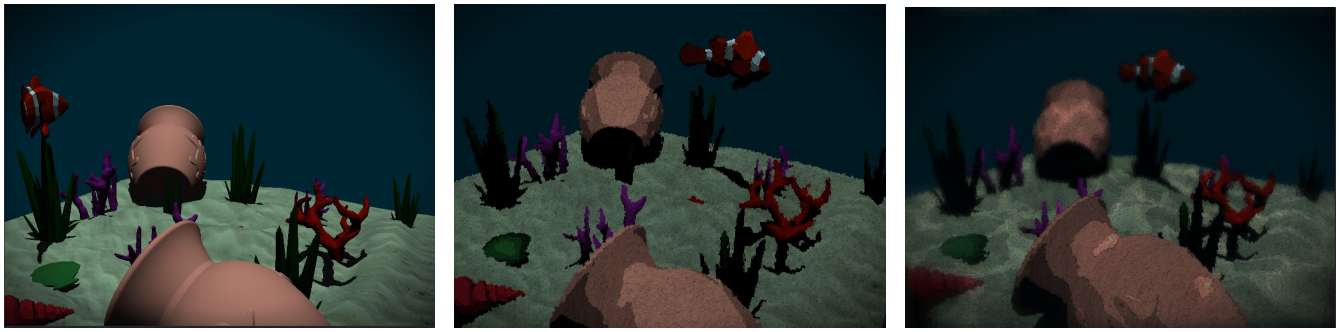
Watercolor rendering and caustics effect for underwater scene

Arnaud Paré-Vogt , **Mehdi Chaid**
Département GIGL Polytechnique Montreal
arnaud.parevogt@gmail.com, mehdi.chaid@polymtl.ca



(a) Scene with simple diffuse light. (b) Scene with watercolor effects. (c) Final scene with all effects.

Figure 1: Final underwater scene with multiple levels of effects.

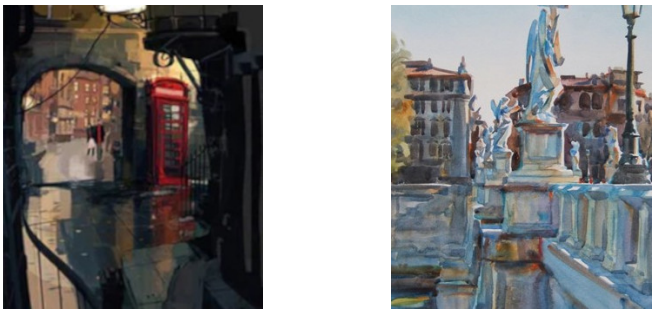
Abstract

Watercolor rendering is a unique non-photorealistic rendering (NRP) technique, used to simulate hand painting art on a 3D scene. In this paper, we attempt to combine the effects of watercolor rendering with additional caustics and underwater effects in order to produce an abstract scene with a swimming fish in OpenGL. Figure 1 shows the resulting scene with various levels of effects applied.

1 Introduction

The use of watercolor in the 3D industry is rare and usually limited to static images. It goes again the general trend to make environments more realistic and detailed. This unique style, on the other hand, allows for some interesting variations of saturation and brightness, as well as vibrant, eye-catching colors that no other medium can match. Creating pleasant watercolor results requires careful planning, as well as a great deal of artistic understanding and experience.

When used properly, this mix of bright, dancing colors and textured brush strokes creates images that appear simple at first glance, while retaining a surprising amount of details. As such, this style is often used in architectural visualisation to produce rich images that allow for a quick understanding of the content, as can be seen in Figure 3.



(a) Dark, narrow street in London. (b) Ponte San Angelo ©Gerald Fritzler.

Figure 2: Watercolor style for architectural visualisation

Please note, however, that these examples were produced in a watercolor style that differs from the one presented in this paper.

Our main inspiration for this work comes from our wish to experiment with cel shading in an underwater scenery. While looking for reference material, we stumbled upon a relevant video that implement this effect using Blender [Oberholster, 2017]. We decided to push the idea further by mixing in a watercolor effect as additional passes on top using a paper from [Bousseau *et al.*, 2006].

(a) (b)

Figure 3: Some results from the reference watercolor paper.

In this paper, we present a brief overview of the scene content, and project framework. We then go over the general pipeline used to produce the results in Figure 1, before diving into the implementation details of each step in their specific section. Lastly, some interesting results and research discussion will be mentionned, before concluding.

2 Scene

2.1 Framework

The source code for the project can be found on Github [Chaid and Paré-Vogt, 2020]. The code was written in C++ with OpenGL 4.6 core. We've used the LearnOpenGL [de Vries, 2014] online resource to bootstrap some of the initial work (camera class, shader class, model loading).

2.2 Content

The scene models an underwater environment with multiple static low polygon meshes scattered around. The floor is made of a square displaced patch produced in Blender, with a sand texture on top. All objects used in this project were not modeled by us, and are available for free on CgTrader [CgTrader, 2011].

In addition, an animated low polygon fish is placed in the water above. Only the sand plane model is textured, all other models use a constant color per face. Figure 4 shows the models of the underwater scene without any effect applied on them.

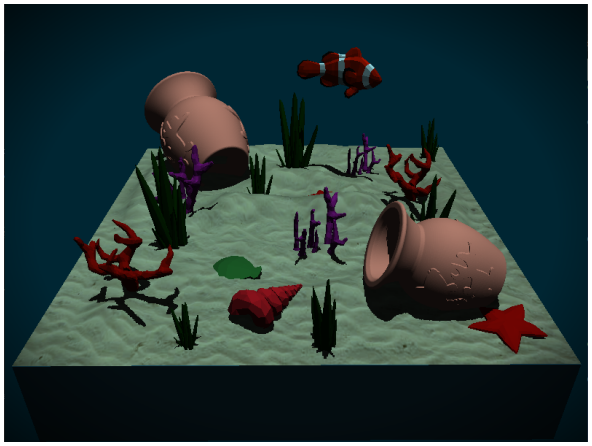


Figure 4: Content of the scene with diffuse light.

