Unity SRP for cartoon style rendering optimisations

Theo Lemoine
theo.lemoine0@gmail.com
CNAM-ENJMIN
France

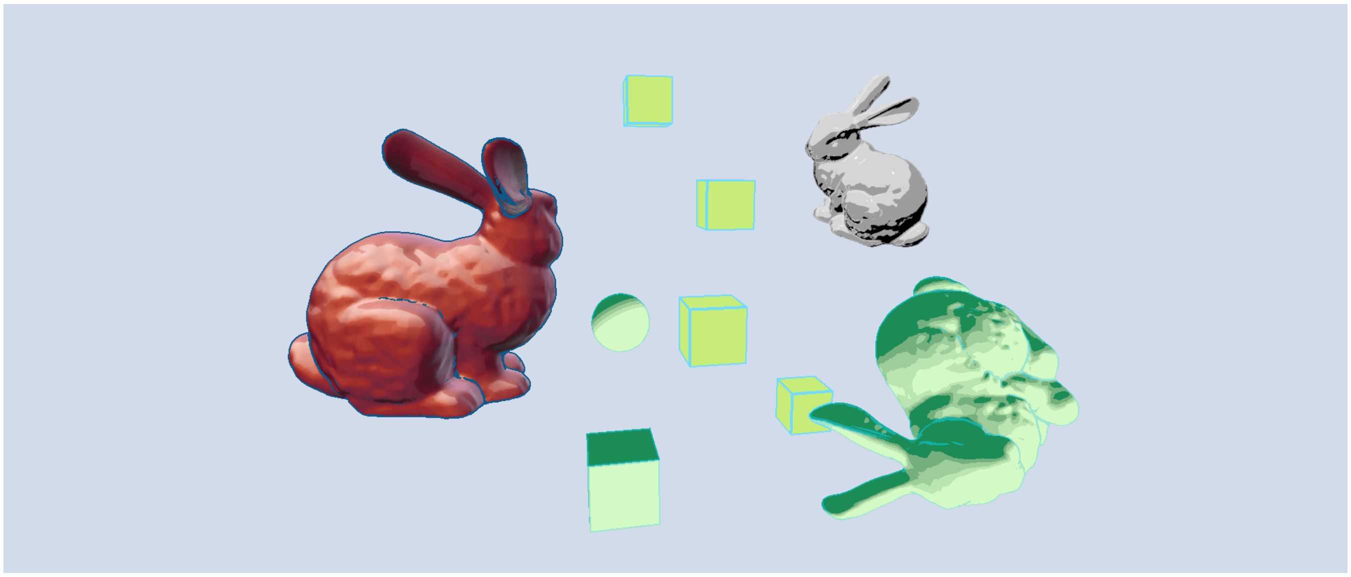


Figure 1. Objects rendered with a Custom Render Pipeline.

Abstract

This article aims to show the benefits of using the Unity SRP to build a custom rendering pipeline fitting specifically the needs of a project, instead of relying on the given High Definition and Universal Render Pipeline.

The goal of the project was to experiment with Unity SRP technology and to create a pipeline optimised specifically for cartoon style rendering.

I tried to achieve a cartoon-looking renderer with the smallest performance cost possible with multiple techniques such as material capture, Multiple Target Rendering, screen-space edge detection or simply limiting lighting computations to one light to prevent inefficient loops in shader code.

CCS Concepts: • **Computing methodologies** → **Rendering**.

Keywords: Unity, Scriptable Render Pipeline, Non Realistic Rendering

ACM Reference Format:
Theo Lemoine. 2021. Unity SRP for cartoon style rendering optimisations. In *Proceedings of CNAM-ENJMIN Experimentation Subject*. Angoulême, France, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Motivation

Video games often use similar rendering techniques (PBR shading, Shadow Casting, Post Processing, ...), especially when the goal is photorealism.

The goal of game engines, such as Unity is to provide standardised tools to use said techniques easily and create the wanted look for your game. But this often means the renderer is "one size fits all" and it's hardly possible to opt-out of some functionalities, or access low level code to add your custom behaviour to the renderer.

Especially for low end devices such as smartphones, or portable consoles like the switch, where hardware is not that powerful, optimizing rendering is really important. As it's often one of the most costly parts of the game in terms of performance. Building your own rendering code fitting the specific need of the game is often way more efficient.

