Extra Features Report

Group 153
Radu Zidaru, Bo Kraijnbrink, Vasil Georgiev

Contents

1 Environment Maps **2**
1.1 Description . . . 2
1.2 Visual Debug . . . 2
1.3 Rendered Images . . . 3
1.4 Location . . . 3

2 Glossy Reflections **3**
2.1 Description . . . 3
2.2 Rendered Images . . . 4
2.3 Location . . . 4

3 Bloom Filter post-processing **5**
3.1 Description . . . 5
3.2 Visual Debug . . . 5
3.3 Rendered Images . . . 5
3.4 Location . . . 6

4 Sources **6**

1 Environment Maps

1.1 Description

First of all, we add the **envMapCube** and **envMap** variables inside of the Scene struct, representing the Mesh of the cube map and the texture we want to load to a specific scene, respectively. We do this inside the Custom scene in **scene.cpp**: we load the **"cube-textured.obj"** mesh given in the data directory, as well as the **"emoji.png"** texture using the **loadEnv** method that we created inside of **scene.cpp**, although any other texture can be loaded by just passing a different path as a parameter to this method.

Inside the Mesh struct, we also added a scale method that takes a float **scaleFactor** as parameter and it is meant to be used to resize the entire cube map. We also call this function using different values for **scaleFactor** inside **scene.cpp** (**30.0f** is default). In order to change the size of the cube map, the **scaleFactor** variable has to be changed manually, since we didn't have time to implement a dynamic slider.

Next, inside of **extra.cpp**, we created a method **getTextureColor** that given a State and a Ray, computes the closest texel to the intersection point of the ray with the cube map and returns the corresponding color from the loaded texture inside the Scene. Inside the **sampleEnvironmentMap** method, we first check if the feature flag is enabled and if there is any texture loaded inside the Scene and if so, we call the **getTextureColor** method, otherwise we simply return black. No further check if the ray intersects with objects in the scene is needed, since this is already checked in the **renderRay** method, inside of **recursive.cpp**, where the **sampleEnvironmentMap** method is called.

1.2 Visual Debug

In order to visualize this feature, we created a red sphere centered at the origin inside of the Custom scene in **scene.cpp**, together with the cube map surrounding it. In order for the cube map to fully encapsulate the sphere, the scale factor needs to be at least 5. We were not able to encode infinity as the scale factor for the cube, but we did try multiple high values and the changes in behaviour seem to be capped out after a certain large number, as we can see in the renders. The effects of applying a cube map on a sphere are clear in the renders: we can see the corners of the cube very clearly in the reflection of the sphere. A more suitable shape for this case would of course be a sphere map. However, if we wanted to capture the details of a room on reflective objects inside the scene, the cube map is superior.

