Cg/HLSL libretro shader tutorial

Hans-Kristian Arntzen, Daniel De Matteis

December 3, 2012

Contents

1 Introduction **1**

1.1 The rendering pipeline 2

1.2 A Cg/HLSL program 2

2 Hello World **3**

2.1 Cg types 4

2.1.1 Float4 4

2.1.2 Semantics 4

2.2 Putting it together 6

2.3 Result 7

3 Expanding further **7**

3.1 Lookup textures 7

3.2 Multipass 9

3.3 Game-aware shaders 9

3.3.1 RetroArch config file 12

3.3.2 How to test when developing for RetroArch? 12

3.3.3 Results 12

1 Introduction

This document is for a (fresh) shader developer that wants to develop shader programs for use in various emulators/games. Shader programs run on your GPU, and thus enables very sophisticated effects to be performed on the picture which might not be possible in real-time on the CPU. Some introduction to shader programming in general is given, so more experienced developers that only need reference for the specification may just skip ahead.

Current emulators that support the specification explained here to a certain degree are:

- RetroArch
- SNES9x Win32

There are three popular shader languages in use today:

- HLSL (High-Level Shading Language, Direct3D)
- GLSL (GL Shading Language, OpenGL)
- Cg (HLSL/GLSL, nVidia)

The spec is for the Cg shading language developed by nVidia. It "wraps" around OpenGL and HLSL to make shaders written in Cg quite portable. It is also the shading language implemented on the PlayStation3, thus increasing the popularity of it.

1.1 The rendering pipeline

With shaders you are able to take control over a large chunk of the GPUs inner workings by writing your own programs that are uploaded and run on the GPU. In the old days, GPUs were a big black box that was highly configurable using endless amount of API calls. In more modern times, rather than giving you endless amounts of buttons, you are expected to implement the few «buttons» you actually need, and have a streamlined API.

The rendering pipeline is somewhat complex, but we can in general simplify it to:

- Vertex processing
- Rasterization
- Fragment processing
- Framebuffer blend

We are allowed to take control of what happens during vertex processing, and fragment processing.

1.2 A Cg/HLSL program

If you were to process an image on a CPU, you would most likely do something like this:

```
for (unsigned y = 0; y < height; y++) {
    for (unsigned x = 0; x < width; x++)
        out_pixel[y][x] = process_pixel(in_pixel[y][x], y, x);
}
```

We quickly realize that this is highly serial and slow. We see that out_pixel[y][x] isn't dependent on out_pixel[y + k][x + k], so we see that we can parallelize quite a bit.

Essentially, we only need to implement process_pixel() as a single function, which is called thousands, even millions of time every frame. The only purpose in life for process_pixel() is to process an input, and produce an output. No state is needed, thus, a “pure” function in computer science terms.

For the Cg program, we need to implement two different functions.

main_vertex() takes care of transforming every incoming vertex from camera space down to clip space. This essentially means projection of 3D (coordinates on GPU) down to 2D (your screen)[1].

Vertex shaders get various coordinates as input, and uniforms. Every vertex emitted by the emulator is run through main_vertex which calculates the final output position[2].

While coordinates differ for each invocation, uniforms are constant throughout every call. Think of it as a global variable that you're not allowed to change.

Vertex shading can almost be ignored altogether, but since the vertex shader is run only 4 times, and the fragment shader is run millions of times per frame, it is a good idea to precalculate values in vertex shader that can later be used in fragment shader. There are some limitiations to this which will be mentioned later.

main_fragment() takes care of calculating a pixel color for every single output pixel on the screen. If you're playing at 1080p, the fragment shader will have to be run 1920 * 1080 times! This is obviously straining on the GPU unless the shader is written efficiently.

Obviously, main_fragment is where the real action happens. For many shaders we can stick with a “dummy” vertex shader which does some very simple stuff.

The fragment shader receives a handle to a texture (the game frame itself), and the texture coordinate for the current pixel, and a bunch of uniforms.

A fragment shader's final output is a color, simple as that. Processing ends here.