The "new" Scriptable Render Pipeline aims to fix the problem by letting developers create their own render pipeline and access low level rendering code to set up render passes, handle GPU buffers, etc.

1.2 Problems with HDRP and URP

SRP comes with 2 standardised pipelines made and maintained by Unity : the High Definition Render Pipeline for high performance, photorealistic games ; and the Universal Render Pipeline, a more versatile, low end device friendly pipeline.

But both pipelines make it really hard to access rendering code. The shader graph, a visual tool to create shaders, is the only way to have custom rendering behaviour and still has no access to lighting computations or shadows.

1.3 Non realistic rendering in video games

Moreover, some game styles do not fit in the realistic shaders and render processes given by the HDRP or the URP. It's often possible to work around and find solutions, but the given render pipelines then becomes more of a burden than a real help.

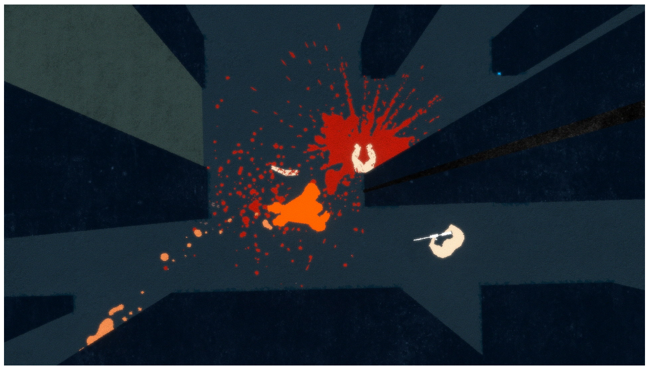


Figure 2. Ape out, renders all 3D geometry with an unlit shader

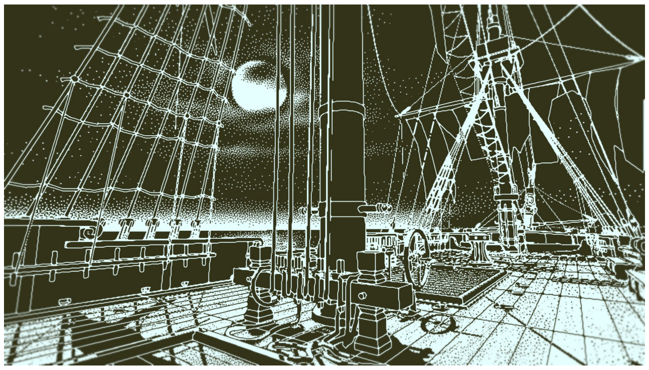


Figure 3. The return of the Obra Dinn, 2 colors only, with cell shading and edge detection

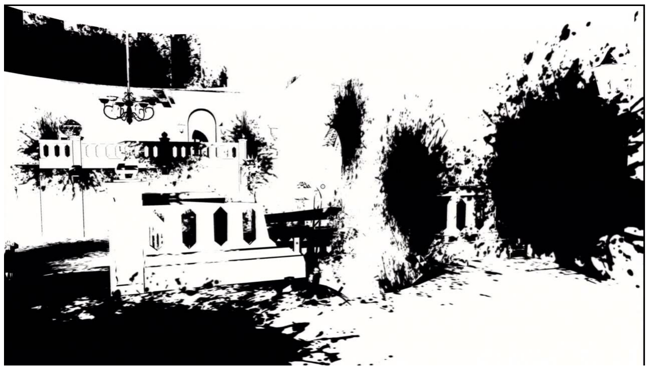


Figure 4. The unfinished swan, you paint you surroundings by throwing paint

So I thought I would try to experiment what was possible with a custom SRP in regards to this kind of non-realistic rendering. What I was going for was a drawn comics-like style, with flat colors, and strong edges.

2 Creating a new render pipeline

[2]

2.1 Introduction to Unity SRP

Unity Scriptable Render Pipeline allows developers to specify the rendering process of the engine using C# code to schedule draw calls, and handle GPU memory buffers through Render Textures. Developers can also make their own shaders using the SRP Shader Library, containing the base functions like applying the Model View and Projection matrices given by unity.

2.2 Creating a C# renderer

Creating a custom Scriptable Render Pipeline is done by creating 2 C# classes : the `RenderPipelineAsset`, and the `RenderPipeline`.

The `RenderPipelineAsset` works in a way close to scriptable objects, Unity makes it possible to instantiate the class as an asset in the editor and you can define members of the class that appear in the inspector just as `MonoBehaviours`.