1.3 Rendered Images

scaleFactor = 5

scaleFactor = 30

scaleFactor = 100

scaleFactor = 10000

1.4 Location

File: *extra.cpp*
Methods: **sampleEnvironmentMap**, **getTextureColor**

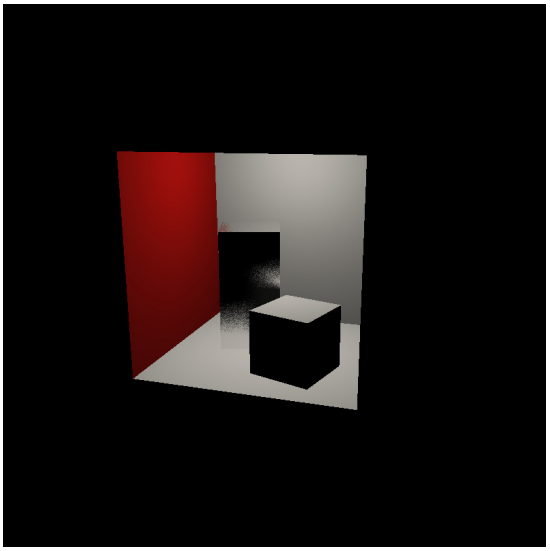
2 Glossy Reflections

2.1 Description

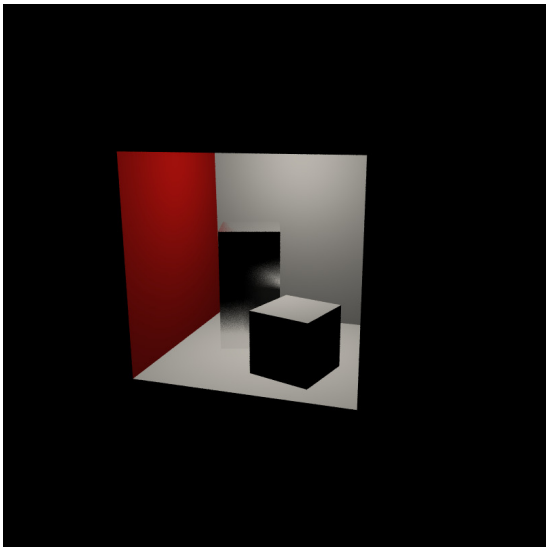
In the **renderRayGlossyComponent** method, we first find a basis for the plane orthogonal to the reflected ray and then sample multiple points (as many as **numSamples**) inside of a disk with radius **diskRadius** and for each one, we update the **hitColor** proportionally. The choice of the formula for **diskRadius** is justified because when the

shininess is high, the disk radius decreases and when the shininess is low, the disk radius increases, just as we want. Multiple glossy samples give us a better quality reflection, however as we raise **numSamples**, the computation time increases significantly, so we have displayed renders for **numSamples = 10** and **numSamples = 20**.

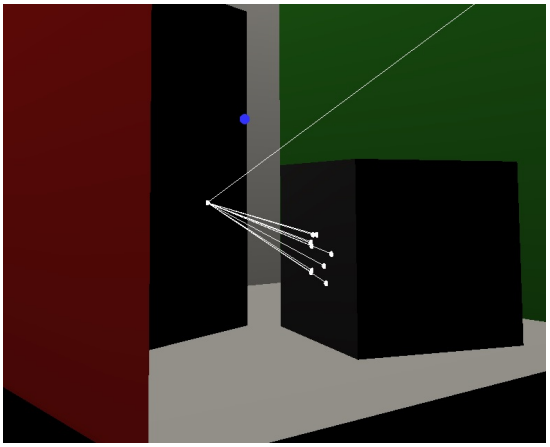
2.2 Rendered Images



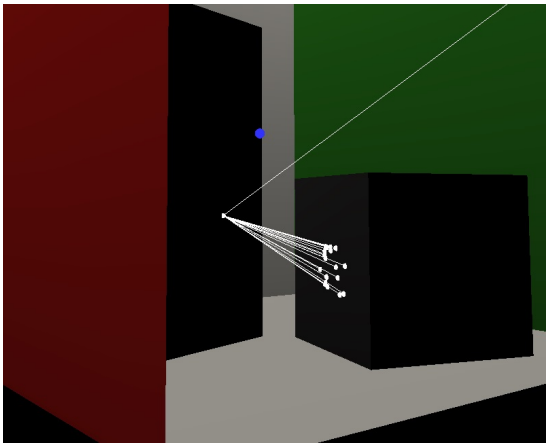
numSamples = 10



numSamples = 20



numSamples = 10



numSamples = 20

2.3 Location

File: *extra.cpp*
Methods: **renderRayGlossyComponent**

3 Bloom Filter post-processing

3.1 Description

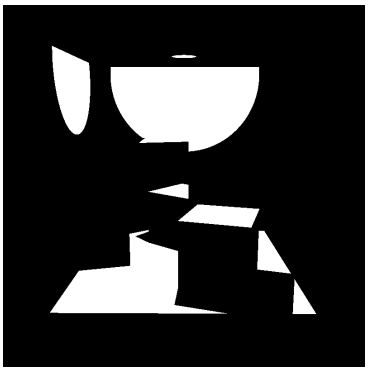
This code implements a bloom post-processing effect by identifying bright regions in an image and applying a weighted blur using binomial coefficients or grayscale intensity-based filtering. The function **binomialCoefficients** computes Pascal's Triangle values, which are used in **calculateWeightsBinomial** to generate a normalized filter kernel.

Bright areas are identified, and isolated in a mask. The bloom effect is applied by first filtering the mask by using a mapping function and then performing a horizontal and vertical Gaussian blur using precomputed weights. Finally, the blurred bloom layer is added back to the original image to create the final effect. The size of filter is taken as a user input from a slider that is defined in main.cpp and changes a variable 'n' in common.h. Additionally, a second button allows the user to upload a grayscale image. The **translateImageToScreen** function extracts a $(2n)^2$ pixel region from the original image, ensuring the processing area is correctly sized. After the image is converted to a filter, **calculateWeightsGrayscale** derives weights from the relative brightness of pixels.

