Screen Space Reflections (SSR)

Josselin Somerville Roberts
https://github.com/JosselinSomervilleRoberts/MyOpenGlRenderer.git

***Abstract*—In this paper, we will describe what Screen Space Reflections are and why they are an interesting approximation of real reflections. Then, we will describe in more detail how to compute such reflections by computing ray marches both in the view space and directly in the textures. Some shading tricks will also be detailed to improve the rendering. Finally, we will showcase a few problems with SSR such as ghost shadowing, aliasing, glitches near the borders, and more.**

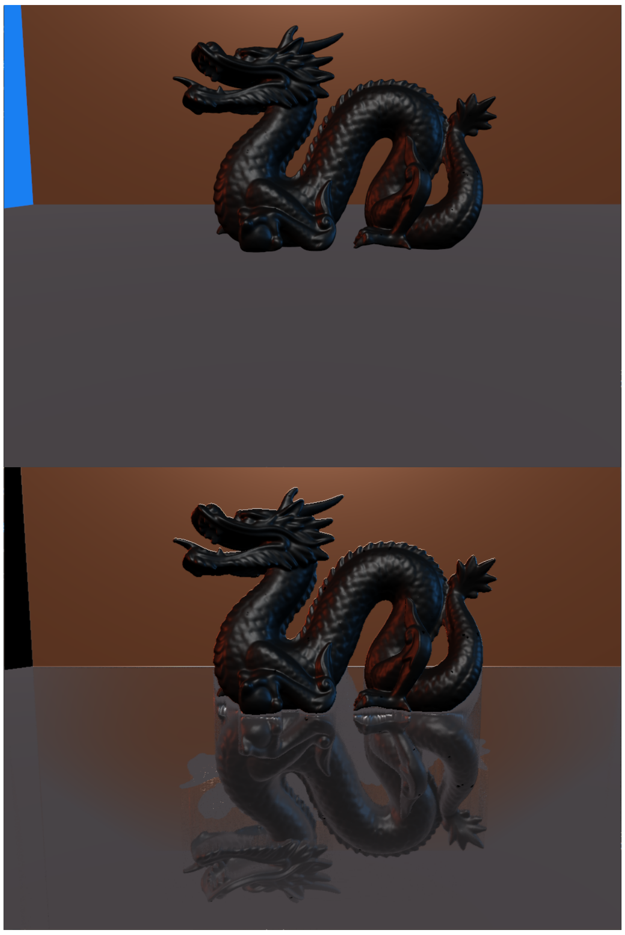
Fig. 1: Without *(above)* and without *(below)* SSR activated

I. Introduction

Adding reflections to a scene is a basic effect increasing greatly the realism of a given rendering. While reflections can easily be computed with ray tracing, this method is incredibly slow as ray tracing is not well suited for parallelism and therefore not able to run on most GPUs. On the other hand, rasterizing is a fast method, that can be computed in parallel. However, most rasterizing pipelines loop through every fragment and save the shaded result based on the depth buffer value. This means that there is no easy way to compute a ray reflection on a fragment and also that even if we do compute such a ray, it is not possible in the rasterizing pass to find if this ray intersects a primitive as the rendering is computed fragment by fragment.

At first thought, cube maps seem like good solutions but they do not allow self-reflection and become extremely slow for moving backgrounds, which is no the case with SSR.

This is where SSR enters. The idea behind Screen Space Reflections is to take advantage of the already rasterized image without reflections. SSR uses deferred shading and while many passes can give better results, only two passes are necessary:

- First do a normal pass to simply save all the information necessary for shading using the depth buffer. In the implementation presented later, the information used is geometry-based (position and normal) and regarding the material (albedo, roughness, and metalicness).
- In the second pass, we simply shade a quadrangle using a classic BRDF thanks to the textures. In addition, we can apply SSR. To do so, for every pixel, if the pixel is metallic, a ray is computed from the camera to the pixel. Thanks to its normal, the reflected ray can easily be computed. Then, the only thing left is to march along the ray and regularly check if the ray intersects an object. To do so, we compute the corresponding UV coordinates of the position along the ray and check the depth buffer to see if the ray intersects the scene. The only thing left is then to shade the pixel if an intersection was found and apply a few tricks to make the results less noisy and better looking.
- (A third pass can be done to filter the results in order to reduce the noise. This is especially useful for rough surfaces which can lead certain pixels to have weird reflections).