Data in the asset is then used (if needed) to construct a `RenderPipeline` instance when the game runs. The RenderPipeline can then schedule draw calls to the GPU, allocate render targets, and pass data globally to shaders.

This is done using both the `ScriptableRenderContext` and `CommandBuffers`.

The `ScriptableRenderContext` class acts like a bridge between our C# render pipeline code and the low level C++ code Unity uses for rendering. It allows us to do culling, draw the skybox, the geometry of the scene, etc.

`CommandBuffers` are objects containing multiple instructions to be made by the GPU. The rendering code is not directly run from the C# side, we fill a buffer with a list of instructions, and ask the `ScriptableRenderContext` to execute them all at once. `CommandBuffers` are used for things like allocating render targets, clearing them, setting the render target to be rendered to, or sending global data to shaders.

2.3 Using Unity SRP tools and Shader library to build custom shaders

Writing a custom shader for a render pipeline is made easier thanks to the SRP Shader Library. It allows access to utility functions to transform vertex from object space to World, View, and Clip space, or to transform normals from object to world space.

To render our shader properly we need to define a `LightMode` both in the C# and the shader passes. The `LightMode` is a keyword for shaders defining on what type of pass they belong : is this pass for shadow casting, depth check or final mesh rendering ? Once the keyword is defined, on the C# side, we can decide which type of `LightMode` will be allowed when we render our geometry. This allows the C# part to set up the right shader properties and render targets when running a specific part of the pipeline.

In the shader :

```
Shader "CRP/Unlit/Matcap" {
  Properties {
    ...
  }
  SubShader {
    Pass {
      Tags { "LightMode" = "CRPUnlit" }
      ...
    }
  }
}
```

In the C# renderer :

```
ShaderTagId UnlitPassId =
  new ShaderTagId("CRPUnlit");
var drawingSettings =
  new DrawingSettings(
    UnlitPassId,
    ...
  );
```

Apart from this this is just like a standard unity shader with a vertex and fragment core.

The SRP allows developers to access most of the rendering process of the unity engine, while giving useful tools for developers to create their own rendering pipeline. This allows for a lot of optimisations, especially for games who aim for really specific, and non-realistic visuals. Now that I've explained how creating a custom SRP works, I'll be diving into some techniques I tried, to create a drawn comics-style rendering.

3 Techniques for non realistic rendering

3.1 Using Material capture to fake lighting with unlit shaders

[3]

The first thing was to try to create efficient and good looking unlit materials using the material capture (MatCap) technique.

MatCap is a technique used to get a non-realistic imitation of a material at a really low performance cost. Usually a sphere is rendered with the original material with a given lighting and reflection setup. The image of the sphere is then saved as a small texture (usually 512*512).

When rendering, the shader will use the normals of the model, convert them into view-space, keep only the X and Y part of the vector, and remap them from -1,1 to 0,1. This vector is then used as UV coordinates to sample the sphere texture.

In the end, the sphere is used as a lookup texture to "what is the color of the material at this angle from the camera". Of course, the technique is only an approximation of the base material (see fig.5), but works really well to believably simulate metallic reflections on an object at a really low cost in performance.

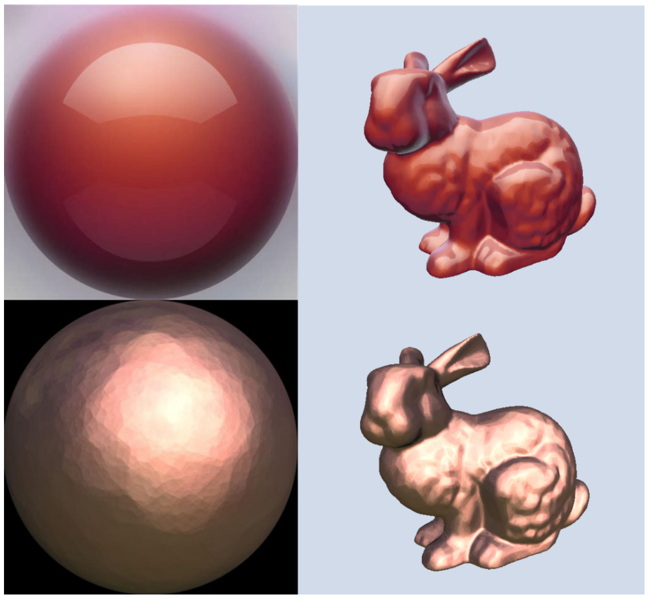


Figure 5. MatCap technique applied to a mesh

This technique was even used in Mario 64 to create the metal-Mario effect. Simulating metallic reflections on the mesh.