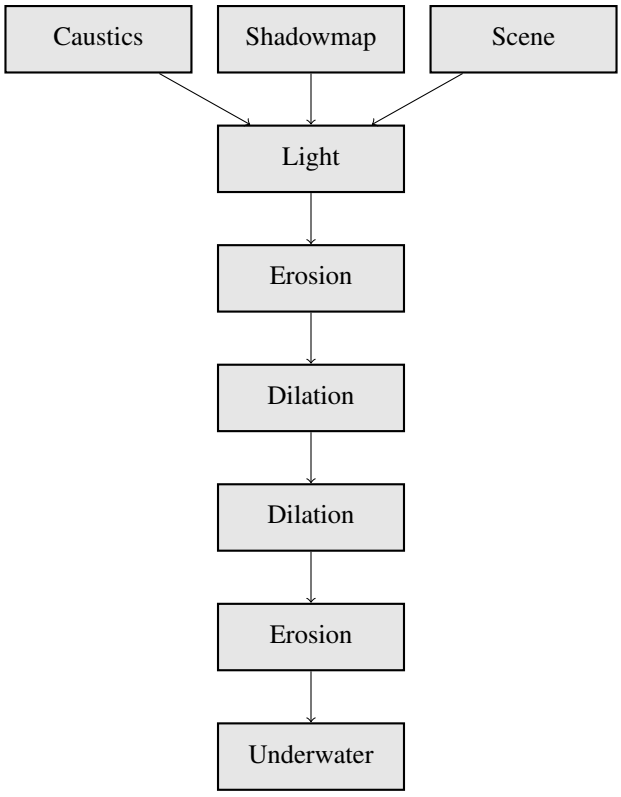


Figure 5: Schema of all the render passes in the program.

2.3 Pipeline

In order to draw an underwater scene, multiple effects must be combined together. This section provides an overview of the rendering pipeline of the project, and explains how each effect has been integrated in the pipeline. For more information about a specific effect, refer to one of the sections below.

Three main effects are added to the scene, and in addition, a very simple shadow map adds some realism. The main effects are enumerated below.

1. Caustics
2. Shadow map
3. Watercolor rendering
4. Underwater effects

The ordering of all render passes is shown in figure 5. Note that a render pass is defined by a series of draw calls that render to a given target. In our case, all render passes except the last one render to framebuffers. The correspondence between render passes and the corresponding effects is described below.

The caustic effect (further described in section 4) works as follows: it renders a caustic texture on a framebuffer and then projects the texture on the scene. This effect therefore requires a separate pass and a dedicated framebuffer for the texture to be rendered in.

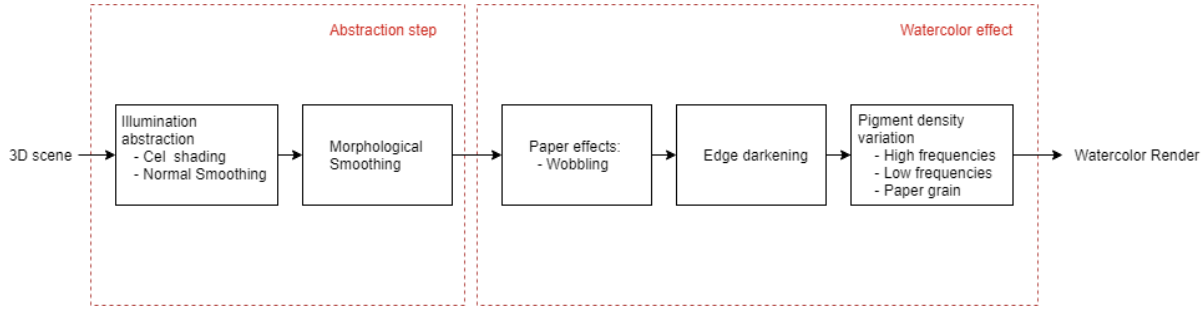


Figure 6: Simplified watercolor pipeline.

The shadowmap works in much the same way, it renders a depth buffer in the point of view of the light, and then a check is made when drawing the light (note that the caustics are part of the light) to see if a given fragment is illuminated or not. Our shadow map implementation is very simple, and does not do a lot of work to obtain good-looking shadows. In practice, the watercolor effect is very prominent, so the irregularities are not very visible.

The caustic texture and the shadow map are applied on the scene in a separate pass, the ”Light” pass. We use deferred rendering to calculate the illumination of the scene, so the light pass requires the position, normals and base colors for every fragment of the scene. The ”Scene” pass is responsible for drawing all objects of the scene in order to retrieve the position, normal and color information per pixel and send it to the light pass.

The watercolor effect (further described in section 3) is a bit more complicated. In order to be able to correctly apply the effect, we need to be able to sample neighbouring pixels. In order to achieve this, the effect is split over multiple passes. Following the light pass, 4 passes perform alternating erosion and dilation effects. Then, the rest of the watercolor effect is applied at the same time as the underwater effects, in the underwater pass.

Finally, the underwater effect (further described in section 5) is applied at the end of the pipeline, in the underwater pass. The underwater effect is therefore computed in the same shader as the end of the watercolor effect. The result is rendered directly on the screen and completes the render pipeline.

3 Watercolor rendering

As mentionned in Section 1, we have based our watercolor implementation on a paper by [Bousseau *et al.*, 2006], with a few modifications to accomodate for the underwater scene context. In this section,we will present the pipeline proposed by the authors to produce the desired effect, as well as the areas where our implementation deviates from the original paper.

Figure 6 shows a simplified version of the pipeline, with only a 3D scene as input, as opposed to a 3D scene or static image used by the authors. We also got rid of the dry brush effect, as it was conflicting with the desired underwater effect.

3.1 Cel shading

The first modification needed for a watercolor effect is to lower the amount of details available in the image by reducing the color variations. On a static image, a color segmentation step, through the use of a mean shift algorithm, would be required to obtain the desired density. In this case however, using the light position and fragment normals, we can achieve good results with a simple cel shader combined with normal smoothing.

For the cel shading, we compute the color of each fragment using a simple diffuse model. We can then transform that rgb color to hsv space in order to access its color intensity value. Through its hsv representation, the color intensity can be clamped to a chosen level using the formula presented in Figure 7. Doing so clamps the intensity value to 4 different levels, while preserving the overall color.

```
vec3 celShading(vec3 rgbCol) {
    vec3 hsv = rgbtohsv(rgbCol);
    hsv.z = (round(rgbCol.z * 3) / 3.0);
    return hsv2rgb(hsv);
}
```

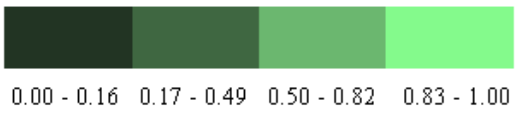
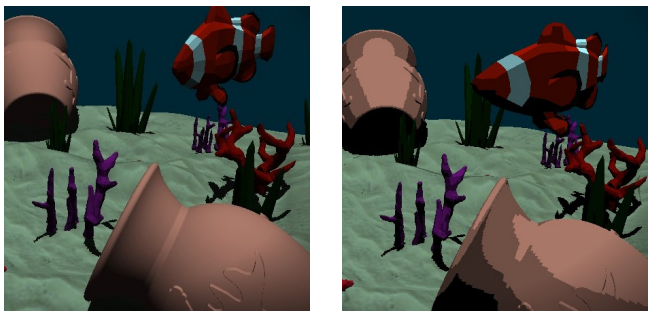


Figure 7: Cel shading color transitions for the green color.



