HDR RENDERING

High Dynamic Range Rendering of a 3D scene made with WebGL and Three.js

Filippo Bianco

September 2018

HDR RENDERING

- The task of the project is to render a computer graphic 3D scene by using lighting calculations executed in a higher range, called *High Dynamic Range*.

- These calculations allow to preserve details that may be lost due to limiting contrast ratios.

WHY HDR ?

- ***Statement***: Brightness and color values by default are clamped between 0.0 and 1.0 when stored into a framebuffer.
- ***Situation***: Bright area with multiple bright light sources that as a total sum exceed 1.0.
- ***Problem***: A large number of fragments' color values getting clamped to 1.0 each of the bright fragments have the exact same white color in a large region, losing a significant amount of detail and giving it a fake look.
- ***Weak solution***: Reduce the strength of the light sources and ensure no area of fragments in your scene ends up brighter than 1.0.
- ***Strong solution***: Allow color values to temporarily exceed 1.0 and transform them back to the original range of 0.0 and 1.0 (*Tone Mapping*) as a final step, but without losing detail. By allowing fragment colors to exceed 1.0 we have a much higher range of color values available to work in known as high dynamic range (HDR).

PROJECT’S STEPS

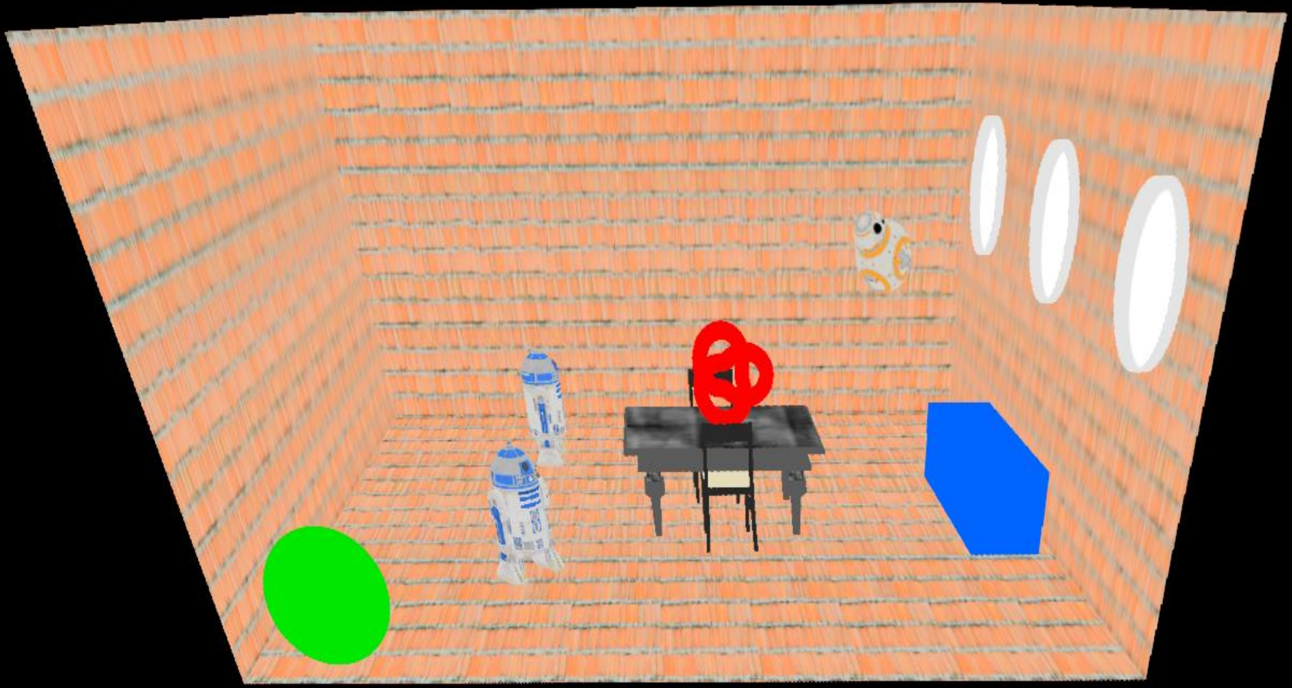
1. Build the scene with strong light sources
2. Allow color values to exceed the range
3. Perform the *tone mapping*
4. Render the new scene

BUILD THE SCENE

- The 3D scene, on which HDR Rendering is applied, has been created from scratch with the help of *Three.js* and *Clara.io*.
- *Three.js* is a cross-browser Javascript library and Application Programming Interface (*API*) used to create and display animated 3D computer graphics in a web browser.
- *Clara.io* is a web-based freemium 3D computer graphics software developed by Exocortex, useful in my project because of its 3D model library.
- Web browser used: *Mozilla Firefox*.