2 Hello World

We'll start off with the basic vertex shader. No fancy things are being done. You'll see a similiar vertex shader in most of the Cg programs out there in the wild.

```
void main_vertex(
    float4 pos : POSITION,
    out float4 out_pos : POSITION,
    uniform float4x4 modelViewProj,
    float4 color : COLOR,
    out float4 out_color : COLOR,
    float2 tex : TEXCOORD,
    out float2 out_tex : TEXCOORD
```

[1]Since we're dealing with old school emulators here, which are already 2D, the vertex shading is very trivial.

[2]For our emulators this is just 4 times, since we're rendering a quad on the screen. 3D games would obviously have a lot more vertices.

```
)
{
   out_pos = mul(modelViewProj, pos);
   out_color = color;
   out_tex = tex;
}
```

This looks vaguely familiar to C, and it is. Cg stands for "C for graphics" after all. We notice some things are happening, notable some new types.

2.1 Cg types

2.1.1 Float4

float4 is a vector type. It contains 4 elements. It could be colors, positions, whatever. It's used for vector processing which the GPUs are extremely efficient at.

2.1.2 Semantics

We see various semantics. The POSITION semantic means that the variable is tied to vertex coordinates. We see that we have an input POSITION, and an output (out) POSITION. We thus transform the input to the output with a matrix multiply with the current model-view projection. Since this matrix is the same for every vertex, it is a uniform. Remember that the variable names DO matter. modelViewProj has to be called exactly that, as the emulator will pass the MVP to this uniform. It is in the specification.

Since we have semantics for the POSITION, etc, we can call them whatever we want, as the Cg environment figures out what the variables mean.

The transformation happens here:

```
out_pos = mul(modelViewProj, pos);
```

The COLOR semantic isn't very interesting for us, but the example code in nVidias Cg documentation includes it, so we just follow along.

TEXCOORD is the texture coordinate we get from the emulator, and generally we just pass it to the fragment shader directly. The coordinate will then be "linearly interpolated" across the fragments. More complex shaders can output (almost) as many variables they want, that will be linearily interpolated for free to the fragment shader.

We also need a fragment shader to go along with the vertex shader, and here's a basic shader that only outputs the pixel as-is. This is pretty much the result you'd get if you didn't run any shader (fixed-function) at all.

```
float4 main_fragment(uniform sampler2D s0 : TEXUNIT0,
   float2 tex : TEXCOORD) : COLOR
{
   return tex2D(s0, tex);
}
```

This is arguably simpler than the vertex shader. Important to notice are:

sampler2D is a handle to a texture in Cg. The semantic here is TEXUNIT0, which means that it refers to the texture in texture unit 0. This is also part of the specification.

float2 tex : TEXCOORD is the interpolated coordinate we received from the vertex shader.

tex2D(s0, tex); simply does texture lookup and returns a COLOR, which is emitted to the framebuffer. Simple enough. Practically every fragment does more than one texture lookup. For example, classic pixel shaders look at the neighbor pixels as well to determine the output. But where is the neighbor pixel? We'll revise the fragment shader and try to make a really blurry shader to demonstrate. We now need to pull up some uniforms. We need to know how to modify our tex coordinates so that it points to a neighbor pixel.

```
struct input
{
   float2 video_size;
   float2 texture_size;
   float2 output_size;
   float frame_count;
};

float4 main_fragment(uniform sampler2D s0 : TEXUNIT0,
   uniform input IN, float2 tex : TEXCOORD) : COLOR
{
   float4 result = float4(0.0);
   float dx = 1.0 / IN.texture_size.x;
   float dy = 1.0 / IN.texture_size.y;

   // Grab some of the neighboring pixels and
   // blend together for a very mushy blur.
   result += tex2D(s0, tex + float2(-dx, -dy));
   result += tex2D(s0, tex + float2(dx, -dy));
   result += tex2D(s0, tex + float2(0.0, 0.0));
   result += tex2D(s0, tex + float2(-dx, 0.0));
   return result / 4.0;
}
```

Here we use IN.texture_size to determine the the size of the texture. Since GL maps the whole texture to the interval [0.0, 1.0], 1.0 / IN.texture_size means we get the offset for a single pixel, simple enough. Almost every shader uses this. We can calculate these offsets in vertex shader to improve performance since the coordinates are linearily interpolated anyways, but that is for another time ... ;)