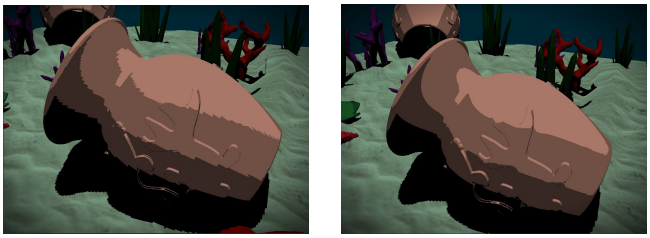
(a) Diffuse shading. (b) Cel shading.

Figure 8: Diffuse vs cel shading with low poly models.

3.2 Normal smoothing

As can be seen in Figure 8, when used on low poly models with curved surface, such as the clay pot, it results in aliasing in the transitions between the levels. To fix that, as well as reduce some additionnal detail in the shape, we use a process called normal smoothing.

Normal smoothing consists of average every vertex normal with its neighbors' normals. Each pass reduce the amount of details in the model, while preserving the overall silhouette intact. The process can be repeated multiple time in order to obtain the desired level of details, or lack of, in this case.



(a) Cel shading, no smoothing. (b) 1 pass of smoothing.

(c) 7 passes of smoothing. (d) 20 passes of smoothing.

Figure 9: Effect of normal smoothing on the cel shading.

Figure 9 shows various level of normal smoothing applied to the pot. We have found that repeating the process a total of 7 times gives the best results across a variety of models, and have kept that for the rest of the experiment.

The initial implementation was done in a fragment shader, using a deferred rendering technique to query the neighboring fragment normals using a post process normal map. The issue with this approach was that we could only do one pass per fragment shader, and that was repeated every frame. To get 7 passes using this technique, we would have to write the normals to a texture and do a pass with the post process shader multiple times every frame.

As a tradeof, in order to save rendering time, we've instead decided compute the normal smoothing on a per vertex basis, on the CPU at load time. This only happens at the beginning of the program once, and can easily be repeated any number of time necessary. The issue however, is that it is done on the CPU, and can slow down considerably the program if a large number of detailed models is used.

In the context of this experiment, the additional load time at the opening of the program was not an issue given the small amound of low poly models used.

3.3 Morphological smoothing

An additionnal morphological smoothing pass can be applied on the resulting image to further reduce the level of details in the scene. Just like most of the other watercolor steps, they are applied on a post-process basis. We've decided to apply it in this case as it helps reducing the sharp noise introduced by the wobbling to an acceptable range, as the reminder of the noise is necessary to obtain the desired watercolor effect.

The smoothing we've applied is composed of an initial opening sequence (Erosion - Dilation) followed by a closing sequence (Dilation - Erosion). Erosion consists of applying a kernel over a black and white image, and setting the target fragment to 1 if every fragment in the kernel is also 1 (white). Dilation is the opposite operation, it sets a fragment to 1 if at least one fragment in the kernel is 1.

(a) Opening sequence. (b) Closing sequence.

Figure 10: OpenCV documentation on morphological operations.

A difficulty arise when dealing with colored images: What do we define as 0 and 1 respectively? Since images are not black and white, each pixel queried by the kernel is likely to be slightly different, and both operations cannot be applied as is. Another consideration is that the morphological operation should not introduce new colors to the image, as the goal is to reduce the amount of details, not increase it.

A possible solution for that is to use a gray-scaled approach to the morphological smoothing, and process each r, g and b channels individually. Alternatively, the smoothing could be applied to the value part on the HSV space of the image. Both of those approach break the color preservation criterion. [Sartor and Weeks, 2001] perform more advanced operations on the chromatic scale, by mixing the reduced and conditional ordering of the underlying image. [Benavent *et al.*, 2012] construct a specific color ordering for each particular image and perform morphological operations using this ordering.

Such extended approaches were not needed in the scope of this experiment, as the morphological smoothing serves as an additional polish between the normal smoothing and the wobbling effect. For this reason, we've settled on a simple, straightforward solution.

A 3x3 kernel is applied over the image, with each r, g and b channel being processed independently. For the erosion operation, the lowest channel value in kernel is taken as the fragment's channel value. For the dilation, it's the opposite, the highest channel value is kept.

Appendix 22 and 24 show an overview of the scene, before and after applying the morphological sequence. Due to the small sequence that was used, only a little bit of the details around some edges were smoothed out. Using a bigger kernel, and possibly a different approach to the smoothing could yield better results.

Appendix 23 and 25 show the result of each operation. Erosion expand and darken the edges around the objects. It can be used in conjonction with the edge darkening effect used later on in the pipeline. Dilation, on the other hand, lighten the edges and make the objects thinner.

3.4 Wobbling effect

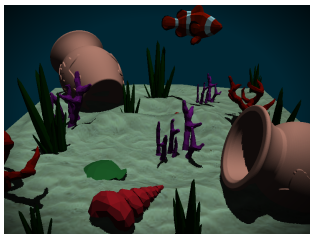
In order to simulate the displacement of paint due to the paper height, a wobbling effect needs to be added. The wobbling effect consists in displacing fragments along the viewport, based on the paper height.

In practice, in order to get the horizontal and verical displacement of the pixel, the gradient of the paper height is used. To compute the gradient, the finite element method is used. The following code listing shows how the gradient of the paper height map:

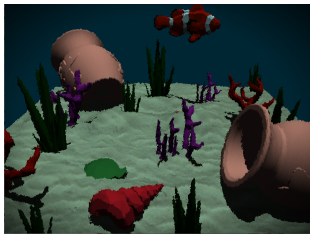
```
vec2 getPaperGradient(vec3 tex) {
    float wobF = 0.05;

    float x_p = tex + vec2(wobF, 0);
    float x_m = tex - vec2(wobF, 0);
    float y_p = tex + vec2(0, wobF);
    float y_m = tex - vec2(0, wobF);

    float dx = texture(paper, x_p).r
             - texture(paper, x_m).r;
    float dy = texture(paper, y_p).r
             - texture(paper, y_m).r;

    return vec2(dx, dy);
}
```

Here the `paper` texture corresponds to the height of the paper. This was obtained by using pictures of paper. The `wobFactor` variable is a parameter that is adjusted depending on the texture so that it corresponds to the difference between two pixels. Once the paper gradient is computed, the fragments position is changed based on the gradient (multiplied by a parameter to control the intensity of the effect). This is done by adjusting the coordinates at which the fragment is sampled (`vertTexCoord`).