BUILD THE SCENE

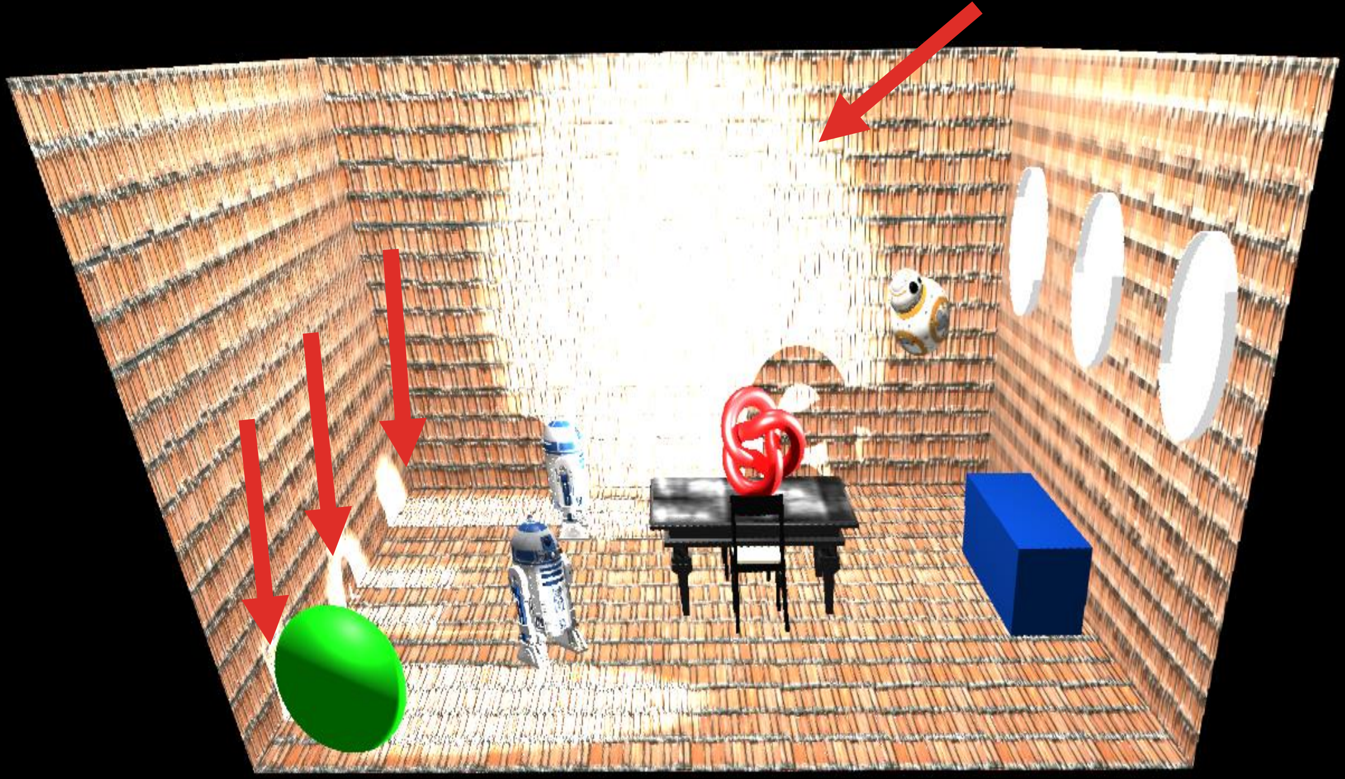
- Scene built in THREE.BoxBufferGeometry whose internal walls are covered with loaded textures.
- Geometric shapes belong to *Three.js* library.
- The shapes of windows and other objects belong to *Clara.io* library.
- The only ambient illuminates the scene.



3D scene with simple ambient light

LIGHT SOURCES

- Adding some light sources the scene become more dynamic and realistic.
- Shadows and reflections have been added consequently.
- Despite the positive aspects we can already see lighting problems.

