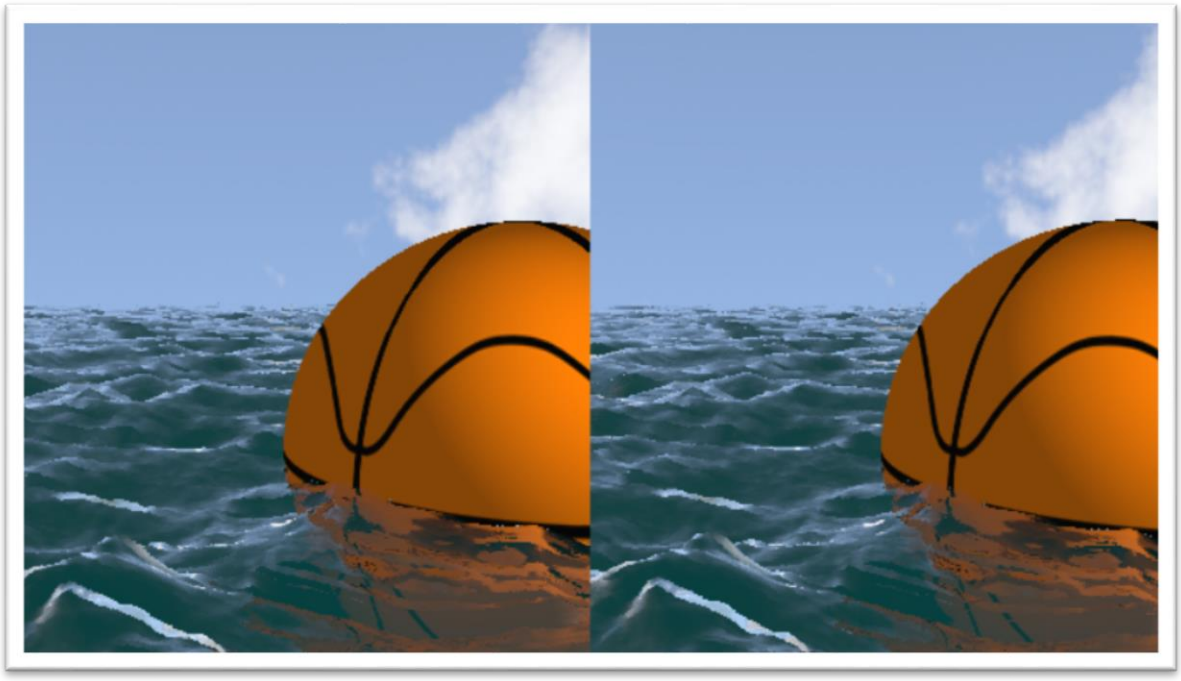


RAY-MARCHING BASED WATER RENDERING

網媒所 r05944045 陳卓晗

網媒所 r05944043 宋焱檳

Abstract

我們本次項目實作的目的是在網頁上用 WebGL 實現對海水的模擬。並且在此基礎上添加 VR 的顯示效果。海水模擬的部分是採用 ridged multi-fractal algorithm 來生成海面，再使用 ray-marching 的方式來渲染屏幕的每一個像素。因為最終還要加上 VR 的效果，所以海水的渲染速度不能太慢，否則觀看的效果會大打折扣。所幸 ray-marching 的實作方式使得即使 VR 打開之後也並不會影響渲染的性能，算是使用這個方式的一個優點。另外，為了讓海水的折射與反射效果更明顯，在渲染海水的基礎上還加入了 skydome texture，以及一個會隨著海浪上下浮動的球。

