Matrix in Real Time

By Artem Shtefan and David Koplevatskii

Supervisors: Boaz Sternerfeld and Aaron Honen
Introduction

Considering existing matrix effect shader that was built as one of the previous projects of the lab, the main goal of this project is to apply the digital rain effect on the mesh that was created in real time using some kind of sensor, meaning - to move the matrix effect to AR.

As we decided to use Unity as a base engine for the shader itself, we tried few options on different platforms, including Android and iOS. Using one of the recent iPhone models with LiDAR sensor we got the best results in terms of mesh building and rendering time and the quality of the scanned mesh itself, but we faced one serious problem. The mesh generated by LiDAR does not support UV coordinates - and that means that usual shader building techniques wouldn’t work. The solution that worked both towards avoiding UV coordinates and improving performance - triplanar shader.

After a long story of failures - first ARKit didn’t work, then LiDAR mesh generation was not working for a while, after that there were quite a few tries to make the triplanar shader work. We didn’t believe when we finally put this together. The application uses AR Foundation which uses updated ARKit on iPhones. This allows us to access and use the sensor API and receive a mesh. The mesh receives a shader and the shader itself is being managed through a script (we need to update the raindrops falling).

Other options were not as successful. On Android we can’t use the same component and could only identify a very basic planes using the basic AR Foundation and ARCore components. This is why we consider the obtained result the best we could achieve.
LiDAR & Unity

One of the central pieces in our project is the LiDAR sensor. LiDAR is an acronym of “laser imaging, detection, and ranging”. It is a technology which basically consists of a pulsed laser, sensor and an algorithm. A pulsed laser records the time it takes (in order of nano-seconds) for that signal to return to source, enabling it to generate a 3D model with greater accuracy than just a simple camera.

LiDAR laser visualization

This kind of technology provided us with almost state-of-art scanned mesh in the real-time and that is fascinating.

We incorporated the usage of the scanner in iPhones using the ARMeshManager component in AR foundation.
Shader implementation

The shader we used is based on the shader written by Bill Kirby (https://www.shadertoy.com/user/WillKirkby) and adapted by Shahriar Shahrabi (https://shahriyarshahrabi.medium.com/). The original shader on GLSL consists of two simple functions: rain, which provides us with rain dropping effect and text, which generates the random letters. The shader was later adapted to Unity’s HLSL/ShaderLab and GPU random numbers generation was added instead of pseudo-random numbers generated from a white-noise picture. We also introduced a few minor changes to adapt the shader for AR. Below is the general breakdown and purpose of those components.

1. The rain function

   This function generates a matrix that is filled with either green or black pixels by blocks in our pre-defined columns. With each frame green blocks are being skewed down and new ones generated at the top. This is how we receive a raindrop effect. We can use this matrix as a mask to paint our pixel black or green accordingly. If the value is 0, multiplication will give us the black color, otherwise the pixel belongs to a block and will light up.

2. The text function

   This function generates random numbers in our predefined block matrix. The output is essentially just a table of letters randomly changing each frame. First, for a given pixel, we identify the corresponding row and column. After the appropriate block id has been found, pixel receives color based on the letter generated in that block.

   The letters themselves are being generated by a GPU command buffer that is being deployed at the start. Using simple white noise generation, we acquire new grid of letters each frame.
Triplanar mapping

1. Intro

In order to achieve the desired effect from the “Matrix” movie, our application should include two important parts, the first one being scanning the real world space and building the appropriate 3D model in the scene, and the second one - implementing the shader itself, basically drawing the digital rain animation with ShaderLab/HLSL code that Unity uses. The final step would be just applying the shader to the generated mesh, but this is where the main problem was lying - you just can’t use a regular shader with the mesh that LiDAR generates. The meshes that this scanner generates are very good both in terms of building time and precision, the best that we have achieved. Hence, to preserve the ability to use LiDAR and still be able to come up with something that recalls the original film in the shader itself, we needed to deepen our knowledge in the coordinates mapping in 3D.

2. Standard approach - UV coordinates

When we want to create a shader in the traditional meaning, we basically create something that replaces or modifies the texture of a 3D object. So, for the simplicity of the explanation, let’s assume we have a simple 3D cube with a mesh, and we want to apply a very simple image texture to it.