Although the sphere texture is usually the capture of a real material in a 3D engine, it can also be manually edited to create the wanted effect on the final object. The sphere texture can be drawn by an artist on a 2D software, to create the look of a drawn or comics-style material.

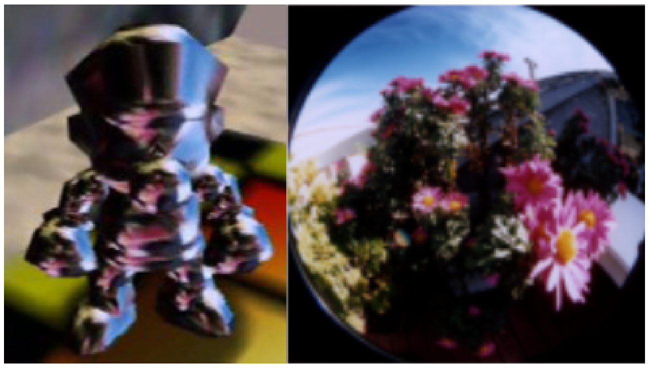


Figure 6. MatCap used for metal Mario in Mario 64

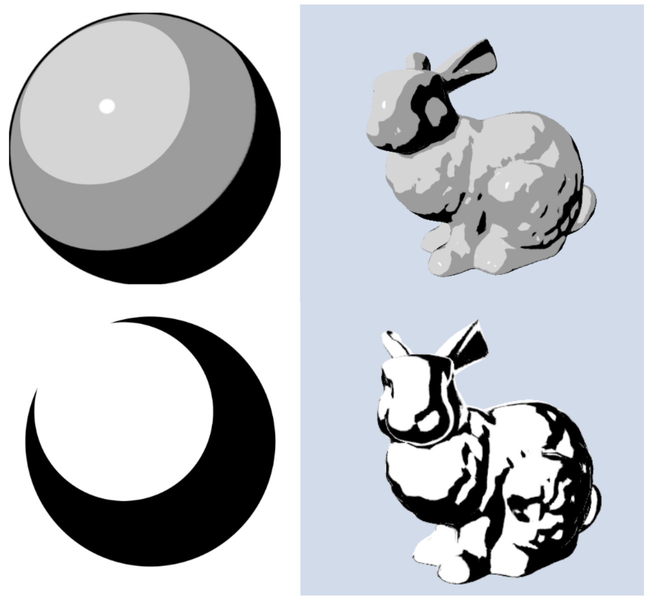


Figure 7. Examples of drawn MatCap texture, imitating painted or drawn shaders

This allows to replicate a drawn-comics style with no light calculations whatsoever. The cost of such a shader is really low and can be used to get a good looking effect on really low-end devices, or in performance heavy contexts, such as particle system shading, where light calculations would be too heavy.

Combined with color and normal texture mapping, this technique can be used to simulate complex lighting and reflections without the need of any light computations.

However, this effect can look strange if the camera angle changes a lot, as it becomes clear the lighting is fake. So MatCap is better applied to games with a fixed camera angle such as the top view of RTS or MOBA games.

3.2 Lit shading with only one light

For games with a fast moving camera, such as FPS or TPS, it is better to have consistent lighting. However, for cartoon aesthetics, one directional light is often enough to get a good lighting effect using cell shading.

Unity allows to get the strongest light in the current scene as `RenderSettings.sun`, its direction can then be transmitted to all shaders as a float3. To create a cell shading effect, we only need the direction of the light, the color of the object and it's shadow will be set on the material.

On the shader side we simply need to compute the dot product between the normal and the direction of the light. This dot product ranges from 1 to -1, 1 is fully lit and from 0 is in shadow. Standard cell shading would apply uniform shadowing only from 0, instead of using the dot product as a gradient.