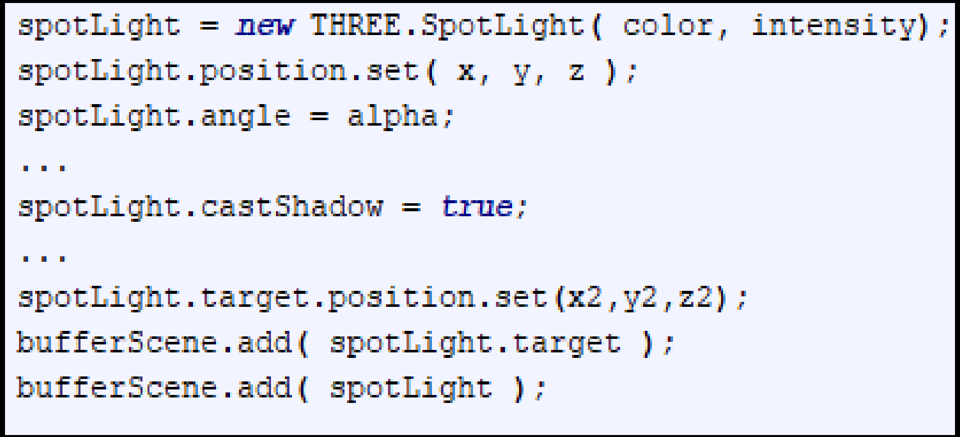


3D scene with 4 light sources:

- ***3 coming from the windows on the right wall***
- ***1 coming from the observer's view***

LIGHT SOURCES

- These light sources are represented thanks to *Three.js* with the *SpotLight* object.
- This light gets emitted from a single point in one direction, along a cone that increases in size the further from the light it gets.



```
spotLight = new THREE.SpotLight( color, intensity);
spotLight.position.set( x, y, z );
spotLight.angle = alpha;
...
spotLight.castShadow = true;
...
spotLight.target.position.set(x2,y2,z2);
bufferScene.add( spotLight.target );
bufferScene.add( spotLight );
```

- There are many parameters to help you choose the right light for the scene.

HIGHER RANGE AND FBO

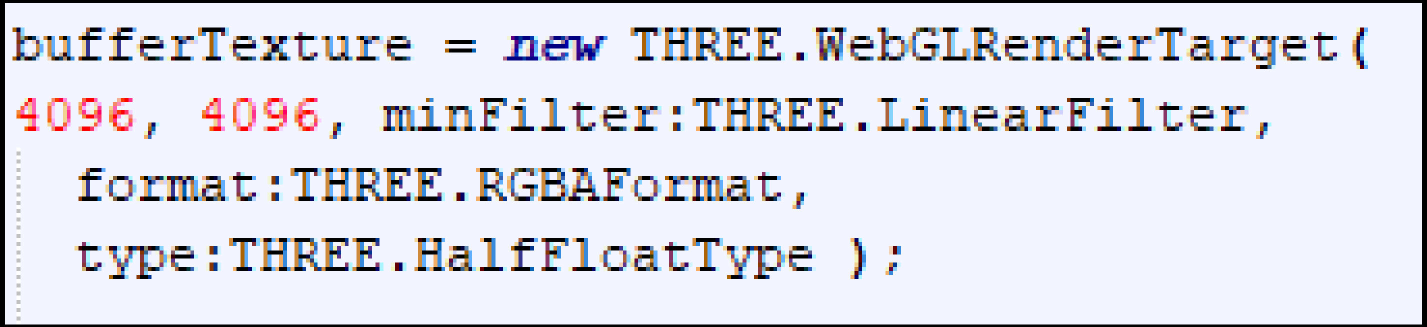
- As we saw, to implement high dynamic range rendering we need some way to prevent color values getting clamped after each fragment shader run.
- ***Floating point framebuffer***, thanks to the internal format of the attached colorbuffer, can store floating point values outside the default range.
- To replicate a WebGL setup of a *Framebuffer Object* containing a single rendered texture, the steps are:

1. Create a scene to hold your objects (***already done***).
2. Create a texture to store what you render.
3. Render your scene onto your texture.