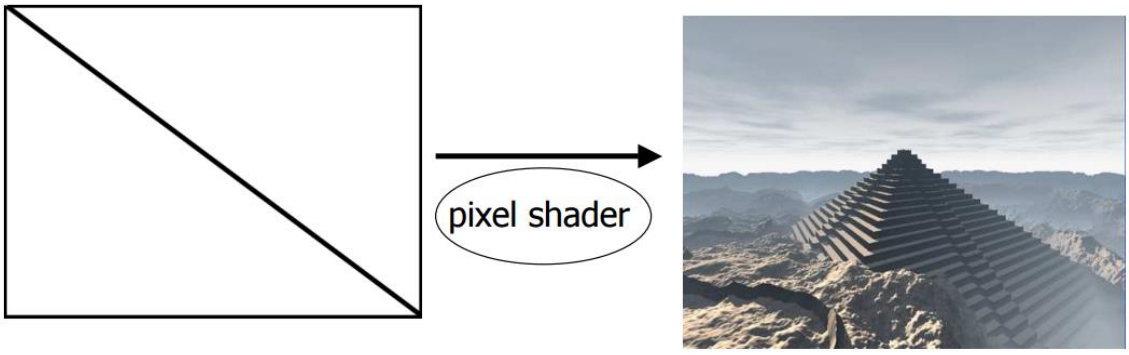
Ray-marching

ray-marching 的渲染方式其實和普通的 ray-tracing 一致，都是從攝像機出發在屏幕上的每個像素產生一條射線，檢測和場景中的物體是否有碰撞。但是 ray-marching 有兩個不同的地方。第一個則是場景的描述不一樣，第二個是每條射線檢測碰撞的機制略有不同。

在一般的 ray-tracing 算法中，場景是事先準備好的一系列的三角形、圓柱體等一般圖形的結合體。渲染過程中會檢測射線和這些物體最近的碰撞點，并在碰撞點上做進一步的計算和渲染。但是 ray-marching 中場景是由算法生成的 procedural image，其中每一個物體都是一個 signed distance function，這個函數返回它代表的物體和場景中某一點的距離。由於場景是由算法生成的，所以它的場景描述可以達到非常小，圖一中的場景由 4kb 的一段程式碼生成。

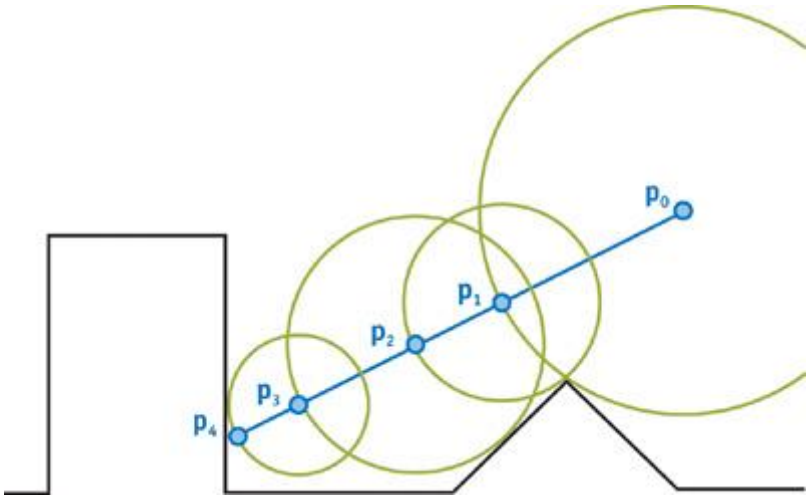
圖一：4kb 程式碼生成的場景

為此，我們先繪製兩塊覆蓋整個屏幕空間的三角形，這樣每個 fragment 都代表屏幕上的一個像素。然後我們將這個場景的描述直接丟進 fragment shader 中，讓每一個 shader 來負責計算射線和場景中具體某個物體的交點，如圖二所示。



圖二：用兩個三角形來繪製場景

檢測交點的方式採用 NVidia GPU Gems 2 – Chapter 8 中所提到的步進式的檢測方法。如圖三，首先假設光線從 p0 點出發，這時候場景中離 p0 最近的點的距離為 r0，於是 p0 往射線方向前進 r0 的距離到達 p1。P1 繼續檢測它和場景中物體最近的交點的距離，然後往前進。直到最後到 p4 時和場景的距離小於一個值，則判斷光線已經和物體有了交點。

