TRABALHO FINAL DE GRUPO

Técnicas Avançadas da Programação 3D

Diogo Veloso nº21097

José Alves nº21081

José Monteiro Nº21083

15 DE JUNHO DE 2023

IPCA

Conteúdo

TRABALHO DE GRUPO

INTRODUÇÃO

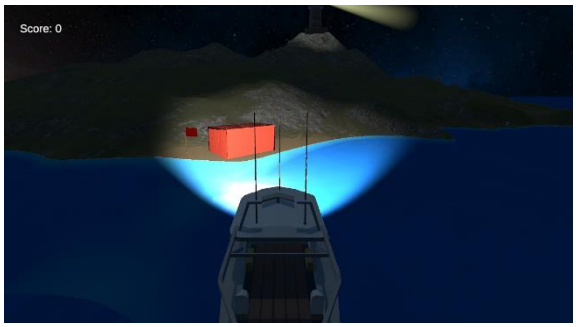
Para estre trabalho decidimos criar um pequeno jogo de entrega de encomendas no mar.

O jogo tem um total de 5 ilhas, e cada ilha tem um local de recolha e de entrega de encomendas.

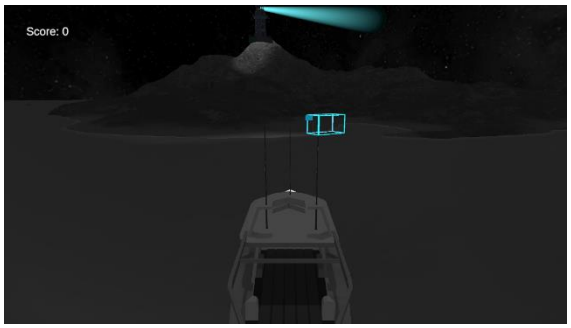
O objetivo do jogo é ir buscar encomendas em um dos pontos de recolha e levá-lo num ponto de entrega. Estes pontos são escolhidos aleatoriamente. Apos isso repete-se o ciclo.

Existem também certos obstáculos e um sistema de visão Highlight que vai deixar em foco o ponto onde temos de nos dirigir.

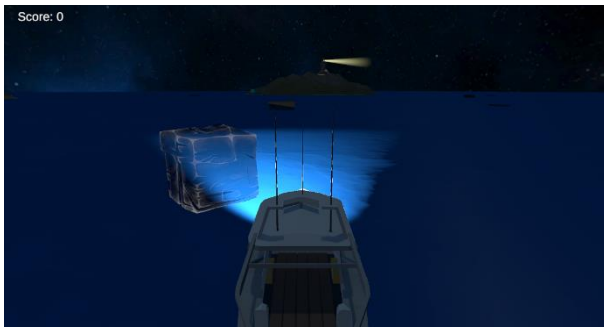
Para fazer isto tivemos de criar diversos shaders para cobrir alguns aspetos do jogo.



Ponto de recolha



Ponto de entrega/Visão Highlight



Obstáculo

1. SHADERS GERAIS

1.1 : SHADER DO OCEANO

Um dos shaders criados foi o shader do oceano.

Este shader foi o primeiro a ser criado e foi a base do jogo, e terá pelo menos duas funcionalidades:

- Servirá de "base" onde o barco irá se mover;
- Servirá para simular as físicas de "floaters" que existem no mar.

Para fazer este shader começamos por fazer as ondes do oceano, para isso tivemos que mexer nos vértices do nosso objeto, neste caso, será um plano.

Para fazer isso eu poderia ter optado por usar vertex e fragment shader, mas acabamos por optar por usar surface shader.

Usamos surface shader porque acabaria por tornar o processo de criação da textura em si muito mais fácil, ou seja, temos acesso a um leque muito maior de variáveis de uma textura, como por exemplo a Smoothness do material, de uma forma mais fácil e rápida.

Também optamos por esta maneira porque é possível mexer nos vértices através de surface shader, caso contrário este shader teria de ser feito todo em vertex e fragment shader o que iria tornar o processo muito mais complicado e complexo.