2.2 Putting it together

The final runnable product is a single .cg file with the main_vertex and main_fragment functions added together. Not very complicated. For the icing on the cake, you should add a license header.

```
/* Stupid blur shader.
   Author: Your friendly neighbor.
   License: We don't have those things!
*/

struct input
{
   float2 video_size;
   float2 texture_size;
   float2 output_size;
   float frame_count;
};

void main_vertex(
   float4 pos : POSITION,
   out float4 out_pos : POSITION,
   uniform float4x4 modelViewProj,
   float4 color : COLOR,
   out float4 out_color : COLOR,
   float2 tex : TEXCOORD,
   out float2 out_tex : TEXCOORD
)
{
   out_pos = mul(modelViewProj, pos);
   out_color = color; out_tex = tex;
}

float4 main_fragment(uniform sampler2D s0 : TEXUNIT0,
   uniform input IN, float2 tex : TEXCOORD) : COLOR
{
   float4 result = float4(0.0);
   float dx = 1.0 / IN.texture_size.x;
   float dy = 1.0 / IN.texture_size.y;

   // Grab some of the neighboring pixels and blend
   // together for a very mushy blur.
   result += tex2D(s0, tex + float2(-dx, -dy));
   result += tex2D(s0, tex + float2(dx, -dy));
   result += tex2D(s0, tex + float2(0.0, 0.0));
   result += tex2D(s0, tex + float2(-dx, 0.0));
   return result / 4.0;
```

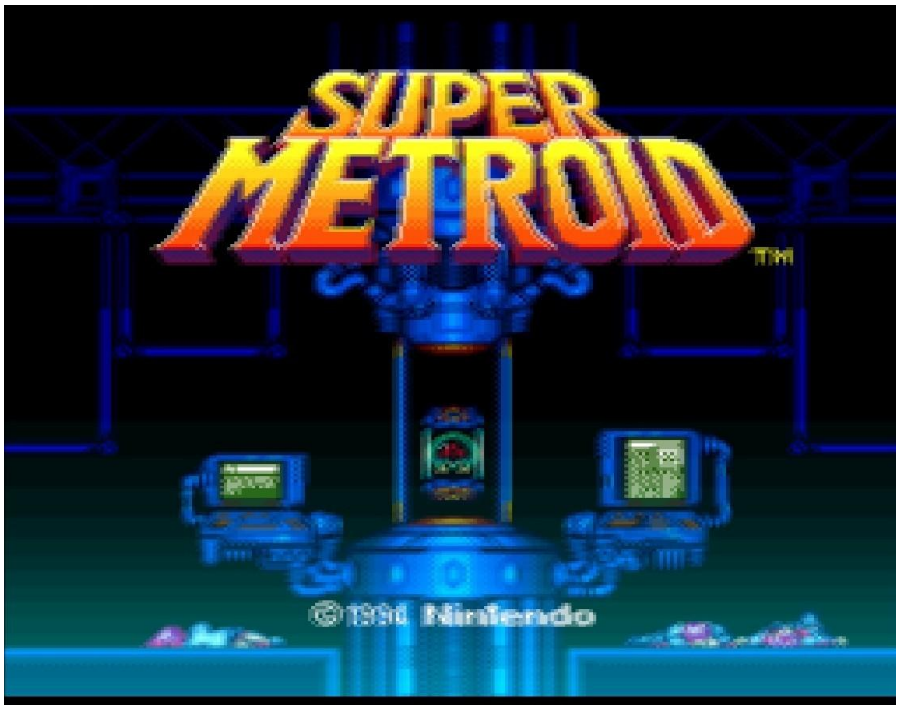

Figure 1: The result of the shader code.

```
}
```

2.3 Result

As you can see, it's not a practical shader, but it shows the blurring effect to the extreme.