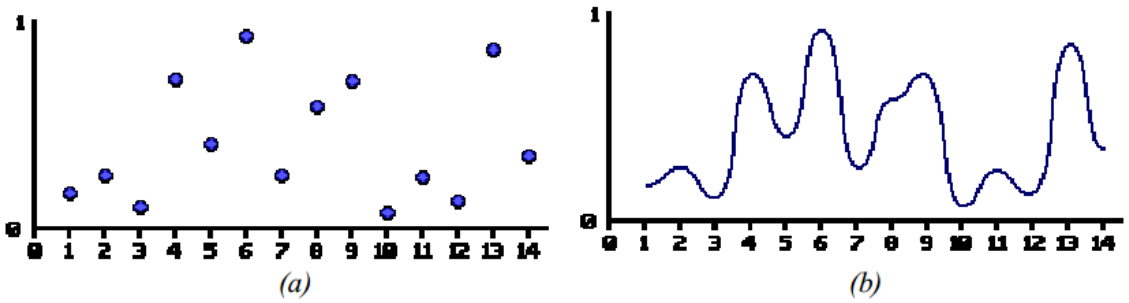


圖三：步進式檢測射線與場景的交點

Water Generation

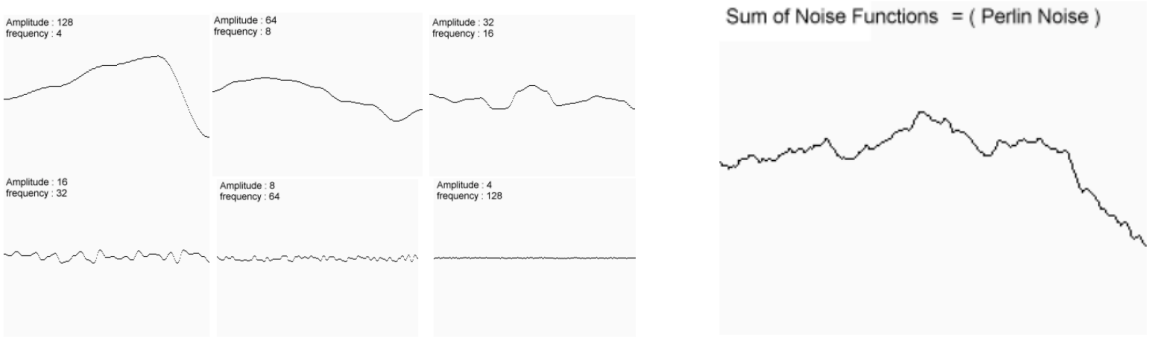
水的生成部分採用的是 ridged multi-fractal 算法。這種算法一般用於生成隨機的地形，在這裡我們用這種“地形”來代表水面。

Fractal 算法的關鍵是用 noise 算法，在某點產生一個隨機的值來代表該點的高度。圖四 b 就是根據 perlin noise 產生的一維的高度圖。



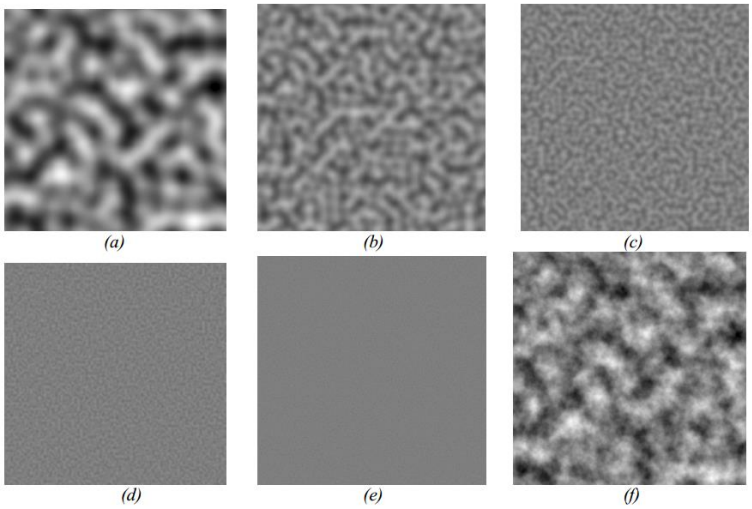
圖四：noise function 生成高度曲線

為了讓曲線不同的縮放程度都有一定的細節，我們通過使用不同的 amplitude 和 frequency 來生成高度值不同的波形，稱為 octave，并將他們疊加來產生在整體上有起伏，並且在不同的細節上也有抖動的形狀更豐富的海水波浪（如圖五）。



圖五：不同的 amplitude 和 frequency 生成曲線之後進行疊加

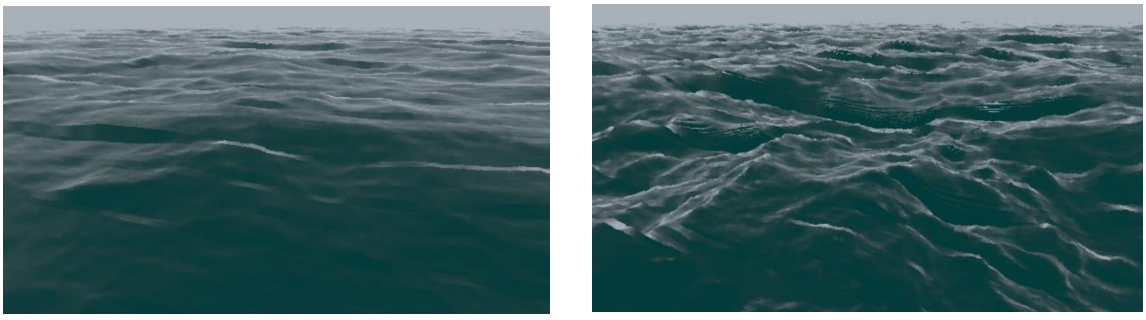
我們將一維的 perlin noise 算法擴展到二維，就能得到類似下圖的結果，圖六 f 是前面圖六 a~e 圖疊加的結果。