Para fazer as ondes iremos usar a Fórmula de Gerstner Waves.

Enquanto uma sin wave acaba por ser uma onda bastante simples e constante, uma Gerstner wave conseguimos fazer uma onda mais customizada e mais voltada para o realismo.

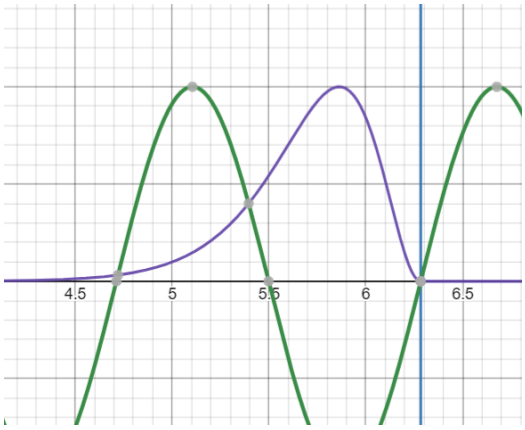


Figura 1: Diferença entre sin wave e Gerstner Waves

Com este tipo de waves conseguimos controlar vários aspetos como por exemplo:

- Amplitude das ondas;
- A velocidade das ondas;
- O tamanho das ondas (espaçamento entre as várias ondas);
- Direção das ondas.

```
void vert(inout appdata_full vertexData,out Input o) {

    UNITY_INITIALIZE_OUTPUT(Input,o);
  float k = (2 * PI) / (_Wavelengh);
  float speed = sqrt(9.8 / k) * _Speed;
  float2 d = normalize(_Direction);

  float2 dotp = dot(d,vertexData.vertex.xz);

  float f = k * (dotp - speed * _Time.y );

  float a = _Steepness / k;


  if(_toggleWaves == 1)
  {
  vertexData.vertex.y = (a * sin(f));
  vertexData.vertex.x += (d.x * (a * cos(f)));
  vertexData.vertex.z +=  (d.y * (a * cos(f)));

o.localPos = vertexData.vertex.xyz;

  }
}
```

Este tipo de waves irá nos fazer alterar todos os valores dos vértices, ou seja, iremos alterar tanto no X como no Y e no Z.

Primeiro começamos por calcular a o tamanho da das ondas em radianos.

Depois disso vamos calcular a velocidade da onde e normalizar a direção delas.

Também teremos de calcular a amplitude das ondas, para isso iremos dividir a Steepness, uma variável criada nas propriedades, pelo tamanho das ondas que calculamos.

Também teremos de calcular o produto escalar entre a direção da onde com os vértices de x e z.

Depois é só aplicar a fórmula para cada um dos eixos.

Equation 9

$$\mathbf{P}(x,y,t) = \begin{pmatrix} x + \sum\left(Q_i A_i \times \mathbf{D}_i.x \times \cos\left(w_i \mathbf{D}_i \cdot (x,y) + \varphi_i t\right)\right), \\ y + \sum\left(Q_i A_i \times \mathbf{D}_i.y \times \cos\left(w_i \mathbf{D}_i \cdot (x,y) + \varphi_i t\right)\right), \\ \sum\left(A_i \sin\left(w_i \mathbf{D}_i \cdot (x,y) + \varphi_i t\right)\right) \end{pmatrix}.$$

Após isso, teremos que trabalhar na textura em si.

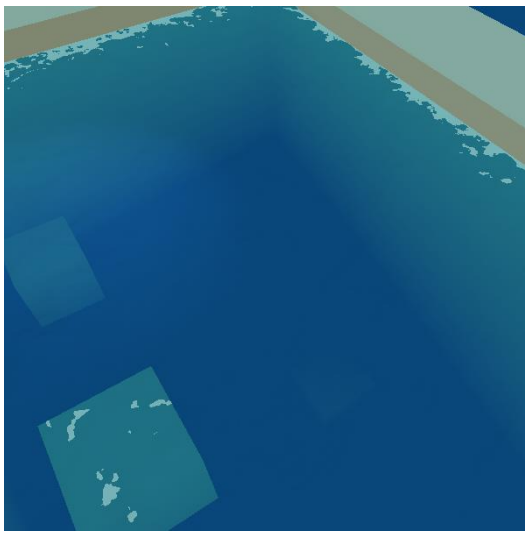
Primeiro, colocamos o shader como transparente, porque a água por si só é transparente.