3 Expanding further

3.1 Lookup textures

We'll first mention a very popular feature among RetroArch users the ability to access external textures. This means we have several samplers available for use. In the config file, we define the textures as so:

```
textures = "foo;bar"
foo = path_foo.png
bar = bar_foo.png
foo_linear = true # Linear filtering for foo.
bar_linear = true # Linear filtering for bar.
```

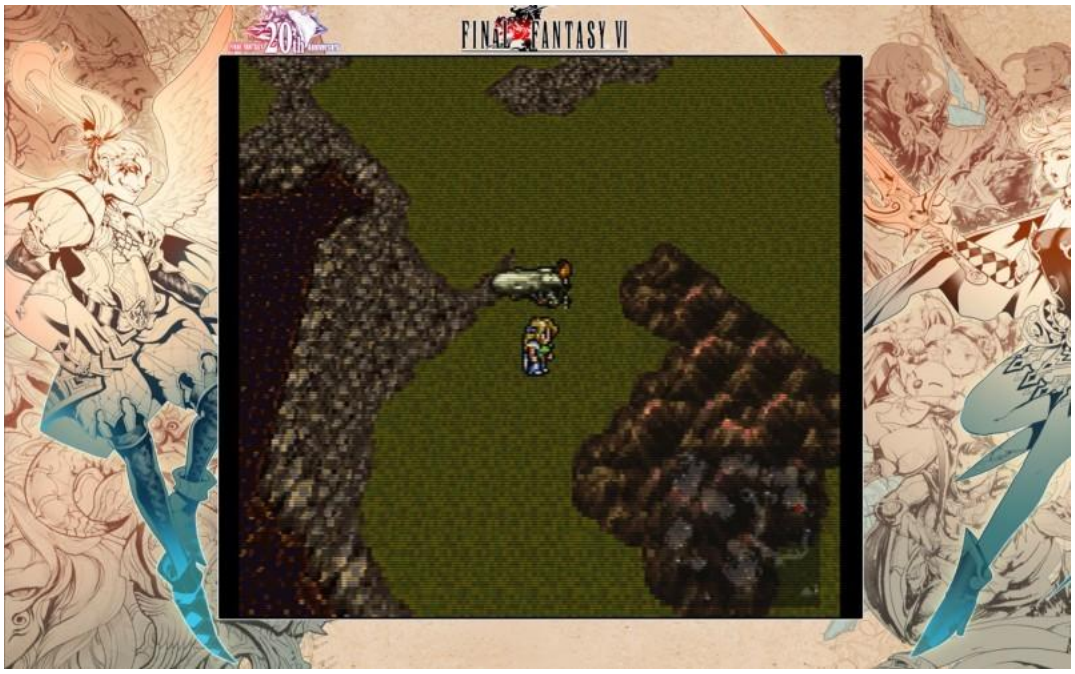

Figure 2: A shader making use of a lookup texture for the purpose of drawing a background border.

RetroArch PS3 uses PNG as the main format, RetroArch can use whatever if Imlib2 support is compiled in. If not, it's restricted to lop-left ordered, non-RLE TGA.

From here on, "foo" and "bar" can be found as uniforms in the shaders. The texture coordinates for the lookup texture will be found in TEXCOORD1. This can simply be passed along with TEXCOORD0 in the vertex shader as we did with TEXCOORD0. Here we make a fragment shader that blends in two background picture at a reduced opacity. Do NOT assign lookup textures to a certain TEXUNIT, Cg will assign a fitting texture unit to the sampler.

```
float4 main_fragment(uniform sampler2D s0 : TEXUNIT0,
   uniform sampler2D foo, uniform sampler2D bar,
   float2 tex : TEXCOORD0, float2 tex_lut : TEXCOORD1) : COLOR
{
   float4 bg_sum = (tex2D(foo, tex_lut) + tex2D(bar, tex_lut)) * 0.15;
   return lerp(tex2D(s0, tex), bg_sum, bg_sum.a); // Alpha blending.
}
```

Here's an example of what can be achieved using borders (which are just a simple lookup texture):

3.2 Multipass

It is sometimes feasible to process an effect in several steps.

```
shaders = 2
shader0 = pass1.cg
shader1 = pass2.cg
scale_type0 = source
scale0 = 2.0
filter_linear0 = true
filter_linear1 = false
```

3.3 Game-aware shaders

This is a new and exciting feature. It allows shaders to grab data from the emulator state itself, such as RAM data. This is only implemented for SNES so far, but the idea is quite extendable and portable.