圖六：二維的 noise function 產生的結果

但是，普通的 noise 算法依然會產生有規律的圖形。Multi-fractal 算法根據不同的位置增加其他的變數來降低結果的規律性。而 ridged 部分則通過放大計算結果的最大和最小值，讓地形中高的地方更高，低的地方更低。來產生比較狹長的海浪。

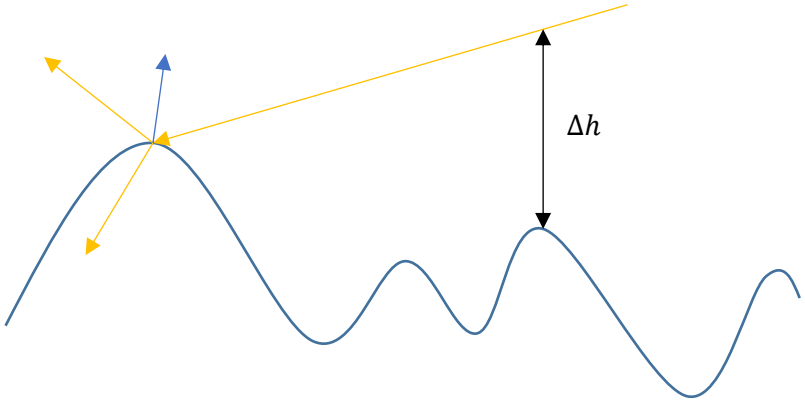
在實作中使用一張 noise 圖片來生成某點位置的高度，通過不同的 amplitude 和 frequency 的疊加可以控制海浪的大小。然後用一個時間變數來控制海水隨時間偏移的程度，這可以用來模擬海水的流動速度。



圖七：不同的 amplitude 和 frequency 得到的海水繪製結果

Water Rendering

有了海水的場景描述之後，我們將其放在 fragment shader 中。接著每個 fragment shader 會射出一條射線來計算它與海平面的交點。如圖八所示，首先計算射線上某一點距離當前海平面的高度差Δh，若高度差大於一定的數值則往前進Δh的距離并再次檢測高度差。直到高度差小於一定的數值，那麼就是射線和海平面的交點。



圖八：計算光線和海平面的交點

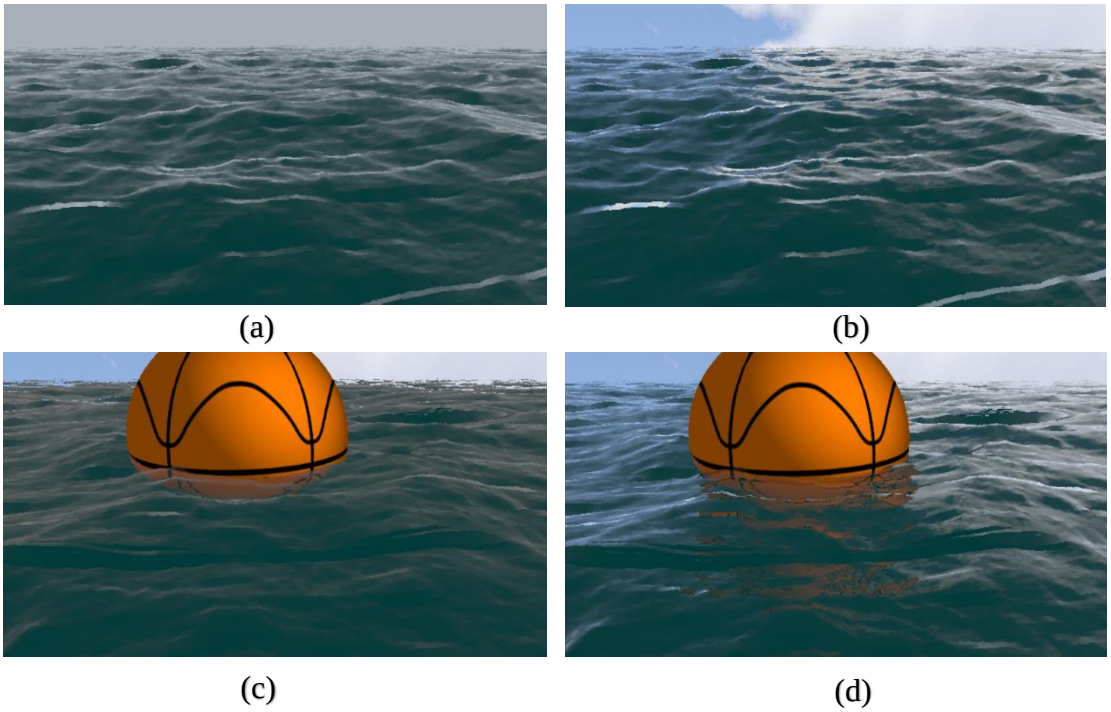
得到和海平面的交點之後，就可以計算該點的法向量，以及光線在該點處的折射光線以及反射光線。