Para isso, iremos usar a _CameraDepthTexture, que nos irá dar a distância da camara até aos objetos, quanto mais longe, mais escura ficara a textura.

Precisamos de extrair isso para dentro de uma textura, mas ao guardar essa textura numa variável teremos que usar o tex2Dproj () , que nos irá dar a profundidade da agua entre os valores de 0 e 1.

Para termos a profundidade a água em relação a camara teremos de calcular a diferença entre o resultado obtido da variável guardada em cima com a posição do ecrã em profundidade, através do W da posição do ecrã.

Com isto iremos ter uma textura em que quanto maior for a profundidade mais clara será a cor, a cor será branca.



Após isso, iremos fazer com que os valores fiquem entre 0 e 1 e vamos dar lerp de duas cores, sendo uma delas mais escura que a outra.

Iremos usar um azul mais escuro para simular a profundidade do oceano e um azul mais claro para simular as zonas menos profundas. Como estamos a usar um lerp, o shader vai fazendo uma “transição” entre essas duas cores consoante o valor da profundida.

Após isso, tivemos de criar as bordas da água.

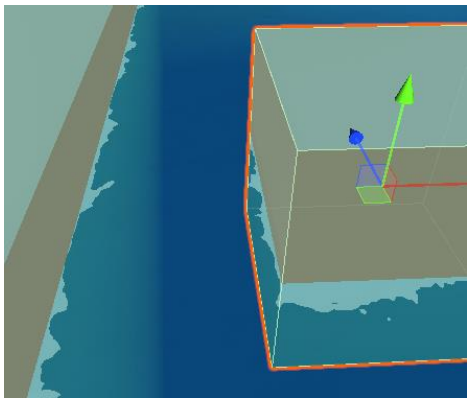
Primeiro teremos de arranjar um noise texture para ser usada. Iremos usar essa textura para criar as bordas e também para colocar uma textura para simular o pico da onda.

Para isso primeiro teremos de extrair o r da textura, que será o channel red, que irá armazenar a intensidade do “caos” do noise texture. Este valor da intensidade pode ser obtido por qualquer um dos canais seja r,g ou b.

Depois iremos fazer com que essa quantidade seja o menor, para isso teremos de criar uma variável para pedir essa intensidade ao utilizar, teremos que criar um range de 0 a 1, porque será o valor que irá ser guardado.

Depois teremos que uma variável para armazenar a distância de Foam que queremos utilizar, este valor irá ser personalizável, ou seja, iremos criar uma variável nova dentro das propriedades.

Para conseguirmos ter as bordas da textura teremos que ir buscar acima o valor da profundidade e depois dividir esse valor pela distância de Foam que nos queremos, assim iremos ter a "real" distância de Foam na textura.



Depois disso, teremos que remover a parte preta da textura, caso contrário, o efeito irá ficar estranho.

Por fim, iremos adicionar esta o resultado obtido ao albedo da textura, mas antes iremos multiplicar esse valor pela cor que nós queremos associar ao Foam.

Para termos pequenos detalhes no ponto máximo das ondas, acabamos por utilizar exatamente o mesmo código para criar as bordas do oceano, mas com algumas alterações:

```
if(max(0,IN.localPos.y))
{
float surfaceNoiseSampleF = tex2D(_FoamTexture, IN.uv_FoamTexture).r;
float surfaceNoiseCutofffF = saturate(_WaveFoamDistance / _WaveFoamDensity);
float surfaceNoiseF = surfaceNoiseSampleF > surfaceNoiseCutofffF ? 1 : 0;
o.Albedo += _FoamColor * surfaceNoiseF;

}
```

- Criar uma nova variável para controlar a intensidade de "caos" presente na textura nesse instante;
- Criar uma nova variável para controlar a distância/tamanho do Foam no ponto máximo da onda;

No caso deste shader, o ponto máximo dá-se sempre que o valor de y for maior que 0.