2. Allow color values to exceed the range

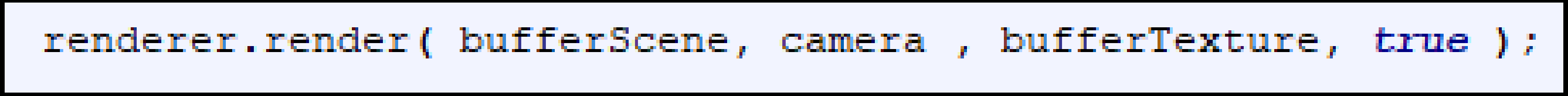
2. CREATE A TEXTURE TO STORE WHAT YOU RENDER

- ***WebGLRenderTarget*** is a buffer where the video card draws pixels for a scene that is being rendered in the background.
- This is what *Three.js* uses to let us render onto something other than the screen.
- The choice of the type allows exceeding color values.



```
bufferTexture = new THREE.WebGLRenderTarget(
4096, 4096, minFilter:THREE.LinearFilter,
  format:THREE.RGBAFormat,
  type:THREE.HalfFloatType );
```

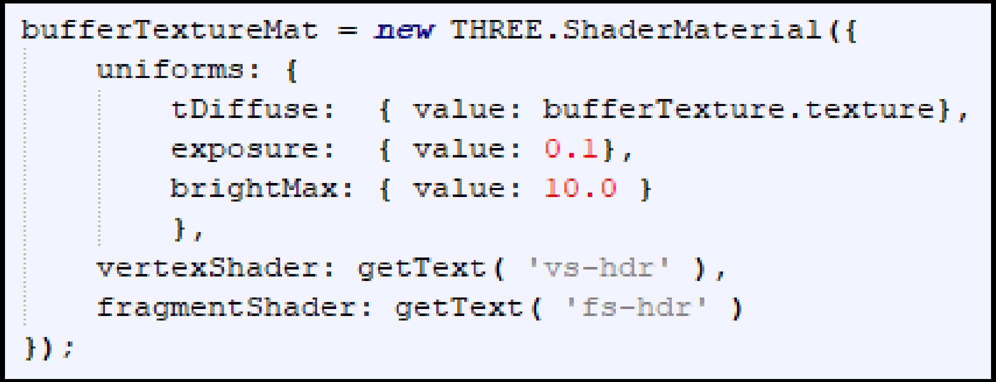
3. RENDER YOUR SCENE ONTO YOUR TEXTURE



```
renderer.render( bufferScene, camera , bufferTexture, true );
```

SHADERS

- On the main scene we can use the *bufferTexture* (WebGLRenderTarget Object) to create a material rendered with custom shaders and applied on a simple plane.
- *Tone mapping* is, obviously, implemented in the fragment shader (*fs-hdr*).



```
bufferTextureMat = new THREE.ShaderMaterial({
    uniforms: {
        tDiffuse:  { value: bufferTexture.texture},
        exposure:  { value: 0.1},
        brightMax: { value: 10.0 }
        },
    vertexShader: getText( 'vs-hdr' ),
    fragmentShader: getText( 'fs-hdr' )
});
```

VERTEX SHADER

- The custom shaders are developed using the GLSL language, to model the colors of the scene.

- The vertex shader (*vs-hdr*) simply computes the *gl_Position* vector using the *projectionMatrix,* the *modelViewMatrix* and the *position*.

FRAGMENT SHADER

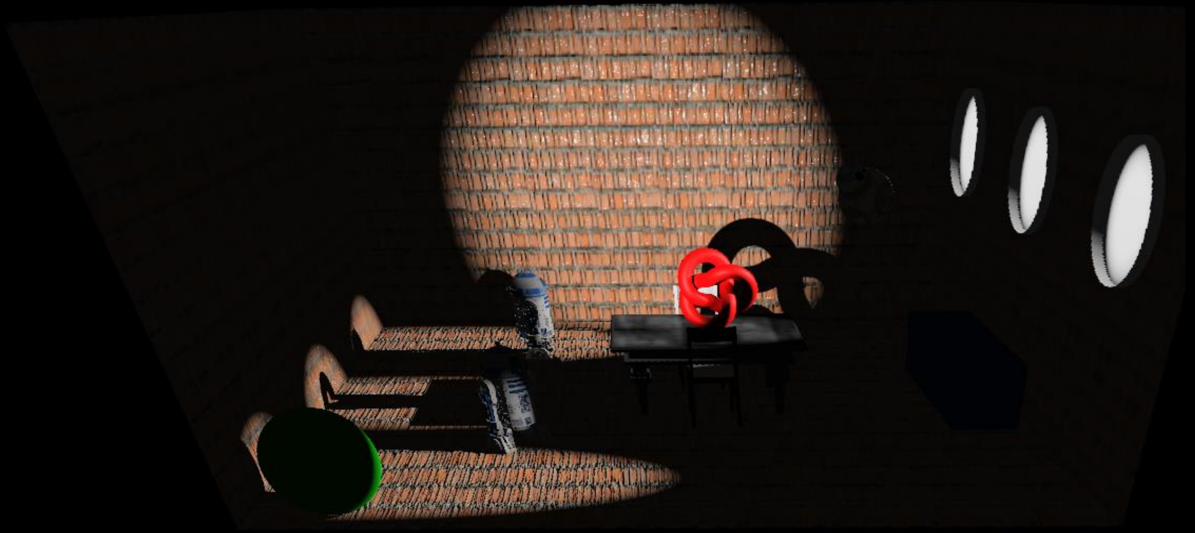
- Before performing the tone mapping operation, the function *texture2D()* returns a texel : the (color) value of the texture for the given coordinates. The function has one input parameter of the type sampler2D and one input parameter of the type vec2 : *tDiffuse*, the uniform the texture is bound to, and *vUv*, the 2-dimensional coordinates of the texel to look up.

```
vec4 pixel = texture2D( tDiffuse, vUv );
```