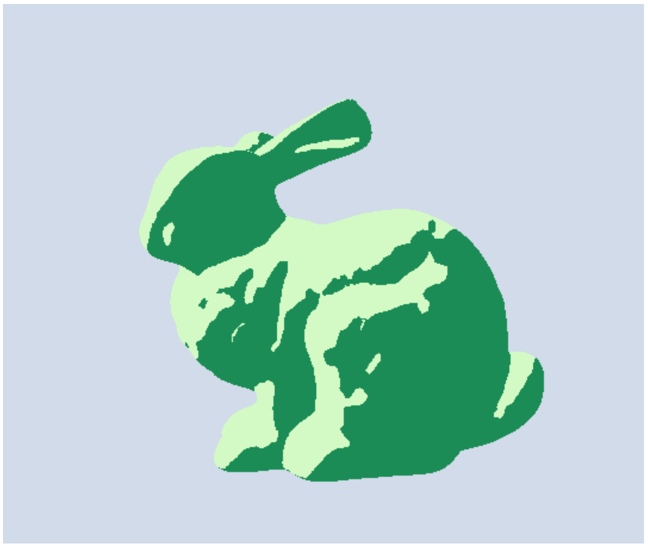


Figure 8. Standard Cell shading

To make a multiple step cell shading as we want to, we need to define the number of steps, and the size of a step. We take the values of the dot product from 0 to -1, negate them, and divide by the size of a step. the integer part gives us the step we are on, minus one, so we just `ceil()` the value. After we divide by the total number of steps to get the value of the shadow at the current step between 0 and 1.

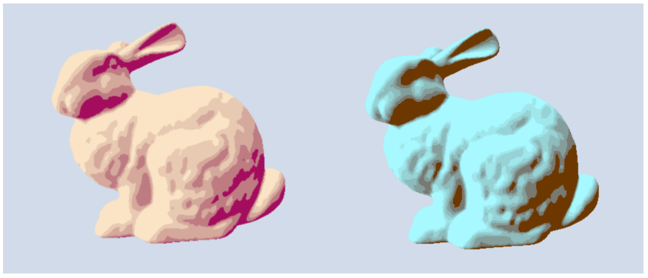


Figure 9. Cell shading with 3 (left) and 6 (right) steps

Having full control on the pipeline allows us to limit light computation to just what we need, hence optimizing the shader code, especially since we have only one light, we get rid of any loops that might slow down rendering.

3.3 Line rendering with edge detection

An important part of the drawn comics-style is the lines. I tried to achieve this effect through screen-space edge detection, using the sobel filter.

But instead of detecting edges on the color of the rendered scene, I wanted to run the filter on the depth and normals. So I needed access to multiple buffers containing the needed information. [1]

3.3.1 MRT - Multiple render textures. The goal was to create a rendering process inspired from deferred rendering : Surfaces would be rendered in a forward-rendering way, and edges would be rendered in deferred, just like a post processing effect.

In deferred rendering, the geometry of the scene : color, normals, and other information, is rendered in separate buffers called the geometry buffer. My first idea to do this was to run multiple passes to gather wanted information. But I learned that one shader pass could render into multiple frame buffers, what is called MRT : Multiple Render Target.

On the shader side, we need to define a structure for the output of the fragment shader, just like we usually do with the vertex shader. Each member of the struct has a semantic, like `COLOR1` or `COLOR2`, indicating which render target the data is meant to be written to.

```
struct Pixels
{
  float3 color : COLOR0;
  float3 normals : COLOR1;
  float4 borderColor : COLOR2;
};
```

On the C# side, we need to allocate temporary render textures for each type of data we need out of the shader, And assign them all to be rendered at the same time with the RenderTargetBinding class.

```
// setup RTs to be rendered to
RenderTargetIdentifier[] targets = {
  ColorRT,
  NormalsRT,
  BorderColorRT
};
RenderBufferLoadAction[] loadActions = {
  RenderBufferLoadAction.DontCare,
  RenderBufferLoadAction.DontCare,
  RenderBufferLoadAction.DontCare
};
RenderBufferStoreAction[] storeActions = {
  RenderBufferStoreAction.Store,
  RenderBufferStoreAction.Store,
  RenderBufferStoreAction.Store
};
var rtBinding = new RenderTargetBinding(
  targets, loadActions, storeActions,
  DepthRT,
  RenderBufferLoadAction.DontCare,
  RenderBufferStoreAction.Store
);

_buffer.SetRenderTarget(rtBinding);
```