Mas para isso eu preciso de ir buscar a informação dos vértices a cada instante para usar dentro do surface shader.

Para isso basta apenas criar uma variável nova para armazenar a posição local de cada vértice, e igualar esses valores dentro da parte de mexer nos vértices do plano.

```
o.localPos = vertexData.vertex.xyz;
```

Esta foi uma das partes mais demoradas de se fazer no shader, visto que não sabíamos como é que iriamos passar os vértices do vert para surface.

Para acabar o shader, decidimos animar-lo.

Para isso simplesmente multiplicamos as UV´s da textura noise pelo Time dentro do shader, e associamos essas novas UV´s à respetiva textura.

Para adicionar mais um bocado de detalhe ao shader, decidimos color um Normal map e criar variáveis para mexer com a Smootheness da textura da água.

Para criar este shader, também pensamos em usar vertex e fragment shader, em que a parte das ondas estava a funcionar direito, o "problema" seria na parte do fragmanent shader, que iria fazer com que o shader fica-se demasiado complexo para ser mexido, porque em surface shader, nos temos acesso a um leque maior de opções na parte da textura e mais fácil.

Para termos os mesmos efeitos a serem criados na textura, seria uma complicação demasiado grande para o trabalho/projeto que é, tornando assim muito mais complicado e demorado de se mexer no shader.

1.2 : SHADER DAS ENCOMENDAS

Este shader será usado para quando o barco apanha as encomendas a caixa dissolver.

Para este shader decidimos fazer com que a caixa expanda e dê dissolve ao mesmo tempo, para criar esse efeito tivemos que:

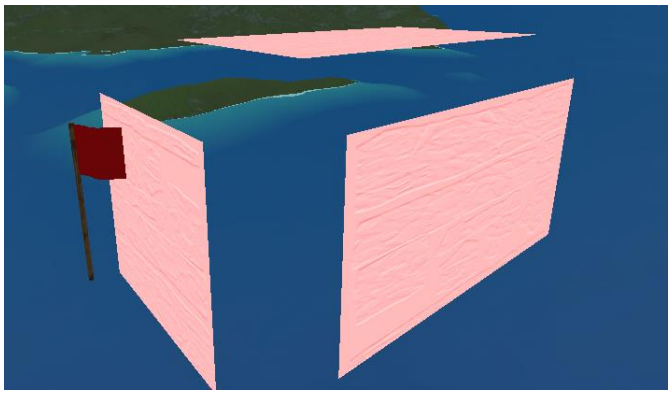
- Mexer nos vértices de mode a que estes expandissem;
- Criar um efeito de dissolve na textura do shader.

Começamos primeiro por alterar a posição dos vértices para criar o efeito de expandir a caixa.

Para isso precisamos de criar uma nova variável nas propriedades para controlar esse efeito.

Conseguimos obter este efeito através das normais do objeto, o objetivo seria expandir os vertíveis do objeto na direção de cada normal.

Primeiro precisamos de ir buscar as normais do objeto, para isso precisamos de as pedir dentro da struct do appdata e do v2f.



Após isso, iremos multiplicar as normais pela variável que criamos para controlar o extrude, por fim, teremos de somar o resultado dessa multiplicação por todos os vértices do modelo.

Depois fizemos a parte de dissolver a textura.

Para começar precisamos de associar um noise texture para servir de base para dissolver.

Iremos precisar de ir buscar qualquer um dos valores dos canais desta textura, para conseguirmos fazer desaparecer a textura.

Para criar o efeito de desaparecer, teremos de dar discard de todos os valores da textura que sejam menores que o valor da nossa variável da “intensidade” do dissolve. Estes valores iram estar entre 0 e 1, por isso, quanto maior for a nossa variável maior será a quantidade de dissolve que irá acontecer.