While many post-processing treatments such as ray reuse, multilevel raymarching or even temporal filtering can be added to improve the results, in this paper we will focus on only the two first passes and highlight common problems with SSR.

II. Basic Implementation

A. First Pass

The First Pass is pretty straightforward. On the C++ side, we initially create the **gBuffer**, a frameBuffer used to store the textures. In order to have as less textures as possible, we store the position (3D) and depth (1D) in one 4D texture, the normal (3D) and the roughness (1D) in another 4D texture and the albedo (3D) and the metalicness (1D) as the last texture. We decided not to handle transparent materials as it would complicate the SSR process *(a solution would be to render transparent objects after opaque objects, sorted by distance to the camera without overwriting the depth buffer as it is usually done. Then to handle reflections, the closest depth of an opaque object could be stored as the depth but for the color, there are no good solutions. One solution would be to simply ignore the transparent objects for the reflections. Another solution would be to change the albedo of the opaque fragment due to ray going through the transparent object -* $albedo_{new} = (albedo_{opaque} + \alpha * albedo_{transparent})/(1+\alpha)$. *However, this solution would lead to some artefacts as there are no guarantees that the ray would go through this transparent fragment; in fact, this is only true for rays parallel to the view direction (therefore rays are not deflected)).*

At first thought, it seems like shading fragments and saving the resulting colors to a texture would increase the performance as some fragments will be used for several pixels: one time for the corresponding pixel and other times if a reflected ray hit this fragment. However, the view positions will be different for every shading of the same fragment. For the direct rendering, the view position is simply the camera position but for reflected rays hitting this fragment, the view position will then be the camera position reflected. Therefore, saving the shaded result will not make a big difference, this is why shading is done during the second pass.

B. Second Pass

During the second pass, we render a basic quadrangle using the previously filled textures. As explained above, for every pixel we shade the pixel and compute a reflection.

Initially, to find an intersection, the method implemented was a simple linear search, searching at regular intervals on reflected rays. This search uses 3 parameters:

- $SSR_{linearsteps}$, the number of iterations
- D_{max}, the maximum distance of the reflection (reflection further than that will not be detected).
- t_{ray}, the thickness of the ray. If the difference between the ray depth and the texture depth is less than the thickness of the ray, then we consider that there is a hit. *The thickness needs to be set with respect to the number of iterations as the less iteration there are, the bigger the steps will be, therefore the thicker the ray must be.*

The ray marching is done on the view space. To improve a little bit the performance, this search can be directly done in the textures. To do so, we compute two points along the reflected ray, one at the fragment position where the ray is reflected and the other along the reflected ray at D_{max}. Then, we can convert the view-space coordinates of these points as UV coordinates using the following formula:

$$pos_{projected} = projectionMat * [(pos_{view-space}, 1]$$
$$pos_{UV} = 0.5 + 0.5 * (pos_{projected}.xy/pos_{projected}.w)$$

Of course this requires to check that the end position is still in the texture coordinates and to clip it otherwise, but with this method we get a simple 2D-line to follow in the ray-marching process.

C. Shading

Simply assigning the shaded fragment hit does not yield realistic results, as shown below.

In all the following images, if not precised, the parameters used are the following:

- $SSR_{linearsteps} = 500$
- $D_{max} = 9$
- $t_{ray} = 0.1$

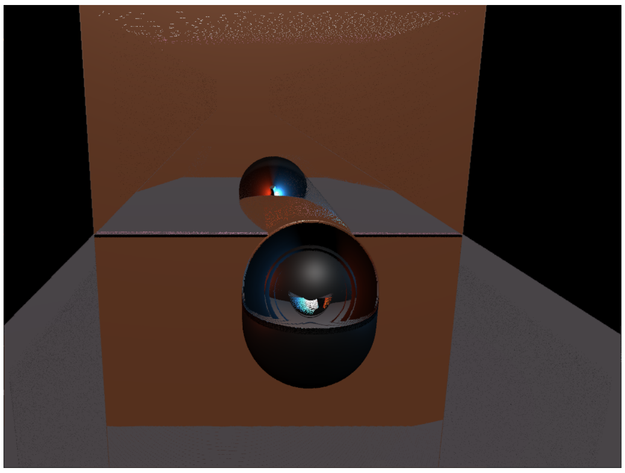

Fig. 2: Assigning the fragment hit shaded color

To solve this issue, we propose a shading taking into account the fragment shading and the reflected fragment shading using the following formula (with C the shaded color):