The basic idea is that we capture RAM data in a certain way (semantic if you will) from the SNES, and pass it as a uniform to the shader. The shader can thus act on game state in interesting ways.

As a tool to show this feature, we'll focus on replicating the simple tech demo shown on YouTube: http://www.youtube.com/watch?v=4VzaE9q735k

What happens is that when Mario jumps in the water, the screen gets a "watery" effect applied to it, with a rain lookup texture, and a wavy effect. When he jumps out of the water, the water effect slowly fades away.

We thus need to know two things:

- Is Mario currently in water or not?
- If not, how long time was it since he jumped out?

Since shaders do not have state associated with it, we have to let the environment provide the state we need in a certain way. We'll call this concept a semantic.

To capture a RAM value directly, we can use the "capture" semantic. To record the time when the RAM value last changed, we can use the "transition" semantic. We obviously also need to know where in RAM we can find this information. Luckily, the guys over at SMW Central know the answer: http://www.smwcentral.net/?p=map&type=ram

We see:

```
$7E:0075, byte, Flag, Player is in water flag. #$00 = No; #$01 = Yes.
```

Bank $7E and $7F are mapped to WRAM $0000-$FFFF and $10000-$1FFFF respectively. Thus, our WRAM address is $0075.

In the config file, we can now set up the uniforms we'll want to be captured in the config file.

```
imports = "mario_water;mario_water_time"
mario_water_semantic = capture
```

```
# Capture the RAM value as-is.
mario_water_wram = 0075
# This value is hex!
mario_water_time_semantic = transition
# Capture the frame count when this variable last changed.
# Use with IN.frame_count, to create a fade-out effect.
mario_water_time_wram = 0075
```

The amount of possible “semantics” are practically endless. It might be worthwhile to attempt some possibility to run custom code that keeps track of the shader uniforms in more sophisticated ways later on. Do note that there is also a %s_mask value which will let you bitmask the RAM value to check for bit-flags more easily.

Now that we got that part down, let’s work on the shader design. In the fragment shader we simply render both the full water effect, and the «normal» texture, and let a “blend” variable decide. We can say that 1.0 is full water effect, 0.0 is no effect. We can start working on our vertex shader. We will do something useful here for once.

```
struct input
{
   float frame_count;
};

void main_vertex(
    float4 pos : POSITION,
    out float4 out_pos : POSITION,
    uniform float4x4 modelViewProj,
    float4 color : COLOR,
    out float4 out_color : COLOR,
   float2 tex : TEXCOORD0,
   out float2 out_tex : TEXCOORD0,
   float2 tex1 : TEXCOORD1,
   out float2 out_tex1 : TEXCOORD1,

   // Even if the data should have been int,
   // Cg doesn't seem to
   uniform float mario_water,
   // support integer uniforms
   uniform float mario_water_time,
   uniform input IN,
   // Blend factor is passed to fragment shader.
   // We'll output the same value in every vertex,
   // so every fragment will get the same value
   // for blend_factor since there is nothing to interpolate.
   out float blend_factor )
{
   out_pos = mul(modelViewProj, pos);
   out_color = color;
   out_tex = tex;
   out_tex1 = tex1;
   float transition_time = 0.5 *
   (IN.frame_count  mario_water_time) / 60.0;

   // If Mario is in the water ($0075 != 0),
   // it's always 1 ...
   if (mario_water > 0.0)
      blend_factor = 1.0;
   // Fade out from 1.0 towards 0.0 as
   // transition_time grows larger.
   else
      blend_factor = exp(-transition_time);
}
```