Por fim, acabamos por criar uma pequena borda a volta do efeito de dissolve.

Para isso, iremos precisar de uma nova variável para controlar o tamanho dessa borda e outra para controlar a corda dessa mesma borda.

Após isso, teremos de fazer um lerp entre a textura do objeto com a cor do dissolve que queremos.

Assim, teremos um shader onde podemos dar extrude das faces e fazer dissolve delas.

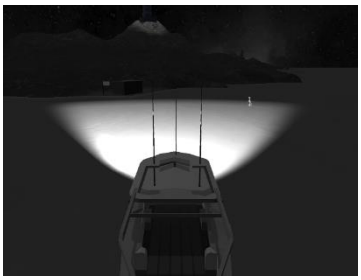
1.3: SHADERS HIGHLIGHT

Durante o jogo existe um sistema de Highlight. Ao pressionar a tecla E o jogo fica em tons de preto e branco com exceção de objetos a sinalizar o objetivo. Relacionado a este sistema existem vários shaders associados**. Este sistema vai ser chamado de visão Highlight durante este relatório.**

1.4: POST PROCESSING HIGHLIGHT

Para criar o efeito de Highlight mencionado nos usamos um Post Processing. Este é a parte mais importante do sistema de Highlight. Ao usarmos Post Processing a imagem da camara vai se transformar numa textura que vai cobrir a camara. Isto por si só não vai alterar a imagem, mas vai nos permitir alterar essa textura para mudar o resultado final da imagem renderizada.

1.4.1: PRETO E BRANCO



Para criar este, inicialmente criamos uma mistura dos três canais de cor da imagem e usamos isso como cor, colocando em todos os canais. Isto vai transformar a imagem em tons de cinza.

1.4.2: RENDERIZAÇAO DE OBJETOS HIGHLIGHT

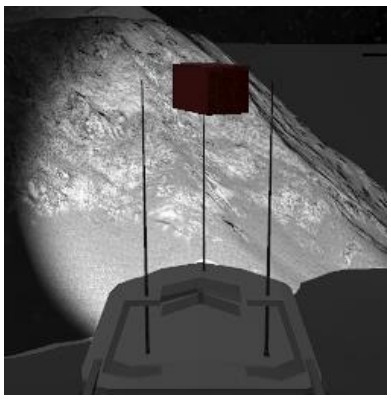
O próximo passo é separar os objetos que pretendemos colocar como Highlight.

Para isto inicialmente pensamos em usar um stencil para separar os objetos, mas rapidamente mudamos de ideias. Isto foi decidido pois o Stencil além de ser relativamente inconsistente com certas coisas como a distancia a camara, também vem com o problema que para qualquer objeto que eu queira usar como Highlight ter de criar um Stencil dentro do material.

Por causa destas desvantagens decidimos arranjar uma outra solução – uma segunda camara.

As camaras do Unity permitem-nos escolher que Layers vão ser renderizadas. Isto é muito útil porque nos deixa escolher os objetos Highlight simplesmente mudando a Layer. Ele permite também escolher como o fundo do render vai ser. Por definição ele renderiza a skybox, mas para esta segunda camara decidimos renderizar a cor preta com o seu alpha a 0. Este passo é importante para depois.

A partir deste podemos criar uma Render Texture e enviare-la ao material usado no Post Processing da nossa camara principal. Lá nos podemos filtrar a parte dessa Render Texture que vai ser renderizada na camara principal através do alpha da cor do fundo, o qual tínhamos colocado a 0. Isto vai fazer com que apenas os objetos com a Layer Highlight não sejam renderizados a preto e branco.



É relevante dizer também que estes vão ficar a frentes de qualquer outro objeto na visão da camara. Isto acontece porque a nossa segunda camara não tem a informação de possíveis objetos que estejam a frente do nosso objeto Highlight. Felizmente isto acaba por ser útil para o nosso jogo, permitindo-nos ver sempre os Highlights.

A troca entre visões é feita em código de C# no script Post Processing e é ativado com a tecla E. Qualquer outra troca de objetos para a Layers Highlights também é feita através de scripts.