$$C_{SSR} = C_{Frag} + (C_{Reflect} - C_{Frag}) * \beta$$

1) Metalicness: To begin with, let's simply take the metalicness into account (using: $spec_{expo} = 3$, the specular exponent):

$$\beta = Metalicness_{Frag}^{spec_{expo}}$$

2) incorrect reflections: One issue at the moment is that some reflections are simply not correct. If we look at the previous figure, we can see that the reflection on the back wall is not correct as it is using the front of the sphere and not its back. On a symmetric object like a sphere, it is not as much as a problem as on other objects but still, here we see that the lights are not correct as all lights are behind the camera, therefore, the back of the sphere and therefore the reflection should be near pitch dark.

This issue can be solved at the cost of disregarding many reflections. To do so we simply add the opposite of the reflected direction z-component into β. This way, the reflected rays towards the camera (with a negative z-component) are not taken into account, and we get a smooth transition by

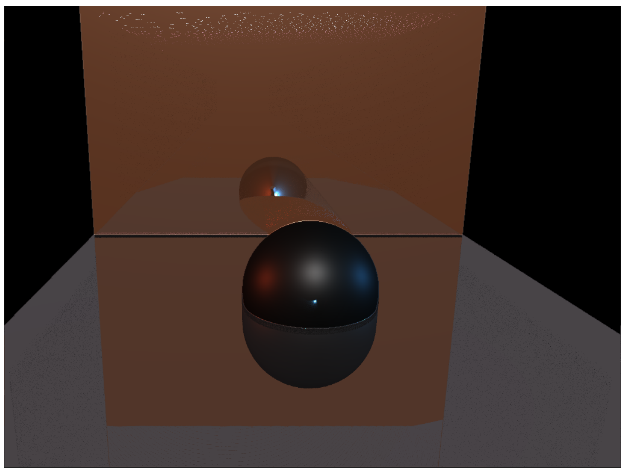

Fig. 3: Adding metalicness *(with values of 0.9 for the floor and 0.5 for the wall)*

multiplying by the z-component even for rays facing the right direction.

$$\beta = -Metalicness_{Frag}^{spec_{expo}} * ray_{reflect}.z$$

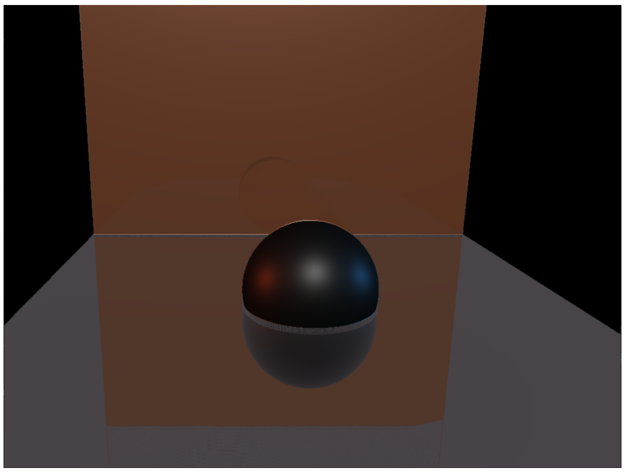

Fig. 4: Removing reflections toward the camera

Hidden reflections could be solved using a multi-depth depth buffer storing the X closest fragments. Of course, this would multiply the complexity by X (in reality even more as a new fragment would now need to be inserted while keeping the depth buffer sorted for this fragment. In the worst case, all X fragments would need to be moved, which means that the complexity would be multiplied by X^2).