As we can see after a second of looking at the picture, the main problem to solve is how to apply a flat 2D image on 3D object with an
already complicated form (even though it’s just a cube). Which edges of the cube should display which parts of the original texture image?

Every mesh of a 3D object has vertices - that’s how we create complex shapes in 3D. So, the proposed solution is to try to cut and flatten a 3D model like an “origami” and lay out the resulting pieces on a texture. After that, check where the original vertices are on the resulting flat plane. New coordinates on that plane are called $u$ and $v$ - because $x, y$ and $z$ are already used in our original 3D space.

3. Problematic cases and alternative mappings
While this may seem like a pretty good intuitive solution, obviously there are some problems with it - i.e. is it always feasible to calculate and store these coordinates? For example, for any large scale 3D scene that includes real-world landscape frequently there is a need to texturize terrain, water or sky. Especially in case of terrain, it is a very large shape with a lot of details, such as dumps, mountains, caves, etc. If we were to use UV-mapping, we would have permanently stored coordinates for every point on the terrain mesh. Obviously, this is a lot of information to store (see image below).

Complicated terrain mesh

![Complicated terrain mesh](image1.png)

Distorted texture on a hill terrain

![Distorted texture on a hill terrain](image2.png)
Not only this approach requires a rather great amount of resources, sometimes it does not provide us with the best results.

There are alternative techniques to use instead of directly storing uv coordinates in all of the vertices. Using these techniques in the appropriate cases not only reduces amount of memory needed for storage, but also applies the original texture on the model with better precision and overall look. One of these techniques is called triplanar mapping. It is specifically useful with working with large scale surfaces like terrain or spheric shapes because it maps the texture seemingly without stretching and seams in the texture. The main thing is - this mapping solution doesn’t require us to keep any pre-calculated uv coordinates at all!

4. Triplanar mapping

So, what essentially is triplanar mapping? In our previous case, the uv coordinates that were stored in the model vertices directly pointed us where to go and where to get a certain polygon from the texture image. But now, assuming that we only have world position of a polygon (our usual x, y and z), we try to look at this polygon from the directions of X, Y and Z axes and try to approximate the uv’s from there. Formally speaking, we use three projections of the world space coordinates vector on the orthogonal planes and then try to blend them into a weighted average and that’s our uv coordinates.

Now, let us run shortly through the math of all of this. To project a vertex on the three standard planes is very simple - we just need to reduce one coordinate for the each projection. Then, we can use normal values almost directly as weights because they essentially tell us to which direction the polygon is turned more. That is exactly what we need, for example if the normal vector is pointing to the Z direction, the resulting color will be very
heavily consisted of the XY projection. Hence, we just normalize the vector and combine the resulting values in a weighted average.

5. How do we use it?
Previously, our shader consisted of two consequent calls to functions `rain()` and `text()`. Now, we just need to do the same for three projections and blend the result together.

```
// sample the texture
fixed4 col = float4(0.,0.,0.,1.);

// evaluate the resulting color
col.xyz = text(i.uv * float2(_screen_width, _screen_height)*scale)*
    rain(i.uv * float2(_screen_width, _screen_height)*scale);
```

As we can see, the number of the function calls has tripled, but in return pre-stored `uv`’s are not needed anymore. And now we can use our shader with the LiDAR.
Future work

A few possible improvements to our project could be:

• Adding a scripting component to identify and correct the direction of the letters.
• It is possible to run the shader in a few layers using multipass and this way add depth to the image and improve the overall look.
• LiDAR API for AR foundation & Unity supports classification of objects. Identification of humans and other complex entities will be very easy to implement. This can be used to customize the shader on these objects and also improve the look.
References:

Shader:
- https://shahriyarshahrabi.medium.com/shader-studies-matrix-effect-3d2ead3a84c5
- https://www.shadertoy.com/view/ldccW4

Triplanar:
- https://www.ronja-tutorials.com/post/010-triplanar-mapping/
- https://bgolus.medium.com/normal-mapping-for-a-triplanar-shader-10bf39dca05a
- http://www.martinpalko.com/triplanar-mapping/