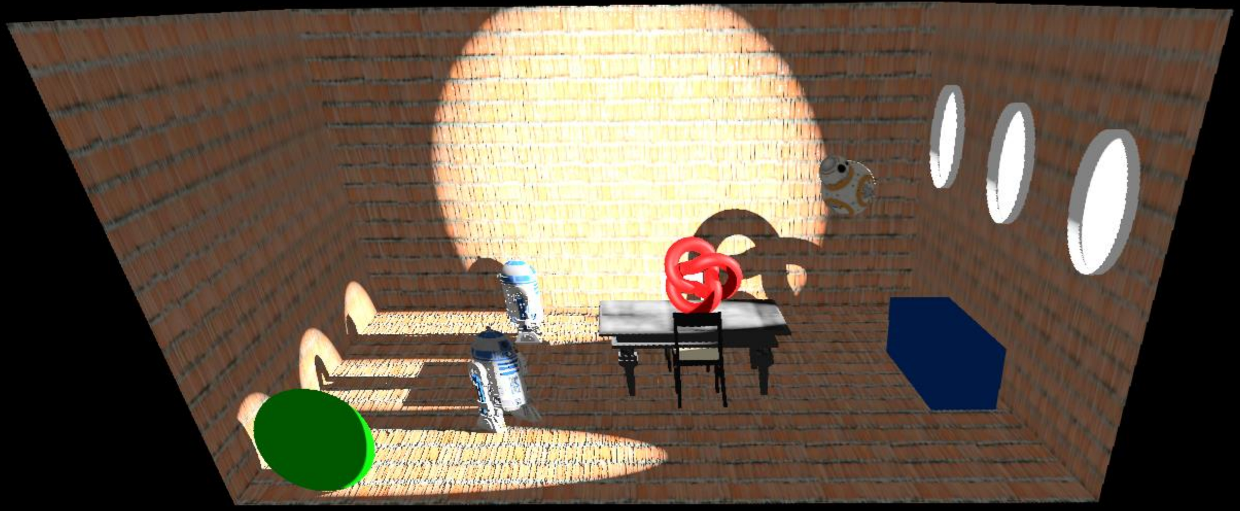
- The resulting texel *pixel* is multiplied by an exposure value and so,by changing this parameter, we get to see a lot of details of our scene.

FRAGMENT SHADER

- In this example, with high exposure values the darker areas show significantly more details. In contrast, a low exposure largely removes the dark region details, but allows us to see more detail in the bright areas of a scene.