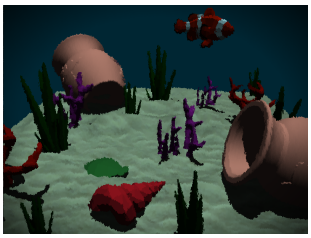
Figure 11 shows the effect of the wobbling on the scene. Adding the wobbling effect makes the edges of the object in particular sand out, and is not very visible inside the object, since the color is fairly constant. In addition, when the wobbling effect is used along the morphological smoothing, the morphological smoothing effect diminished the intensity of the wobbling effect.



(a) No effects.



(b) Wobbling effect.



(c) Wobbling with morphological smoothing.

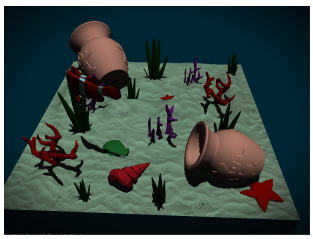
Figure 11: Effect of wobbling.

3.5 Edge darkening

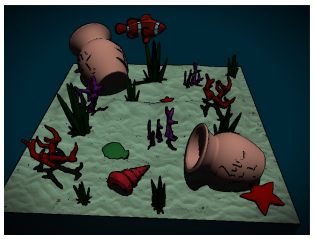
A simple sobel filter (Figure 12) is applied over the image during the underwater pass to detect edges and darken them. We initially set the fragments with a color length, computed as the simple vector length of the rgb color, of over 0.9 to black. The effect was however too strong, and while it was quite aesthetically pleasant, it did not match the watercolor context.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 12: Sobel 3x3 kernel.



(a) No edge darkening.



(b) Initial edge darkening.

Figure 13: OpenCV documentation on morphological operations.

Instead, the new approach was to apply a factor of 0.7 to the intensity value of the fragment in the hsv space. This results in darkening the edges juste enough for it to be noticeable, without being too strong either. The darkening factor can also be tweaked depending on the needs.

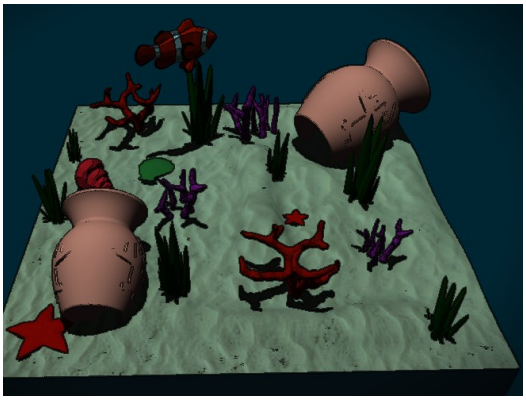


Figure 14: Final edge darkening effect.

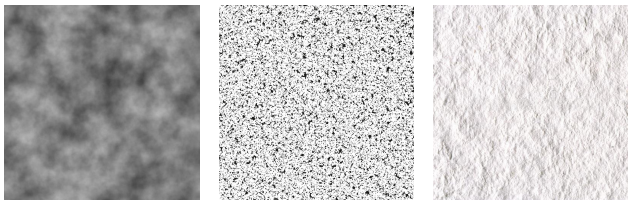
An issue with the current approach is that, since the edge darkening is handled in the underwater pass, the image already contains the caustics from the light pass. As such, using a strong edge darkening results in the transition between water and light being darkened as well, which is undesirable. Moving that step back to the light pass would interfer with the morphological smoothing. A solution to this is mentionned in Section 7.

3.6 Pigment density variation

Variation in the pigment density refers to the various paper effects distinctive of the watercolor style. Such variation usually comes as a result of the lavis, wet on dry, technique when painting. [Bousseau *et al.*, 2006] decompose this effect into three parts: Low frequency variations due to the repartion of water on the canvas, high frequency variation due to the repartition of the paint on the water, and finally, paper grain, due to the canvas' rough surface itself.

Representing those three pigment density variations in OpenGL is quite straightforward. Each variation is produced per fragment on the fragment shader, using a gray-level texture. The turbulent flow (low frequencies) is simulated with a Perlin noise texture, the pigment variation (high frequencies), with a Gaussian noise texture, and the paper grain is a simple paper texture.

These effects are applied during the final rendering pass, using the pixel positions to offset the texture sampling. This allows for better space coherence when moving around in the scene while preserving the pigment dispersion at the same location. An alternative to this is shown in Section 7.



(a) Low frequency. (b) High frequency. (c) Paper grain.

Figure 15: Pigment density variation.

4 Caustics

In order to render caustics, we use a method called Periodic Caustic Textures [Stam, 2004]. This method is relatively simple, and the biggest advantage over other methods (for instance rendering caustics using photon mapping [Johannes Günther and Slusallek, 2004]) is that it is easy to implement and not very computationally expensive. The method used also achieves good-looking results, compared with the GPU Gems method [Guardado and Sánchez-Crespo, 2004], which we also tried to implement.

4.1 Algorithm

First, a grid with a certain resolution is generated. The grid is then rendered on a framebuffer. The vertex shader used in the rendering changes the x and y coordinates of the grid vertex by refracting the vertex as if it was a ray of light coming from above. The distorted mesh is then drawn in the fragment shader, using the ratio of the new area of a triangle over the original area of a triangle to calculate the color. This way, if the vertices are sent close together, the caustic color approaches white, while if the vertices are sent apart, the caustic color approaches black. Figure 16 shows a visual representation of the algorithm used to generate caustics.

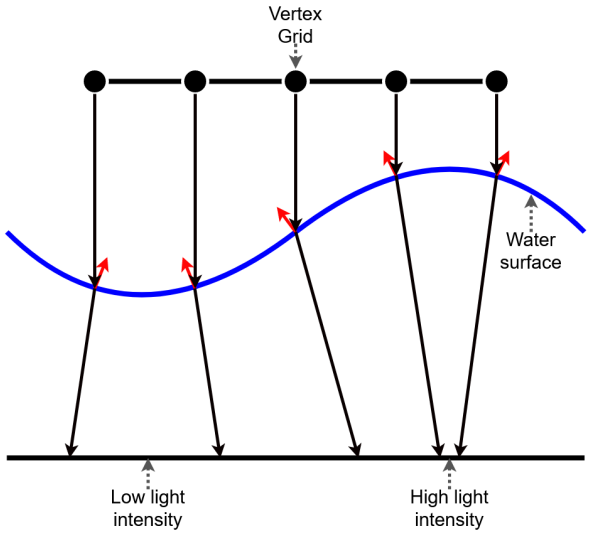


Figure 16: Schema of the algorithm used to generate the caustic texture.