3) Edges artifacts: As in the previous section, a major problem of SSR is that what does not appear on the screen can not be reflected. We treated the case of hidden objects behind others but there is also the problem of objects out of the screen. To solve this issue, we introduce an *edge fading factor* e_{fade} to fade away this problem:

$$e_{fade} = clamp(1 - |hit_{UV}.x - 0.5| - |hit_{UV}.y - 0.5|, 0, 1)$$

$$\beta = -Metalicness_{Frag}^{spec_{expo}} * ray_{reflect}.z * e_{fade}$$

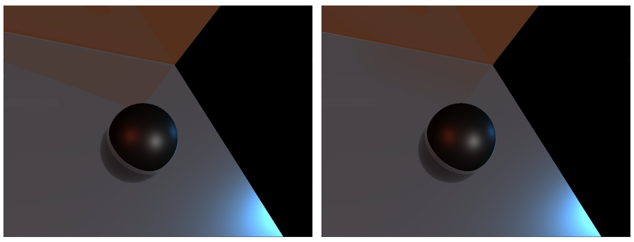

Fig. 5: Without *(left)* and With *(right)* edge fading

D. Improving the quality - Binary Search

With only a few steps in the linear ray march and a large thickness, the rendered reflection tends to make *strips*. This is because due to the high thickness of the ray, the intersection is not computed precisely. to solve this issue, we introduce a **Binary Search** after the linear search to improve the precision of the intersection. Having a large number of steps is still useful as the binary search is only computed when the linear search finds a hit, therefore with few steps in the linear search, we will miss thin objects. However, the binary search improves the rendered results.

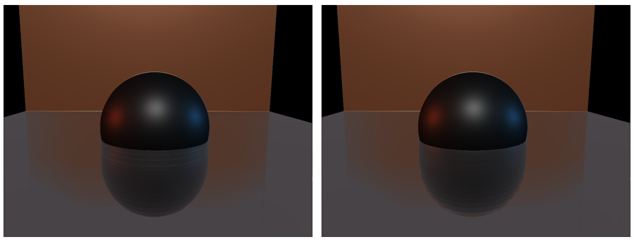

Fig. 6: Withouth *(left)* and With *(right)* binary search with **For this figure:** $SSR_{linearsteps} = 50$ **and** $t_{ray} = 0.1$ **and** $SSR_{binary_steps} = 10$

E. Reducing the noise

In some places, SSR can introduce quite a lot of noise. This can be caused by a bad choice of values for the number of steps and for ray thickness. However, reducing the noise is still important as finding these optimal values can be tricky. One simple approach would be to use a third pass using a simple kernel filter.

Here, we tried to implement an anti-aliasing method directly in the second shader pass. This is usefull as part of the noise is due to aliasing and because by sampling several rays for a given pixel we can check if more than 50% ray hit something removing some noise. The anti-aliasing simply consists of slightly changing randomly the position of the fragment and then averaging the response.

As shown on the figures above, the anti-aliasing method proposed does not improve the result much, leaving a lot of noise while dividing the framerate by 8. *Instead of this method, a third pass should have been used.*

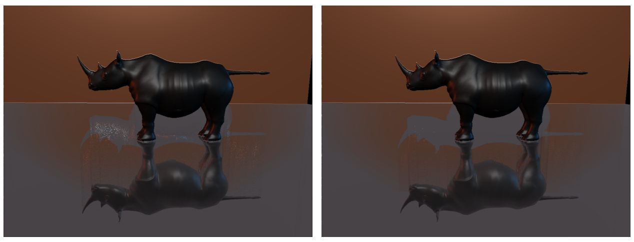

Fig. 7: Without *(left)* and With *(right)* anti-aliasing.
For this figure: $SSR_{linearsteps} = 2500$ and $t_{ray} = 0.01$, With binary search and 8 samples per pixels

III. SSR ISSUES

A. Hidden reflection

In the figure below, the reflection of the rhinoceros is not complete as some elements such as its belly are not visible on the screen.