1.5: WIREFRAME SHADER

Este shader é usado nos pontos de recolha e de entrega em cada ilha.

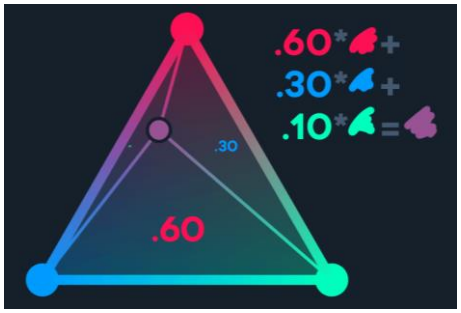
Este só esta ativado em visão Highlight e só para o objetivo atual. Quando nenhuma destas condições se aplicam apenas usa um material básico.

Para fazer este shader eu precisei usar um conceito chamado de coordenadas baricêntricas.

1.5.1: COORDENADAS BARICENTRICAS

Assumindo que temos um triangulo, qualquer ponto dentro desse triangulo vai dividir esse triangulo em três, cada um com a sua área. Estas áreas somadas vão dar a área do triangulo principal.

Quanto mais perto o ponto está de um dos segmentos de reta do triangulo menor a área do triangulo que vai ser formado pela junção desse ponto com os pontos do segmento de reta. A partir disto podemos ter uma noção da distância de um ponto a uma das arestas.



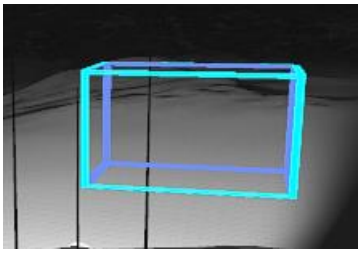
1.5.2: DESENHAR WIREFRAME

Com o conceito que acabamos de observar podemos criar um sistema de Wireframe facilmente. Isto é possível porque o Unity por definição renderiza modelo em triângulos.

A partir dai comparamos qual das três áreas formadas por um ponto é menor, para saber qual a aresta mais próxima, e se essa área for menor do que a nossa largura do Wireframe renderizamos-o com a cor desejada. Os outros pontos que não passam esta condição são descartados.

Outro passo necessário é um segundo Pass. Este segundo Pass é necessário para renderizar a parte de fora do nosso objeto para melhor representar o Wireframe do modelo enquanto o primeiro apenas renderiza a parte de dentro. A ordem é irrelevante.

Por fim adiciona-se um vetor param as coordenadas baricêntricas, que por definição são constantes, assim alterando os cálculos, pra remover as diagonais.



Cada cor representa um Pass

1.6: LIGHTHOUSE SHADER

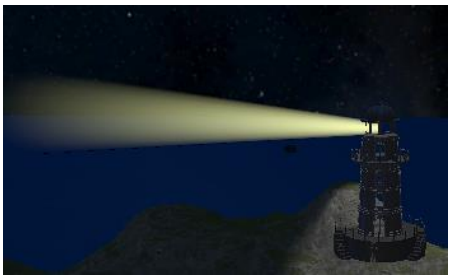
Outro shader que vai ser afetado pela visão Highlight é o da luz dos faróis, que se situam em cima de cada ilha. Estes só estão ligados quando o objetivo é na ilha me questão. ~

1.6.1: CRIAR A LUZ

Para criar esta luz não podemos simplesmente adicionar uma luz Spotlight. Isto porque uma luz por si só não faz nada, para ela ser visível ela tem de estar em contacto com outros objetos. Então decidimos optar por outra estratégia.

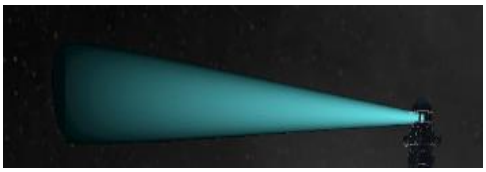
Depois de alguma pesquisa decidimos usar um modelo em forma de cone, e mudar o material dele para simular uma luz. Para isso inicialmente demos-lhe uma cor e reduzimos a sua opacidade. Apos isso metemos a sua Render Queue a Transparent para renderizar a sua transparência sem problemas.