The **postprocessImageWithBloom** function orchestrates this process, determining whether to use binomial or grayscale-based weights before calling **applyFilter**, which performs the entire bloom filtering pipeline. This is done by first extracting bright areas of the image using a threshold-based mapping (binary, linear, or truncate). It then performs a two-pass blur: first horizontally and then vertically, using a weighted sum of neighboring pixels based on precomputed weights. The resulting blurred glow is combined with the original image to enhance bright regions, simulating the bloom effect.

The mapping options I have picked are the standard binary: which sets the value of the bright pixels to 1, truncate, which doesn't change these values, and linear mapping that applies a function: (brightness - threshold)/(1 - threshold). By experimenting with different thresholds and filter sizes, I believe that the linear filter gives the best looking image because the others were too bright in my opinion.

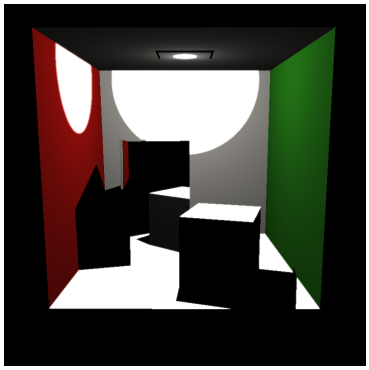
3.2 Visual Debug



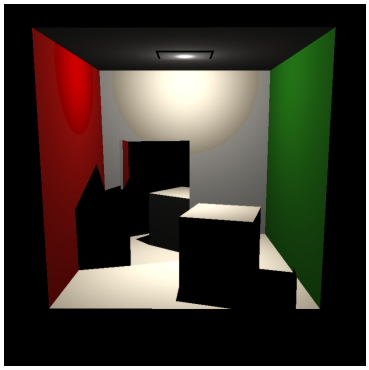
Mask of the image

Bloom applied to mask

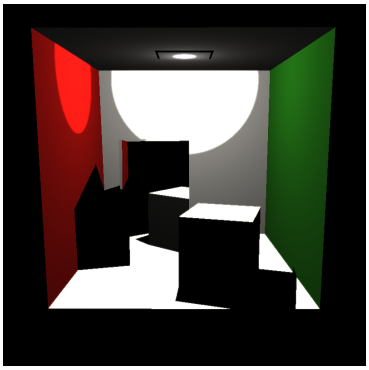
3.3 Rendered Images



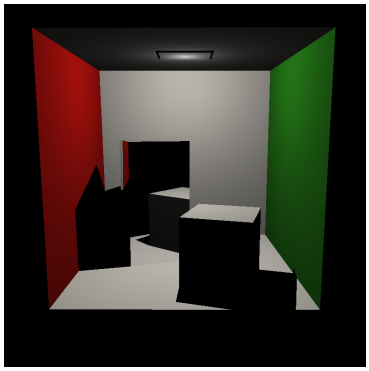
Binary Map



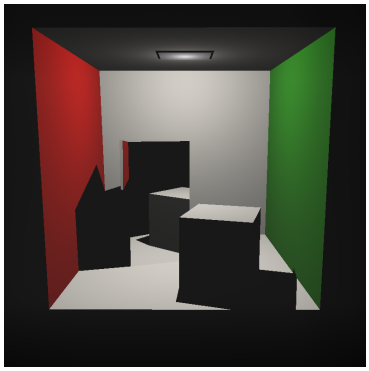
Linear Map



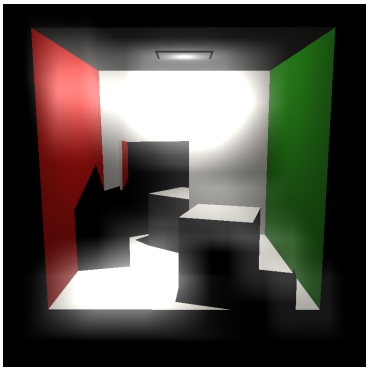
Truncate Map



No filter = 100



Grayscale big filter size



Grayscale small filter size

3.4 Location

File: *extra.cpp*
Methods: **postprocessImageWithBloom**, **binomialCoefficients**, **calculateWeights-Binomial**, **calculateWeightsGrayscale**, **translateImageToScreen**, **binaryMapping**, **linearMapping**, **truncateMapping**, **applyFilter**

4 Sources

Sousa, Tiago, "Adaptive Glare," in Wolfgang Engel, ed.,ShaderX3, Charles River Media,pp. 349–355, 2004.

Gallagher, Benn, and Martin Mittring, "Building Paragon in UE4,"Game Developers Con-ference, Mar. 2016.

Moller, Tomas. Real-Time Rendering, Fourth Edition Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michal Iwanicki, Sébastien Hillaire. CRC Press, 2019.