With this algorithm, there is a risk that a vertex on the edge of the camera travels towards the inside of the viewport of the camera, leaving an unrendered section on the side of the texture. To prevent this, the grid is created too big (in our implementation by a factor of 1.2). This ensures that the texture is always rendered.

Figure 17 shows the caustic texture generated by this approach. No that the squares of the grid are still visible by looking closely at the image despite the resolution of the grid being high. However, once the toon shading has run, these are no longer perceptible.

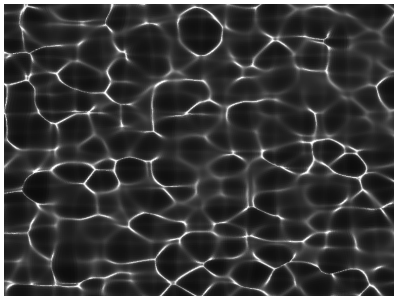


Figure 17: Caustic texture generated from our implementation.

4.2 Water height

In order to generate the caustic texture, the algorithm must refract and project on the ground the grid vertices. This means that the algorithm needs access to a water height and a water normal for each vertex. We can compute the normal if we have access to the water height by using the finite element method. To obtain the water height, a fractal noise (in our case 3 layers of Perlin noise) is generated in the caustic vertex shader. In order to generate the Perlin noise, an implementation from Stefan Gustavson was used [Gustavson and Vivo, 2014].

To animate the water height over time, we used a 3d Perlin noise, with the time as one dimension. This generates unrealistic water heights, but since we only use the water height to compute the caustics, the end result still looks good.

4.3 Projection on the scene

Once the caustic texture is generated, we project the texture on the scene to give the impression that the scene is illuminated by the caustics. This is an approximation, since we assume that all light rays in the water travel vertically, but the result is good enough. Figure 18 shows the scene with the caustics projected on it.

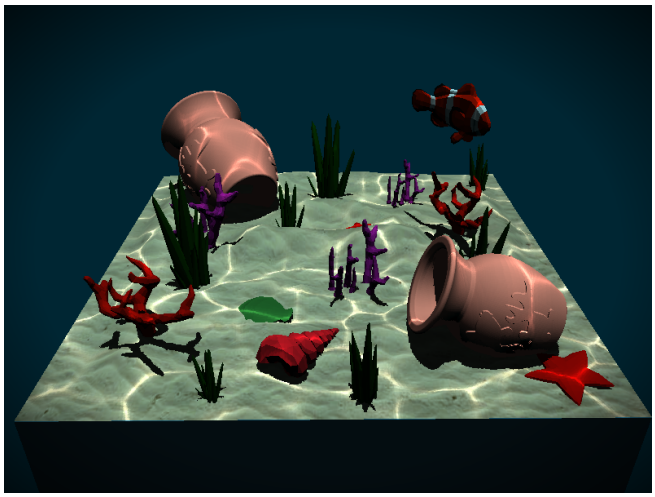


Figure 18: Scene with the projected caustic texture.

5 Underwater effects

In order to make the scene look like an underwater scene and make the final result more attractive, three additional effects have been implemented. These effects are not based on a physical model, but are purely cosmetic.

5.1 Blur

The first effect is a blur. This simulates the blur that is seen underwater with a human eye. The blur we added is dependent on two factors, the distance of a fragment to the camera (fragments further away are blurrier), and the distance of the fragment to the center of the viewport (fragments on the side and in the corners of the viewport will be blurrier). This effect is not realistic, but still gives the feeling of an underwater scene.

In order to implement the blur underwater, a 5×5 Gaussian kernel was used. The kernel is scaled on the image using a range parameter in order to have different blur intensities without changing the speed of the algorithm. This kernel is convolved twice, once using a range of 3 and once using a range of 10. The result of both of these blurs is then mixed with the non-blurred color, using two factors, one dependent on the depth of the fragment and one dependent on the distance to the center of the viewport.

In order to have blur that augments fast passed a certain distance, before being used to mix the colors, the depth of the fragment is raised to the third power ($factor = depth^3$). This gives a better effect for the distance blur, as when the camera is close to an object (distance below 1), the blur is almost non-existent, and one the threshold is passed, the blur increases rapidly. Appendix 26 shows the scene with and without any blur.

5.2 Radial darkening

The second effect is a radial darkening. This means that fragments on the edge and corners of the screen are darker. This darkening was added in order to make the end image more attractive. This is a fairly common effect, and is easy to implement. The code for the effect is shown on the right.

```
float vpDist = length(texCoords - 0.5);
if (vpDist > 0.45) {
    color *= 1 - remap(vpDist, 0.45,
                       sqrt(0.5), 0, 0.7);
}
```

Appendix 27 shows the results of radial darkening on a scene. Notice the corners of the image. The effect is minimal, and can be hard to notice if it is not pointed out, but adds to the overall image attractiveness.

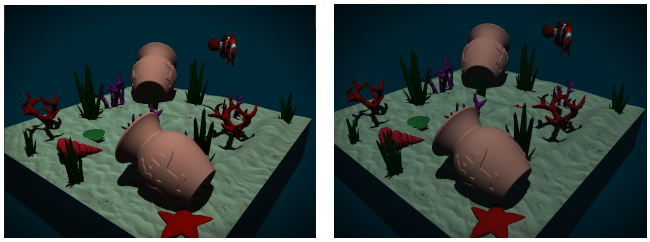
5.3 Color tint

Water absorbs certain light wavelengths faster than others. In particular, red and green light are absorbed by water, so often underwater scenes take a blue tint. In order to create a similar effect, but without making it physically accurate, we tint the

entire scene with a blue color. All fragments of the scene are mixed with a color according to the following formula:

$$color = \text{mix}\,(color, tint, 0.75)$$

Where the variable tint is the tint color, defined with the rgb values $(0.0, 0.07, 0.1)$. Figure 19 shows the effect of adding the color tint on the scene.



(a) Without tint. (b) With tint.

Figure 19: The effect of color tint on the scene.

6 Results

Figure 20 shows the resulting image after all mentionned effects have been applied to the scene.