有了法向量、折射光線和反射光線，接下來就可以根據 fresnel equation 來計算折射光線和反射光線所佔的比率，對該點位置水的顏色進行渲染。結果如圖十 a 所示。

為了讓反射光線的顯示效果更明顯，我們又添加了 skydome texture，它是一個如圖九所示的紋理。根據海水表面的法向量，使用 latitude-longitude 對其進行採樣，結果如下圖十 b 所示。

圖九：skydome texture

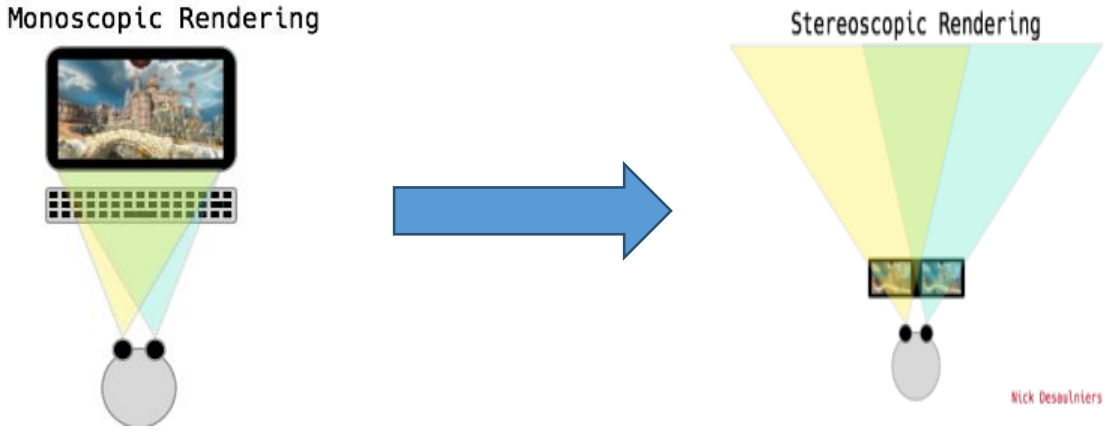
另外，我們也在場景中添加了一個簡單的球體，用於顯示折射光線。球的模型也是用 signed distance function 來描述的，並且設定其會隨著中心位置處海面的高度上下起伏。圖十 c 是只有折射光線時的海水場景。最後全部添加的效果如圖十 d 所示。



圖十：加上不同效果后的水面

VR Implement

在前面的工作中，我們已經完成了 water rendering，現在需要在 webGL 上加入 VR 的功能，讓我們可以在同一場景顯示左右兩個視圖，並且通過 Cardboard 可以觀看到 3D 效果。