All fine and dandy so far, now we just need to use this blend_factor in our fragment shader somehow ... Let's move on to the fragment shader where we blend.

```
float apply_wave(float2 pos, float2 src, float cnt)
{
   float2 diff = pos - src;
   float dist = 300.0 * sqrt(dot(diff, diff));
   dist -= 0.15 * cnt;
   return sin(dist);
}

// Fancy shizz to create a wave.
float4 water_texture(float4 output, float2 scale, float cnt)
{
   float res = apply_wave(scale, src0, cnt);
   res += apply_wave(scale, src1, cnt);
   res += apply_wave(scale, src2, cnt);
   res += apply_wave(scale, src3, cnt);
   res += apply_wave(scale, src4, cnt);
   res += apply_wave(scale, src5, cnt);
   res += apply_wave(scale, src6, cnt);
   return output * (0.95 + 0.012 * res);
}

float4 main_fragment
(
   uniform input IN,
  float2 tex : TEXCOORD0, uniform sampler2D s0 : TEXUNIT0,
   uniform sampler2D rain, float2 tex1 : TEXCOORD1,
   in float blend_factor // Passed from vertex
) : COLOR
{
   float4 water_tex = water_texture(tex2D(s0, tex), tex1, IN.frame_count);
   float4 normal_tex = tex2D(s0, tex);
   float4 rain_tex = tex2D(rain, tex1);

   // First, blend normal and water texture together,
   // then add the rain texture with alpha blending on top
   return lerp(lerp(normal_tex, water_tex, blend_factor),
   rain_tex, rain_tex.a * blend_factor * 0.5);
}
```

3.3.1 RetroArch config file

```
shaders = 1
shader0 = mario.cg
filter_linear0 = true
imports = "mario_water;mario_water_time"
mario_water_semantic = capture
mario_water_time_semantic = transition
mario_water_wram = 0075
mario_water_time_wram = 0075
textures = rain
rain = rain.tga
rain_linear = true
```

3.3.2 How to test when developing for RetroArch?

To develop these kinds of shaders, I'd recommend using RetroArch w/ Cg support, and a debugging tool for your emulator of choice to peek at RAM values (build it for bSNES yourself with options=debugger).

After written, the shader should translate nicely over to RetroArch with some slight changes to the config.

3.3.3 Results

Here are some screenshots of the mario effect (in Super Mario World SNES) we developed. Obviously this is a very simple example showing what can be done. It's not confined to overlays. The imagination is the limit here.