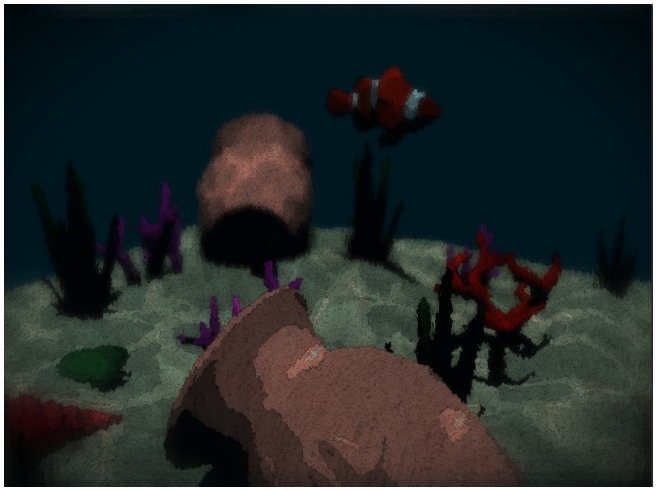


Figure 20: The scene with all effects.

Larger figures are also available in the appendix. Apendix 30 shows the scene from a closer point of view, Appendix 31 shows the scene from a far point of view and lastly, Appendix 32 shows the animations of the caustics through pictures taken at several intervals of time.

7 Future Work

The current implementation has multiple points that could make the results better. This section outlines the major changes that could make our final result better.

7.1 Pigment density in screen space

One of the main problem of the pigment density variation effect is that to be accurate, the effect has to be applied in screen space. The problem with applying the effect in screen space is that when the objects or the camera moves, it gives the impression of a "dirty lens," and the illusion of watercolor is completely broken.

To counter this effect, [Bousseau *et al.*, 2006] suggests two methods that allow applying a texture in screen space, but that allow the texture to follow the scaling and displacements of objects.

Our current object space implementation does not suffer from the dirty lens problem, but faces another issue: The sampling is so dense in object space that some of the pigment variations, namely the high frequencies and paper grain, appear dirty. Scaling the sampling up spread out the pigments and make the effect barely visible.

As such, a tradeof would be to implement the low and high frequencies in object space, and the paper grain in screen space. Better textures could also be generated to mitigate the problem in object space.

7.2 Improving Gaussian blurring

Our current solution uses a 5×5 Gaussian kernel, but using a constant size kernel makes a variable blur impossible. In order to have varying degrees of blurring, we use a variable stride when running our convolutions.

This technique generates artefacts (see figure 21), and these artefacts could be avoided by using a more computationally expensive type of blurring. For instance, a 3×3, a 5×5, a 7×7 and a 9×9 kernel could be run in parallel, and the resulting blur could interpolate between all these blurs.

This approach may be a bit slower, but the Gaussian blur is linearly separable and the computation can be greatly sped up by decomposing it into a vertical and horizontal pass.

Figure 21: Artefacts observed when blurring.

7.3 Collision-less edge darkening

The implementation of edge darkening presented here possesses one glaring issue: When the darkening factor is high, artefacts can be seen around the edges of the cel shading-induced shadows and the caustic lights on the objects, most noticeably on the ground.

This is due to moving the edge darkening process to the final pass, and using the resulting render from the light pass which contains the cel-shading and caustics as a flatten image. This issue was not present during the early days of the project, before introducing the morphological smoothing which required a separate pass after the caustics and cel shading for maximal effect.

A simple, yet effective improvement for that would be to render the scene in two textures during the light pass, once with the effects and once without. The edge darkening process during the underwater pass could then simply sample from the effect-less texture to apply the Sobel kernel, and return the effect-full texture in case of an non-edge fragment.

The additionnal cost wouldn't be too constraining either, as it merely consists of adding one additionnal texture to the output gbuffer of the light pass, which already outputs the color, the position and the normal of each fragment.

7.4 Morphological smoothing operations

As explained in Section 3.3, the algorithm we have chosen for this feature is quite simple and limited, we simply choose the darker value in a 3x3 kernel for each channel, for each fragment during the erosion pass, and the lightest value for the dillation pass.

This technique is not realistic, introduces additionnal colors which is a behavior that the morphological smoothing process tries to avoid, and it is applied on a relatively small kernel. For these reasons, there are big rooms for improvement here.

We could, for example, increase the kernel window to sample in a bigger range (5x5 or evene 9x9), this would result in more details being removed from the image. Another approach is to implement one of the algorithms mentionned in the additional papers [Sartor and Weeks, 2001], [Benavent *et al.*, 2012].

Doing so would most likely be time and computationally expensive, but would result in a sparking improvement in the resulting image, with an increased watercolor sensation.

8 Conclusion

To conclude, we have shown that it is possible to create and render an underwater scene with the use of a watercolor-like effect, through the technique described in [Bousseau *et al.*, 2006]. In addition, this effect blends well with other effects, such as caustics and underwater-specific effects, such as a sharp depth blur and edge darkening.

Although there are large rooms for improvement in this paper, this experiment has shown promising results in the field of non-realistic rendering, and could be used to induce various projects with a unique style.

As mentionned in Section 2.1, all the source code produced here by us is available on Github and contain instructions on how to tweak and navigate the scene for those interested in mingling further with this experiment.

References

[Benavent *et al.*, 2012] Xaro Benavent, Esther Dura, Francisco Vegara, and Juan Domingo. Mathematical morphology for color images: An image-dependent approach. In *Mathematical Problems in Engineering (vol. 2012*, 2012.

[Bousseau *et al.*, 2006] Adrien Bousseau, Matt Kaplan, and Francois X. Sillion. Interactive watercolor rendering with temporal coherence and abstraction. In *In Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pages 141–149, 2006.

[CgTrader, 2011] CgTrader. Cgtrader 3d models. https://www.cgtrader.com/, 2011. Accessed: 2020-12-15.

[Chaid and Paré-Vogt, 2020] Mehdi Chaid and Arnaud Paré-Vogt. Watercolor underwater scene in opengl. https://github.com/JeyzerMC/Projet-INF8702, 2020.

[de Vries, 2014] Joey de Vries. Learn opengl. https://learnopengl.com/, 2014. Accessed: 2020-12-15.

[Guardado and Sánchez-Crespo, 2004] Juan Guardado and Daniel Sánchez-Crespo. *GPU Gems*, chapter Rendering Water Caustics. NVIDIA Corporation, 2004.

[Gustavson and Vivo, 2014] Stefan Gustavson and Patricio Gonzalez Vivo. Classic perlin 3d noise. https://gist.github.com/patriciogonzalezvivo/670c22f3966e662d2f83, 2014.