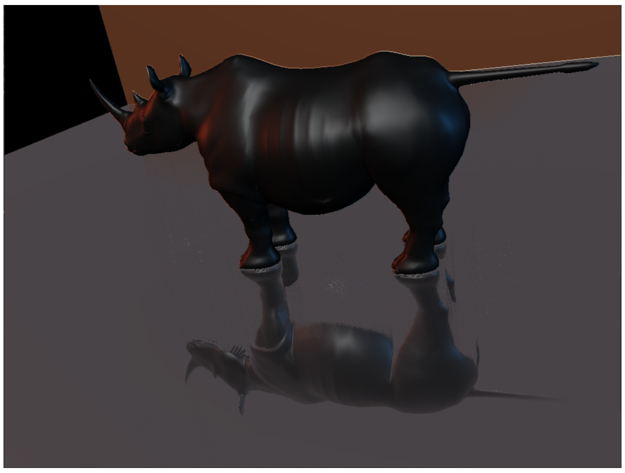

Fig. 8: Illustration of hidden reflections

B. Ghost shadowing

Ghost shadowing is a consequence of hidden reflections: some *fake shadows* can appear. In the figure below, the wall is reflected on the floor but because the rhinoceros hides a part of the wall, in some places on the floor where the wall should be reflected it is not as the rhinoceros was drawn on top of the wall during the first pass.

C. Unrecursive reflections

placing two mirrors in front of each other yields a beautiful result of the object reflected an infinite amount of times smaller and smaller. In other words, the reflection of object A onto B, i reflected onto A, then onto B and so on. With SSR, there is only one step, meaning that the reflection of a given object A onto B will not appear in the reflection onto A.

This can be solved by adding more passes. A solution presented in the paper is to reuse the previous rendering (at the previous frame). Therefore if the camera does not move at each rendering, one level of reflection will be added. However, this can lead to jittering if the camera moves fast.

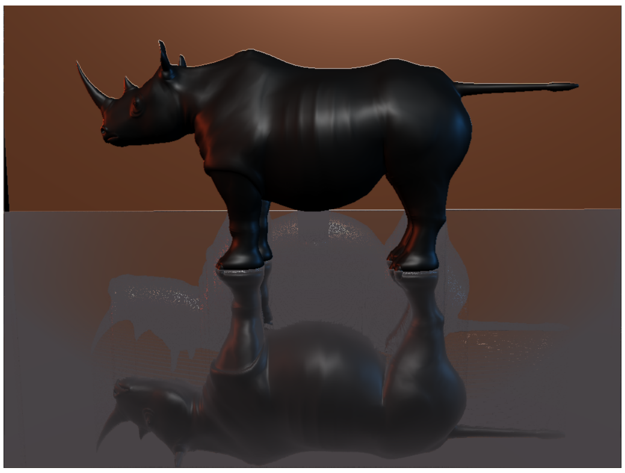

Fig. 9: Illustration of ghost shadowing

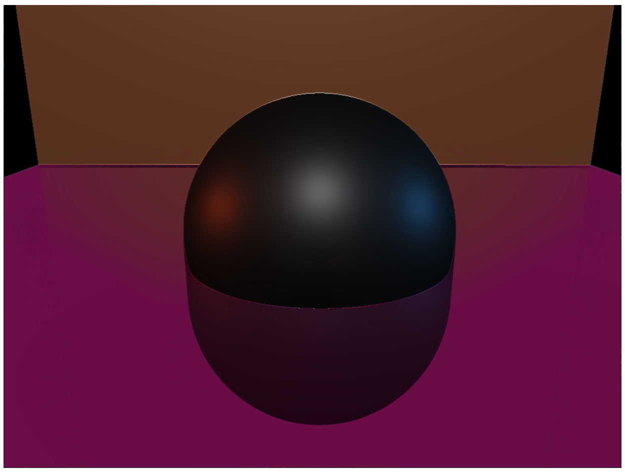

Fig. 10: Illustration of the lack of self-reflections

IV. PERFORMANCES

The performance of SSR highly depends on the parameters (mainly SSR_{linear_steps}) and if we use anti-aliasing. We will use the rhinoceros provided as our benchmark. it is a good model as it suite detailed and has small elements like its tail, which requires a lot of steps in order to display the rhinoceros.

DISCLAIMER: All the following measures were made with a GT1650 and an Intel i7.

With 100 linear steps, 10 binary steps, and a thickness of 0.1, the rhinoceros is pretty well rendered with a frame rate of about 90FPS.