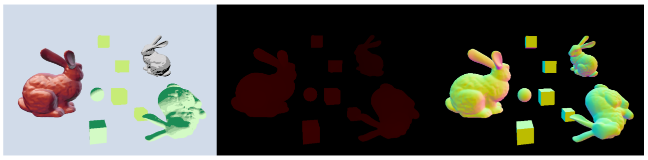


Figure 10. Color, Depth And Normals rendered at the same time

3.3.2 Blit to the screen. Now that the geometry is rendered to a buffer, nothing is rendered to the screen. We need to use the `Blit()` function to render directly to the screen. Blit works by feeding a few vertices to the screen and letting the vertex shader create a quad or a triangle covering the screen without the need for MVP matrixes. The shader also computes the UV for each point and sends it to the fragment shader where we can truly create our screen space effects.

```
Varyings ScreenVertex(Attributes input)
{
  Varyings output;
  output.positionCS =
    GetQuadVertexPosition(input.vertexID);
  output.positionCS.xy =
    output.positionCS.xy * 2 - 1;
  output.texcoord =
    GetQuadTexCoord(input.vertexID);
  return output;
}
```

3.3.3 Line detection with sobel filter. After setting all the textures for our blit shader with C# code, we need to apply a sobel filter to the depth and normal buffers. The sobel filter works by combining the results of two convolution matrices applied to the image.

A convolution matrix is a 3*3 matrix used to modify an image. Considering the current pixel is at the center of the matrix, and the other values are the surrounding pixels, the new value of the pixel is the sum of all the pixels multiplied by the corresponding coefficient in the matrix.

$$S_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \quad S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

After applying the 2 matrices, we can see the results as how much of an horizontal edge there is, and how much of a vertical edge there is. The final value of the edge is computed as the length of a vector containing the two values.

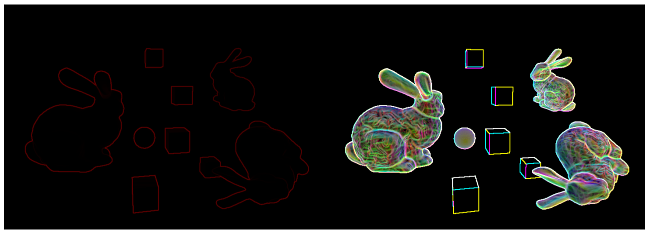


Figure 11. Results of the sobel filter on depth (left) and normals (rigth)

Afterwards we transform it to a boolean value : if the value goes above a certain threshold, there is an edge, else, there is none.

3.3.4 Adding border color and no border.

We may want to customize the color of the border, and if we want one or not, for each object on the scene.

The solution is to add another RGBA buffer, filled while rendering geometry by the wanted border color for the mesh.

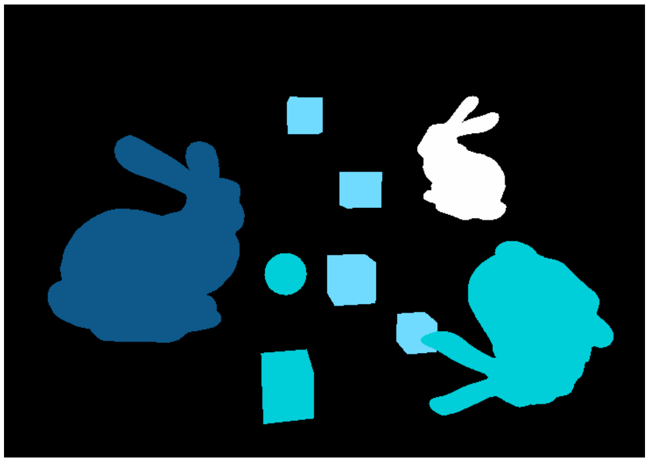


Figure 12. Contents of the border color buffer

Before applying the edge to the final image, we color it with the border color buffer. Any mesh who does not want borders makes it fully transparent, this way we can make borders appear and disappear at runtime.

