invokes.

Animations:

We use Unity Animation System in order to animate our avatars, for example most of our avatars have walking animations in the scene with different speeds (some walk slowly and some fast), some avatars have static animations like standing, sitting or praying and one avatar even sits and yells at someone, and some are dancing in one of the buildings with some music.

- Music: We detect how far away the player is from the dance floor and when he is approaching near the music becomes louder, and quieter when the player moves away from it.
- Dynamic Animations: we use a combination of the animations in order to create a new one which contains all of them with smooth transitions, like our dancing avatars that dance 3 different dances in a loop one after one.

The animation system in Unity is a lot more robust than that. It can be used, for example, to move the camera in a specific path, to make transitions between scenes and so on. For our purposes we used a site called mixamo to download pre-prepared animations to our avatars.

VR:

The user/player experiences our scene through VR headset, so we tried hard in order to make sure that the experience will be as best as possible. With the new Input action-based system, we provide to the user the following options: walk with the thumbstick of the controllers, teleport to a selected (legal) location with the trigger on the controllers, jump with the right primary button controller and even fly like superman with the left primary button controller. Every player can select the way he wants to interact with the digital world and combine them how he wants.

Problems that we encountered while working on our project in Unity:

As all projects we faced some problems. First of all,all of us never worked with Unity before so we had to learn everything from the beginning, and as a result we made many beginner's mistakes that we do not need to explain. We were stuck on some errors and problems for a long time and even needed to start the entire project from the beginning. We will describe the 2 main problems that took us most time and were the hardest to solve:

1. The first problem was when creating the avatars from avatars's generator and importing it to unity. The avatars were imported with no color (just white texture). We found out that the avatars were generated in .ply format and Unity works only with .fbx or .obj format, so we found some online sites that convert .ply to .fbx. But by converting this way, we lost the colors of the avatars, so we searched for others sites that didn't make the avatar lose its color (we checked all possible sites and programs that convert from .ply to .fbx) and almost all of them convert it with no colors until we found this site: imagetostl which converts the avatars with their colors. But then we ran into a new problem - the avatars became really heavy (running the scene with just 2-3 avatars our average fps rate was 3). After some research we found out that our avatars have a few thousands of submeshes for every color material separately. So, when we ran the scene we got an absurd amount of drawn triangles and vertices which caused all resources to be spent on rendering the avatars (we measure that with a Unity analysis tool called profiler, which tracks the numbers of triangles and vertices in the scene along with CPU and GPU usage and in addition track the FPS in the scene), which was the reason for the low fps we faced. The solution to this was to get the avatar from an avatar generator in .ply format. To do this we needed to bake it in a program called Blender in order to create a single mesh and material for the avatar and download the avatar with all the materials in it in .fbx format, then import it into mixamo for rigging and finally adding it to the scene with colors and no low fps rates.

*A video detailing the process described is attached below:
https://drive.google.com/file/d/1Jbrf_Xa9B2QfoCvbGXovUFATCVIrkgKk/view?usp=sharing

2. The second problem was again with the avatars. Firstly, we need to mention that our goal was to export the scene as a .apk file to the headset, and run it on the headset itself. We realized that after adding more than 7-8 avatars to the scene we ran into weird situations when we looked at a group of avatars gathered in one place. The camera of the headset disorientates and becomes very laggy (if we change the direction of the view direction to a wall or floor it goes back to normal). we found out that the hardware of the headset is simply not enough for that amount of avatars (to be consistent with the previous problem, these lags originating from the avatars is unfixable, as we tested the same scene with avatars from other avatars generators sites and ran into the same problem). The solution that we found to that is instead of creating .apk file of the scene and running it on the headset itself, we run it on a computer with compatible hardware, that can run the scene with the amount of avatars we want, and stream it to the headset that is connected to the computer. This method proved to work.