1.6.2: MUDAR A OPACIDADE



No final adicionamos um efeito de Fresnel para diminuir a intensidade nas extremidades da luz através do alpha, e também fizemos um efeito similar nas UVs em y, sendo que a medida que se afasta da origem também vai diminuindo a intensidade.

No entanto este último passo gerou um problema onde este efeito não é visível ao entrar em visão Highlight. Isto acontece porque ao colocar a Render Queue como Transparent, a Render Texture da nossa segunda camara vai assumir o alpha da nossa luz como 0. Isto vai entrar em conflito com a maneira como estávamos a detetar se um objeto Highlight devia ser renderizado ou não na camara principal. Isto forçou-nos a mudar essa condição para filtrar fora qualquer instância da cor preta ou invés de usar o alpha.



Só que isto também gerou o problema mencionado, onde os efeitos de opacidade não são aplicados por causa de não usar o alpha na comparação. No final decidimos deixar assim já que a diferença não é muito visível.

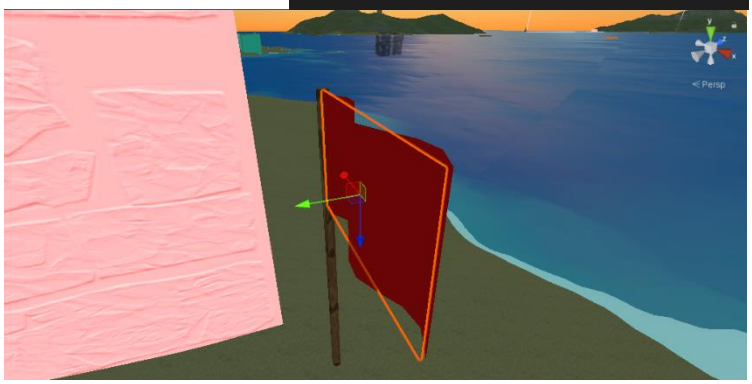
1.7: SHADER DA BANDEIRA

Durante a exploração do mapa o jogador vai encontrar bandeiras em cada posto onde ele vai buscar ou entregar uma encomenda, essas bandeiras possuem um UnlitShader em que esse shader fazemos um simples vertex Displacement.

```
v2f vert (appdata v)
{
    v2f o;
    o.uv=v.uv;
    if(o.uv.x>0.1)
    {
     o.uv.x=sin(o.uv.x*10+_Time.w);
    }
    float3 vert = v.vertex;
    vert.y=o.uv.x;
    o.vertex = UnityObjectToClipPos(vert);
    return o;
}

fixed4 _Color;

fixed4 frag (v2f i) : SV_Target
{
    // sample the texture
    fixed4 col = _Color;


    return col;
}
ENDCG
```

Este Vertex Displacement e feito usando os uvs na direçao do x (o.uv.x) e iguala-lo a uma função sin, em que dentro desta função de sin ponho os valores necessários para que esta pareça que esta a esvoaçar com o vento, sendo estes o próprio o.uv.x vezes um valor neste caso escolhi 10 e por ultimo soma-se o _time.w em que este e que vai fazer com a bandeira se mexa continuadamente porque _time e o tempo do começo da aplicação ou o quando começou a ser renderizado, e pus _time.w porque o entre _time.x,_time.y,time.z e _time.w ,o _time.w e o mais rápido(t/20 , t , t*2 , t*3).

Desseguida guardei o v.vertex dentro de um float3 chamado vert, sendo por ultimo necessário igualar o vert.y ao o.uv.x modificado pela função de sin, fazendo assim com que a bandeira esteja a fazer ondulações na direção do x com os vértices a ir na direção dos y, também se fez um if para indicar onde começa as ondulações da bandeira.