[Johannes Günther and Slusallek, 2004] Ingo Waldy Johannes Günther and Philipp Slusallek. Realtime caustics using distributed photon mapping. *Eurographics Symposium on Rendering*, 2004.

[Oberholster, 2017] Marius Oberholster. Cycles toon shading tip - caustics [tutorial]. https://www.youtube.com/watch?v=JXuiocIkYr0, 2017.

[Sartor and Weeks, 2001] Llyod J. Sartor and Arthur R. Weeks. Morphological operations on color images. In *Journal of Electronic Imaging*, pages 548–559, 2001.

[Stam, 2004] Jos Stam. Periodic caustic textures. https://www.dgp.toronto.edu/~stam/reality/Research/PeriodicCaustics/, 2004.

9 Appendix

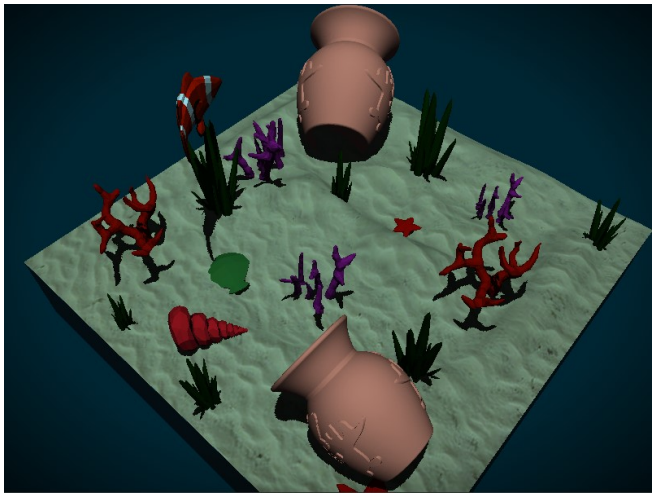


Figure 22: No morphological smoothing.

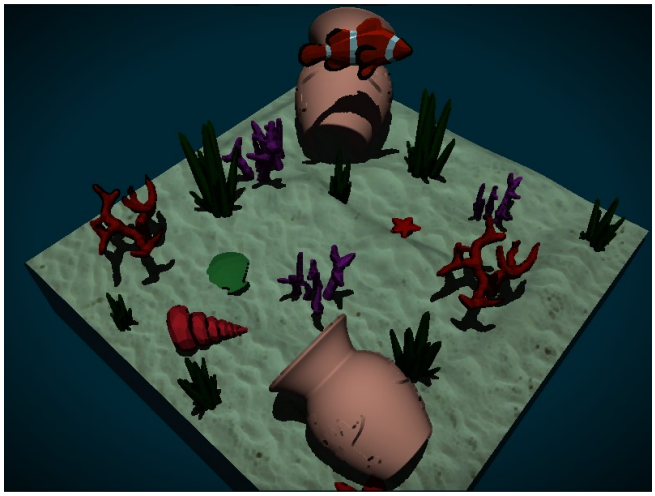


Figure 23: Single pass erosion effect.

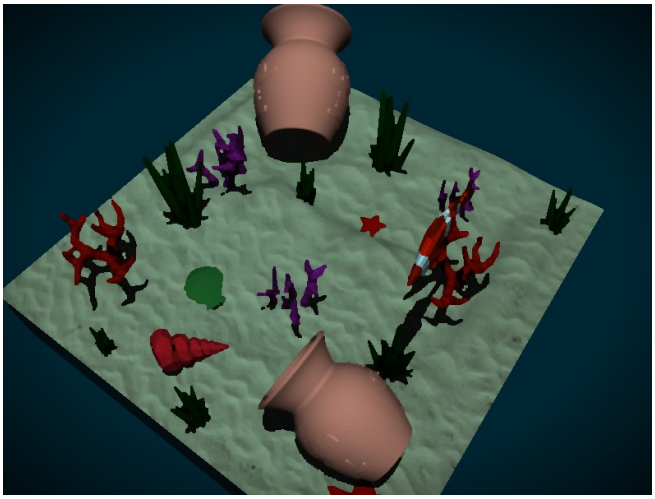


Figure 24: Full opening-closing sequence.

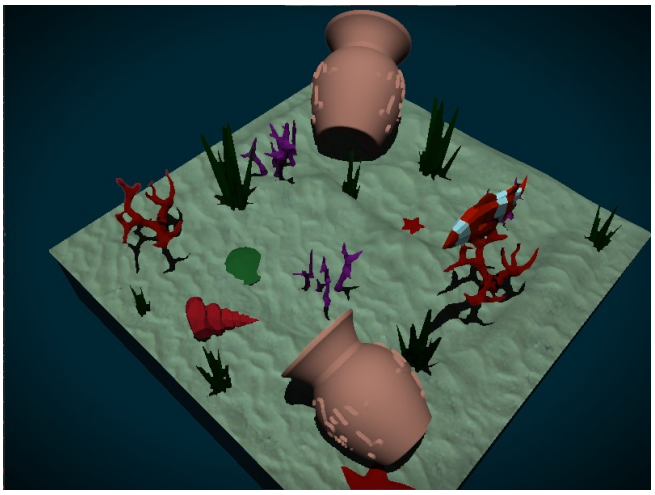


Figure 25: Single pass dilation effect.

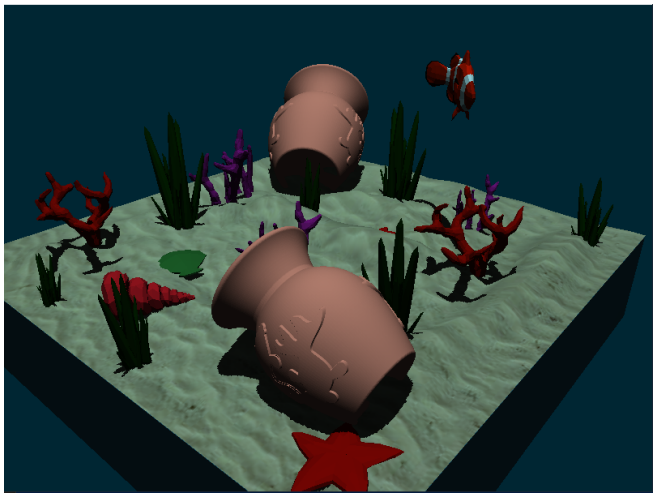


Figure 26: Effect-less point of view.