Low exposure



High exposure

FRAGMENT SHADER AND TMO

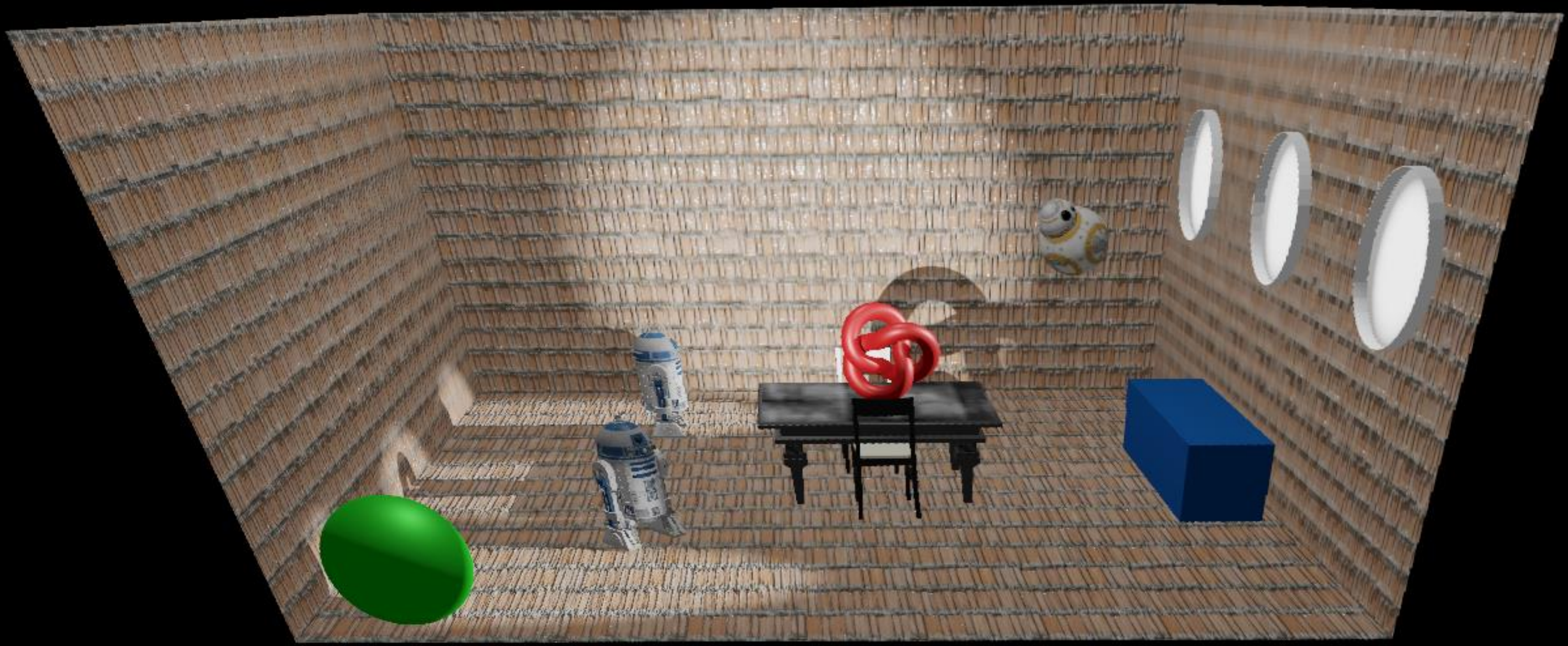
- At this point we can perform the tone mapping to transform the color values back to the original range.

- There are many tone mapping operators (TMO) but we have mainly tested three:

  1. Simple TM
  2. Simple TM with Luminance
  3. Reinhard TM

SIMPLE TONE MAPPING

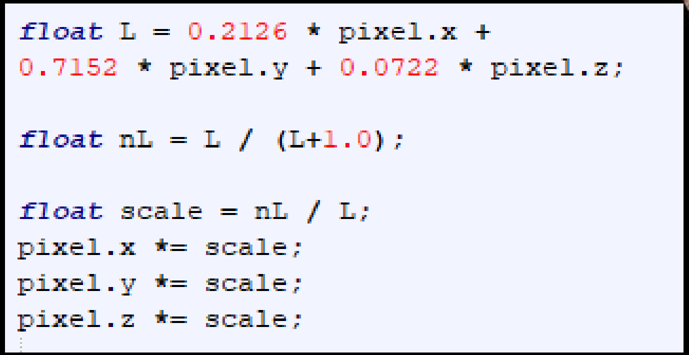
```
pixel.xyz = pixel.xyz/(pixel.xyz + vec3(1.0));
```