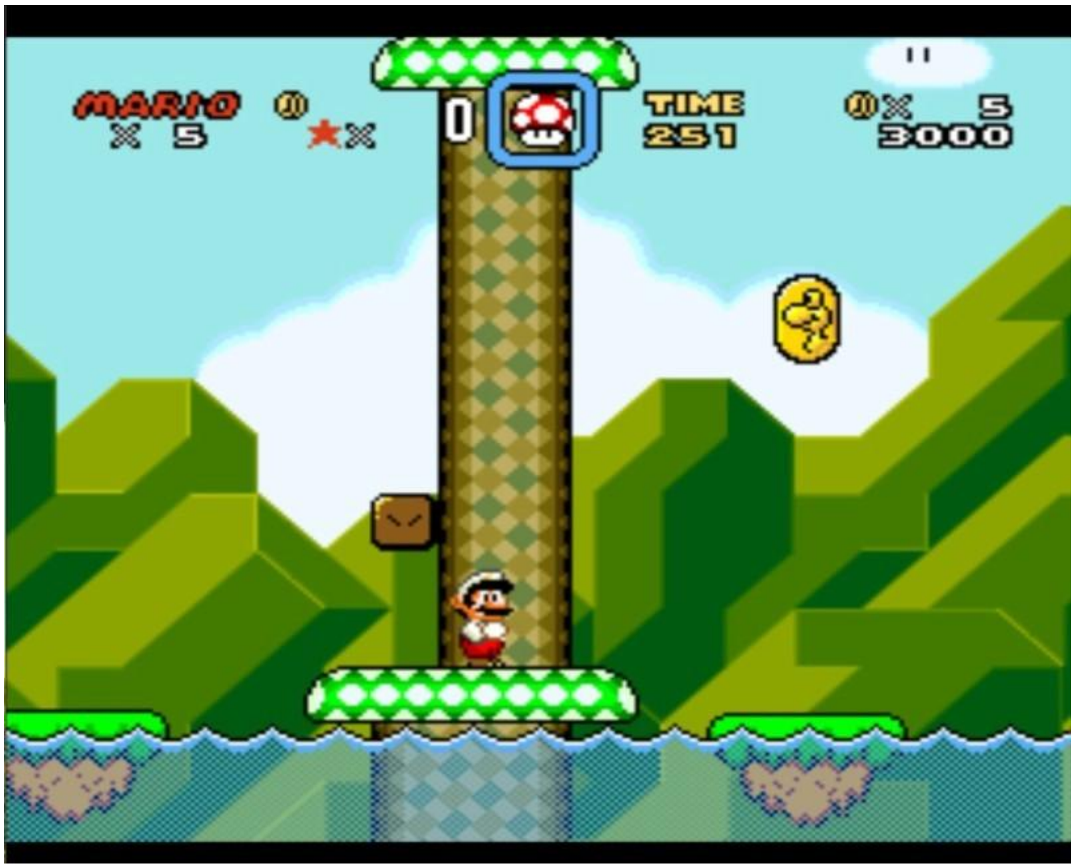

Figure 3: Super Mario World prior to Mario jumping in water.

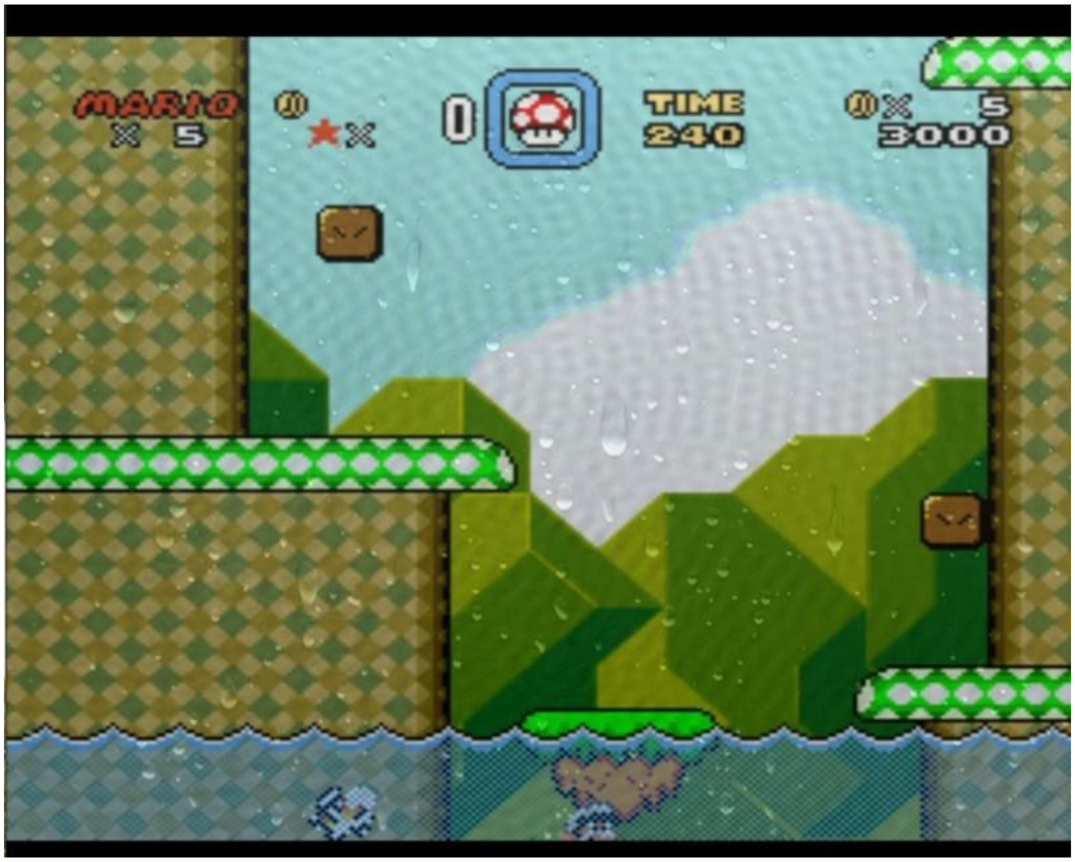

Figure 4: Super Mario World with a game aware shader applying a LUT texture as soon as Mario jumps into the water.

Index

B
bSNES, 12

C
Cg, 2, 4, 12

G
GLSL, 2

H
HLSL, 2

O
OpenGL, 2

P
PlayStation3, 2

R
RetroArch, 1, 12

S
SNES9x, 1