Below, is a figure illustrating the performances as the number of linear steps increases:

FPS in terms of Linear steps							
Steps	25	50	100	250	500	1000	2500
FPS	120	118	91	53	31	18	9
1/FPS	0.008	0.009	0.011	0.019	0.032	0.056	0.111

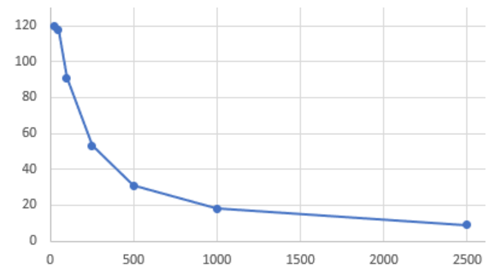

Fig. 11: FPS in terms of the number of linear steps

And if we plot the inverse FPS in terms of the number of steps we check that this is indeed linear, as expected (not for a small number of steps as the frame rate is capped at 120FPS):

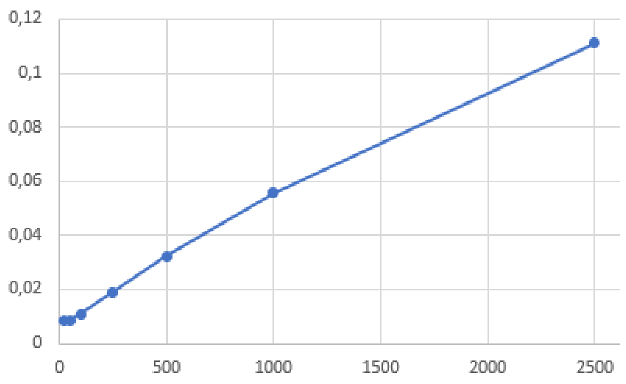

Fig. 12: 1/FPS in terms of the number of linear steps

V. Conclusion

Screen Space reflections is a great way to solve the reflection problem while keeping good performances. However, the process is not as straightforward as it seems; SSR requires fine-tuning some parameters and to then post process in order to get good quality results. Adding a third pass with a temporal filter would greatly improve the results presented in this paper and could be an extension for the future.

Here are a few pictures showing what bad parameters might lead to:

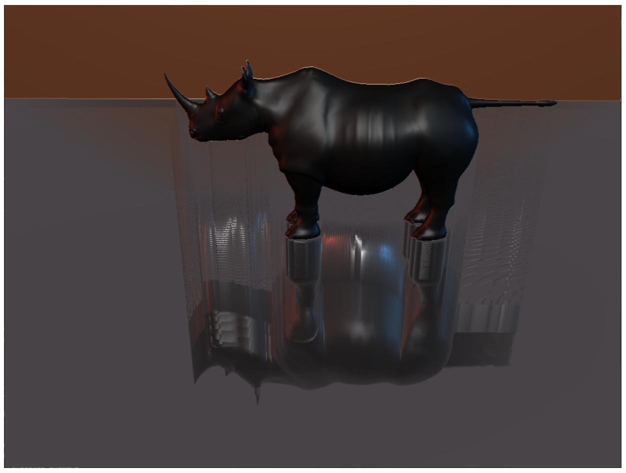

Fig. 13: Thickness too high.
For this figure: $SSR_{linearsteps} = 500$ **and** $t_{ray} = 0.25$

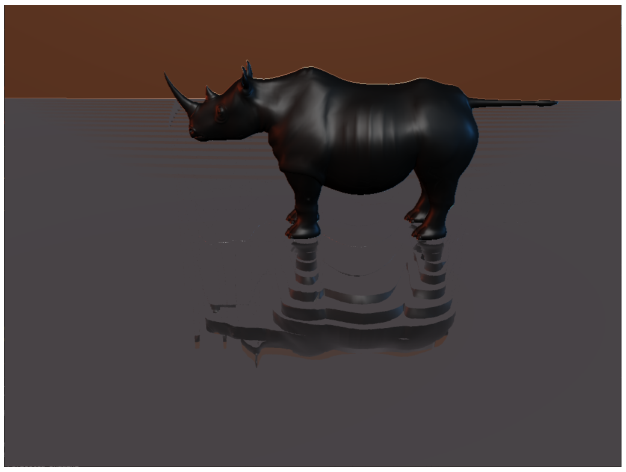

Fig. 14: Not enough steps.
For this figure: $SSR_{linearsteps} = 25$ **and** $t_{ray} = 0.05$