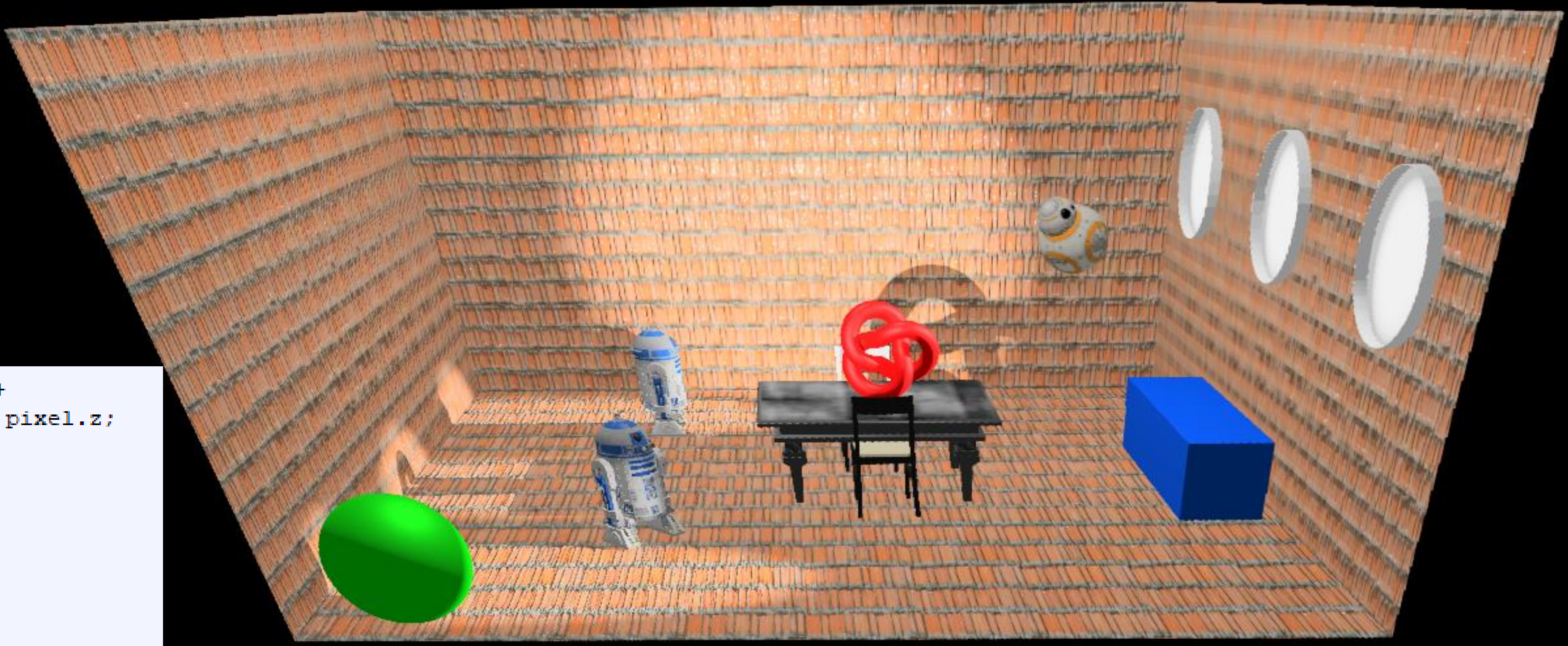
SIMPLE TONE MAPPING

- The problem with this tone mapping operator is that it operates on the RGB channels independently.
- Applying any non-linear transform in this way will result in both hue and saturation shifts, which is something that should be performed during final colour grading, not tone mapping.
- Next TMOs are applied on each pixel's *luminance*, which will preserve both the hue and saturation of the original image.