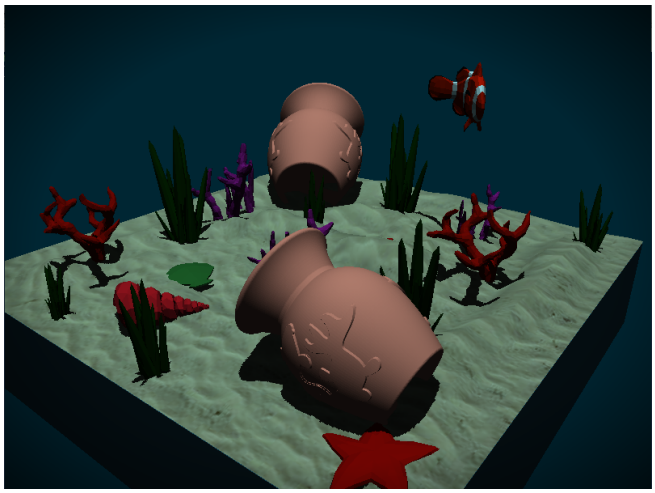


Figure 27: Radial darkening.

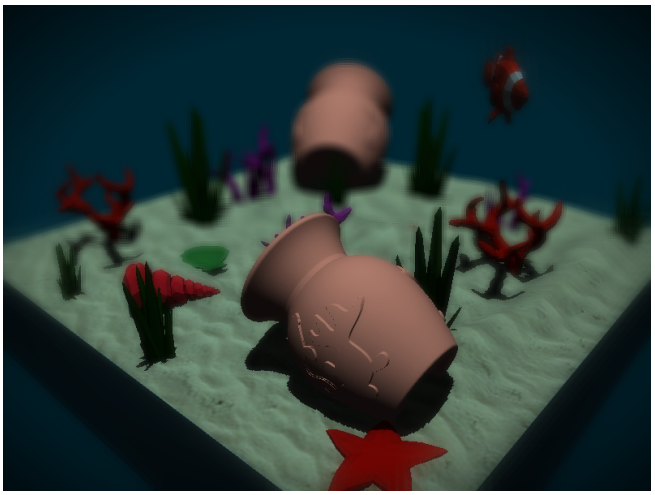


Figure 28: Close up view of the depth blur.

Figure 29: Depth blur as seen from a distant point.

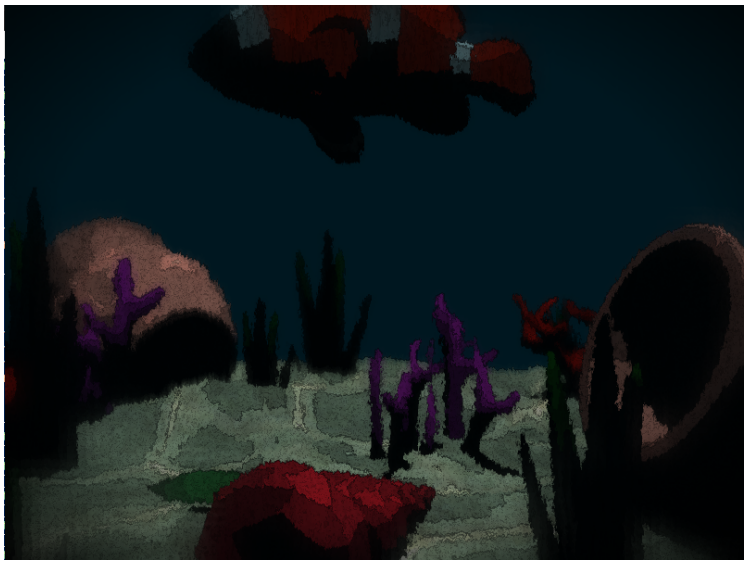


Figure 30: Scene viewed with a close camera.

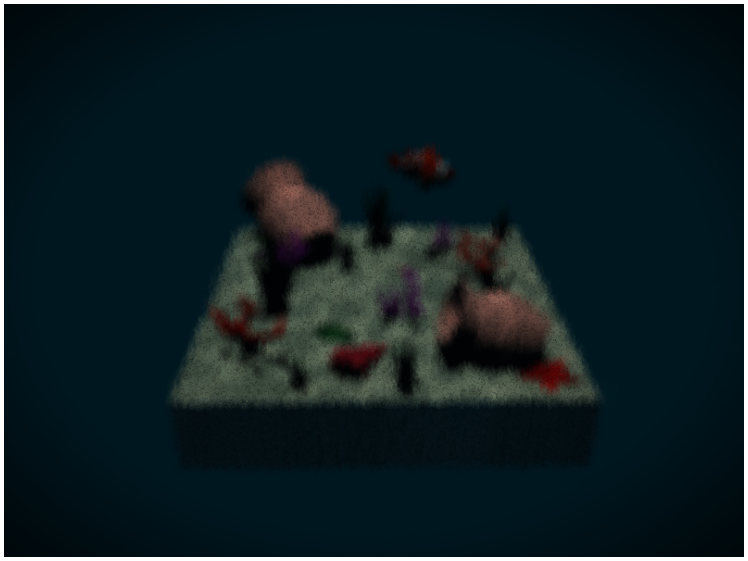


Figure 31: Scene viewed with a faraway camera.

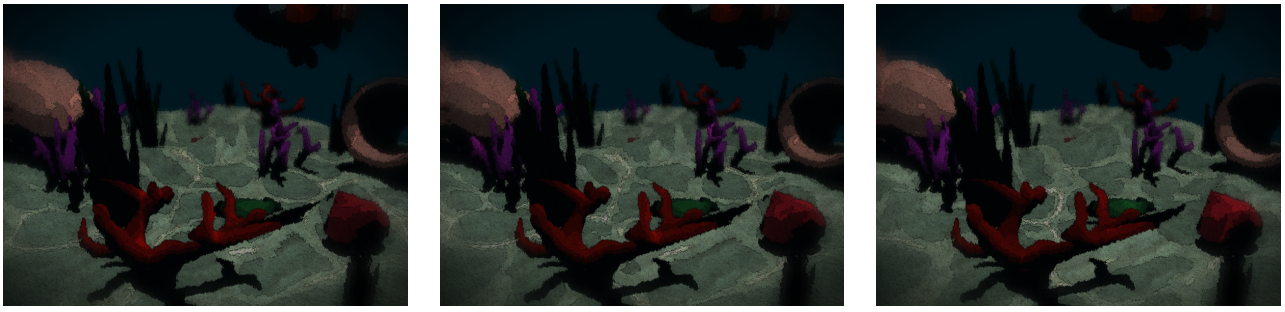


Figure 32: Animation of caustics at 0, 0.13 and 0.26 seconds.