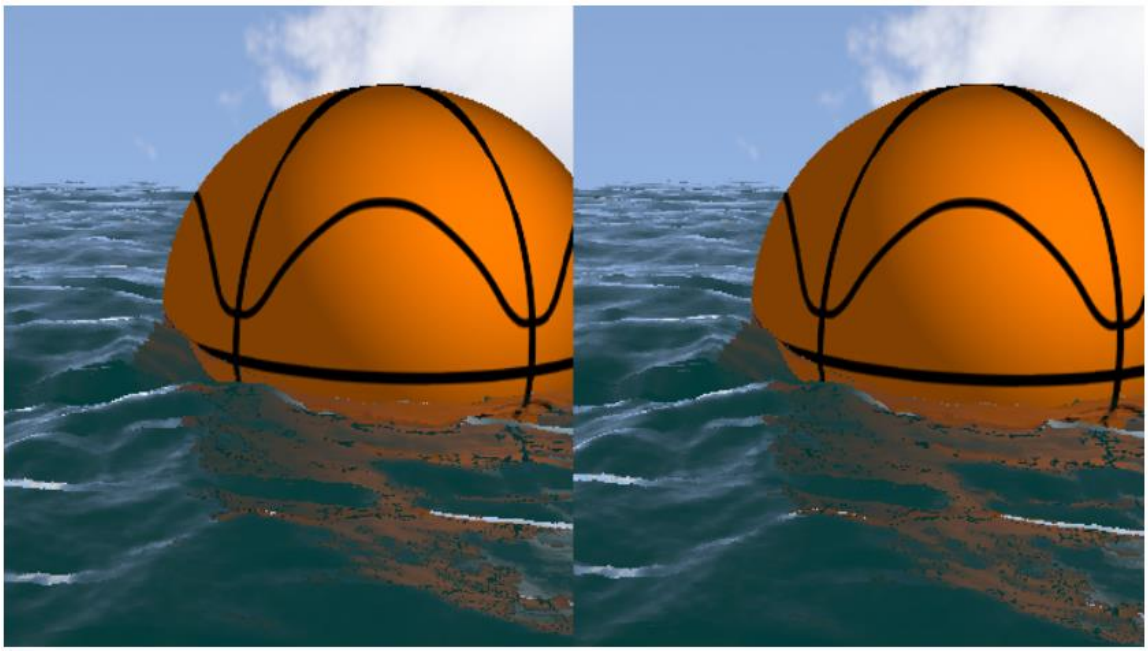
圖十一：monoscopic rendering 和 stereoscopic rendering

從 Monoscopic Rendering 變成 Stereoscopic Rendering 的區別是：傳統觀看模式是通過在顯示器或螢幕上觀看，我們將我們的三維場景展開在一個平面（視口）上。雖然對象可能與視口有不同的距離，但一切都是從單個視點渲染的。我們可能有多個繪製調用來構建我們的場景，但是我們通常使用在場景創建時計算的一個視圖矩陣和一個投影矩陣來渲染所有。現在，當我們將場景從一個視點渲染到頭戴式顯示器（HMD）上時，突然，我們必須從兩個不同的角度渲染兩次。而且，需要考慮瞳距（IPD）的影響。

前面提到，由於我們的實作採取的是 ray-marching 即 ray-casting 的方式，場景是在 fragment shader 中用程式碼描述的，所以我們只需要在計算射線時，對於左邊屏幕和右邊屏幕使用不同的攝像機位置和不同的視覺轉換矩陣即可達到分屏的效果。

Final Result

最終所有的效果全開如下圖所示。使用 cardboard 可以在手機上看出 3d 的效果，但是由於參數的原因，效果難以調節到那麼好，這是後續需要微調的地方。另外，勾選對應的功能可以將其開啟或是關閉。滑動 wind speed 滑塊可以調節風速的大小，控制水流的速度和浪花的高度。



圖十二：最終效果圖

Future work

由於海水完全是由程式生成的，海水的形狀參數不容易調節，導致在幅度變大的時候會出現不真實的抖動。另外，受限于使用這種方式生成場景，也較難添加其他形狀的物體。比如說如果要模擬浪花的形狀就比較難實現。

雖然場景比較簡單，但是在添加了水球之後，由於從屏幕出發的光線和折射光線需要再多做一次碰撞檢測的計算，極大地增加了運算量。也許可以用更好的碰撞檢測的演算法。

Reference

[1] Ondrej Linda. *Generation of Planetary Models by Means of Fractal Algorithms.*

[2] E. Darles, B. Crespin, D. Ghazanfarpour. *Accelerating and Enhancing Rendering of Realistic Ocean Scenes.*

[3] William Donnelly. *GPU Gems 2, Chapter 8. Per-pixel Displacement Mapping with Distance Functions.*

[4] *Scratchpixel, Reflection, Refraction and Fresnel.*

[5] Jamie Wong. *Ray Marching and Signed Distance Functions.*

[6] Inigo Quilez. *Rendering Worlds with Two Triangles with Ray-tracing on the GPU.*

[7] Brian Danchilla. *Beginning WebGL for HTML5.*