TONE MAPPING WITH LUMINANCE



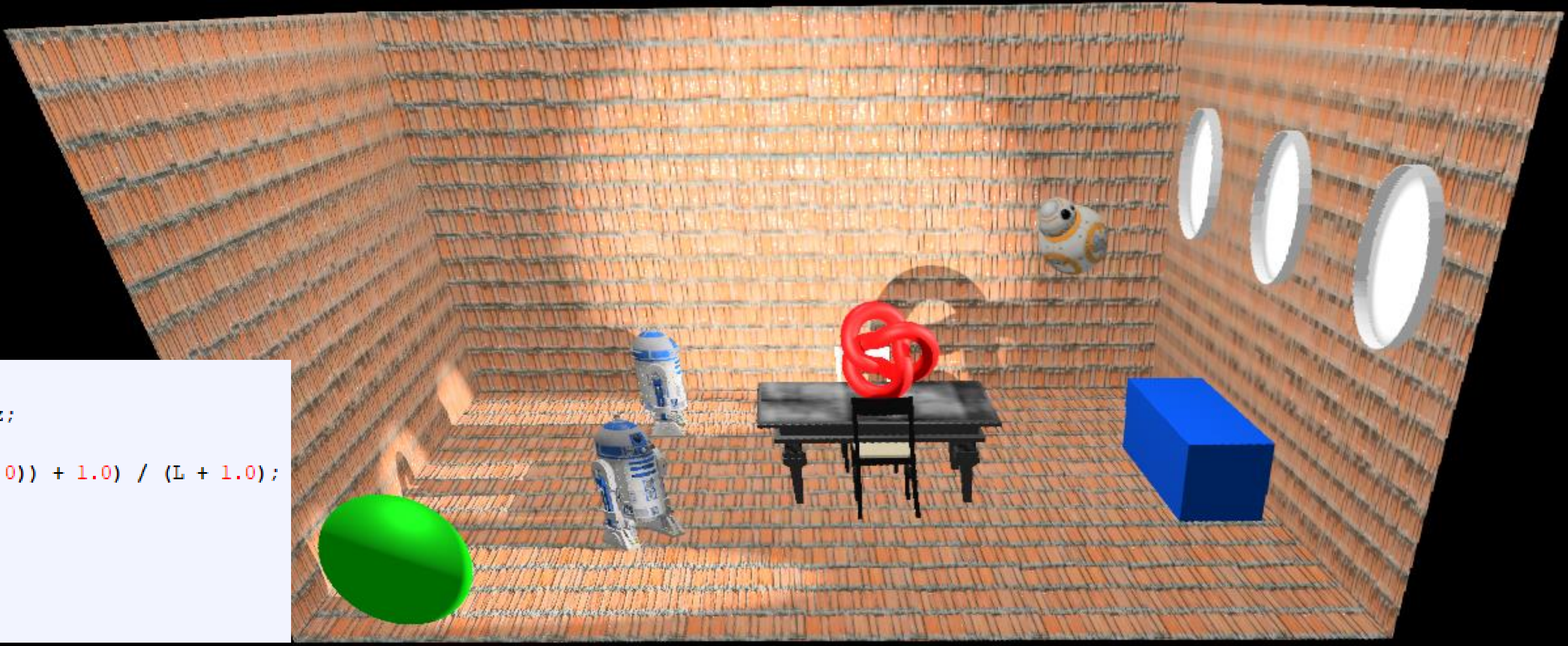
```
float L = 0.2126 * pixel.x +
0.7152 * pixel.y + 0.0722 * pixel.z;

float nL = L / (L+1.0);

float scale = nL / L;
pixel.x *= scale;
pixel.y *= scale;
pixel.z *= scale;
```

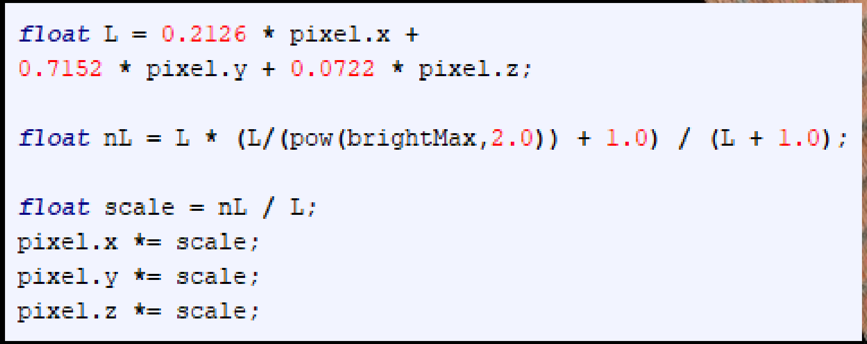


TONE MAPPING WITH LUMINANCE

- This approach is to calculate the luminance directly from the RGB values, perform the tone mapping on this value and then scale the original RGB values appropriately. This formulation is guaranteed to bring all luminances with a displayable range.
- The high luminances are scaled by approximately 1/L, while low luminances are scaled by 1. The denominator causes a graceful blend between these two scalings.
- Now all the whites are gone. The reason is that by preserving the hue and saturation, the operator prevents any colours being blown out to a full white.

REINHARD TONE MAPPING





```
float L = 0.2126 * pixel.x +
0.7152 * pixel.y + 0.0722 * pixel.z;

float nL = L * (L/(pow(brightMax,2.0)) + 1.0) / (L + 1.0);

float scale = nL / L;
pixel.x *= scale;
pixel.y *= scale;
pixel.z *= scale;
```

REINHARD TONE MAPPING

- Tone mapping extended to allow high luminances to burn out in a controllable fashion.
- For many high dynamic range images, the compression provided by this technique appears to be sufficient to preserve detail in low contrast areas, while compressing high luminances to a displayable range.

CONCLUSIONS

- In conclusion we have seen how to perform HDRR on a WebGL scene mainy created with *Three.js* and how the choice of different Tone Mapping operators implementation can modify the scene.
- Actually we have shown simple TMOs but the documentation on more efficient and complex TMOs is wide and strongly linked to the involved subject.
- The results have been useful and meaningful to deepen the project's topic.