Using the SRP to create this effect made possible a lot of optimisation, especially with the MRT part, that would

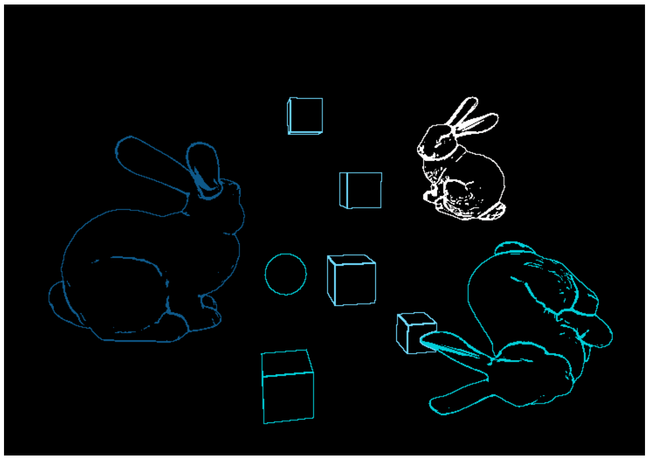


Figure 13. Sobel results combined with the border color buffer

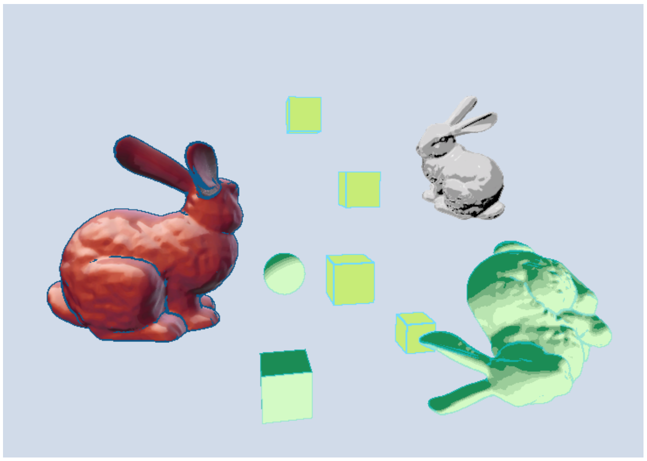


Figure 14. Final result

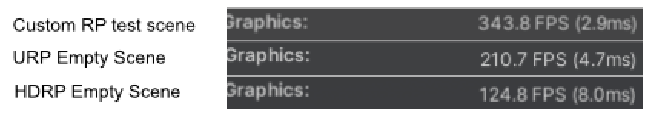
not be available using the URP, HDRP, or even the legacy built-in render pipeline. By writing custom shaders for the pipeline, I was able to merge a lot of GPU processing in the same rendering pass.

4 Conclusion

Creating a custom Scriptable Rendering Pipeline was a good experience, as it allowed me to get a much deeper understanding of how Unity handles rendering with the new render pipelines. It also made me discover new optimised techniques for rendering that I could leverage for future games.

In terms of performance, creating your own pipeline for your game may be really worth the time : Even in a scene with a few objects, my custom render pipeline (>300fps) performs way better than either URP(200fps) or HDRP(120fps) on an empty scene. (measures taken on the same computer)

For games with really specific needs in terms of rendering, that be performance, of specific art style, creating a custom



Custom RP test scene	Graphics:	343.8 FPS (2.9ms)
URP Empty Scene	Graphics:	210.7 FPS (4.7ms)
HDRP Empty Scene	Graphics:	124.8 FPS (8.0ms)

Figure 15. FPS Comparison in editor

SRP could be a great way to get creative and experiment with rendering techniques, while still having access to all other functionalities of the Unity engine.

References

[1] Simon M. Danner and Christoph J. Winklhofer. [n.d.]. Cartoon Style Rendering. Retrieved June 21, 2021 from https://www.cg.tuwien.ac.at/courses/Seminar/WS2007/comicstyle.pdf

[2] Jasper Flick. [n.d.]. *Unity Custom SRP Tutorials.* Retrieved June 21, 2021 from https://catlikecoding.com/unity/tutorials/custom-srp/

[3] Henning Steinbock. 2019. *Custom SRP and graphics workflows | Battle Planet - Judgement Day - Unite Copenhagen 2019.* Retrieved June 21, 2021 from https://www.youtube.com/watch?v=91zUwJwkXNQ

A Online Resources

Images from the games :

- Ape out : https://store.steampowered.com/app/447150/APE_OUT/
- The return of the Obra Din : https://store.steampowered.com/app/653530/Return_of_the_Obra_Dinn/
- The unfinished swan : https://store.steampowered.com/app/1206430/The_Unfinished_Swan/
- Mario 64 Wiki (https://sm64-conspiracies.fandom.com/wiki/Metal_Mario)

Unity and Unity SRP documentation :

- Unity : https://docs.unity3d.com/Manual/index.html
- SRP Core package : https://docs.unity3d.com/Manual/index.html

Github repo of the project : https://github.com/TheoLemoine/CartoonRenderPipeline