1.8: SHADER DOS OBSTACULOS

O objetivo deste shader era fazer com que o barco tivesse tipo um campo de visão só consegue-se ver os obstáculos a uma certa distancia para isto acontecer, usei um Unlit shader que neste shader passei através float3 (_position_boat) a posição do barco sendo preciso recorrer a um script em que nesse script utilizei o método SetGlobalVector para que a posição do barco fosse posta em qualquer shader que possui a variável(_position_boat).

```
void Update()
{
    Shader.SetGlobalVector("_position_boat",transform.position);
```

Também e preciso ter a posição do objeto em si que ira desaparecer e aparecer consoante a distancia ao barco, por isso dentro do Vert passa-mos o objeto para worldPos através do método mul(que multiplica matrizes por vetores, matrizes entre matrizes) entre o unity_ObjectToWorld e v.vertex, sendo assim depois possível usar a posição do obstáculo no mundo.

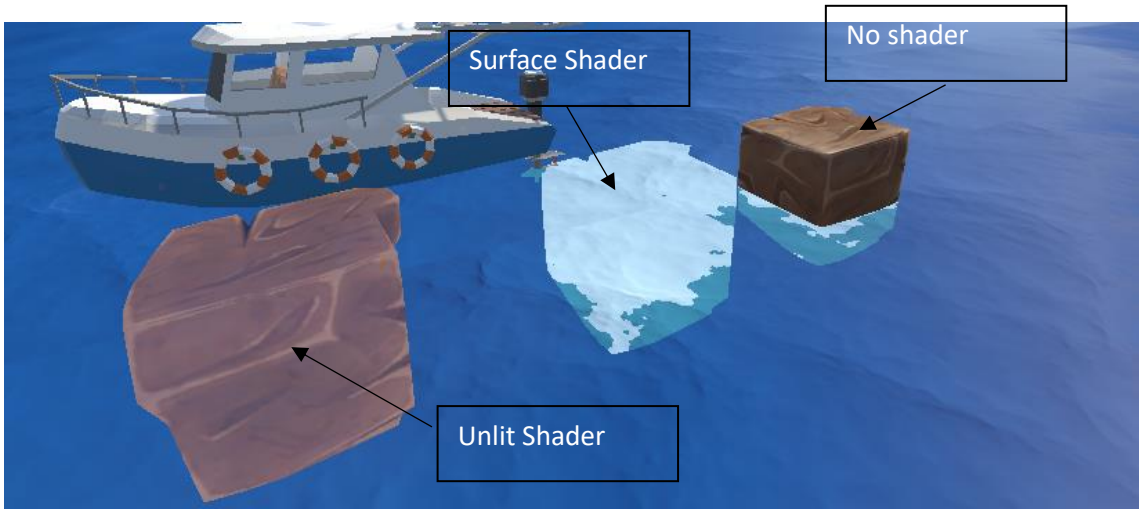
De seguida no frag, vai se aplicar os cálculos para que a pedra desapareça consoante a distancia do barco, para isso primeiro calcula-se a distancia entre o obstáculo e o barco com o método

distance(da o valor da distancia entre dois objetos) guardando este valor dentro de um float inserido na struct v2f,depois divido esse valor pelo um slider para que eu possa customizar a distancia "máxima e mínima" onde o objeto na distancia mínima e suposto estar invisível e na máxima totalmente visível, de seguida uso o método Saturate para fazer que os valores fiquem limitados entre 0 e 1, dentro do método Saturate meto a variável que esta a guardar o valor dado da distancia e faço um menos esse valor, porque se no saturate aplica-se simplesmente a variável iria fazer com que desse o contrario do que e pretendido, sendo o pretendido fazer que quando o barco esta perto do obstáculo este o conseguir ver.

```
        struct v2f
        {
            float2 uv : TEXCOORD0;
            float4 vertex : SV_POSITION;
            float render:TEXCOORD1;
            float3 wpos:TEXCOORD2;
        };

        sampler2D _MainTex;
        float4 _MainTex_ST;
        float3 _position_boat;
        float  _slider;
       //  float _Alpha;

        v2f vert (appdata v)
        {
            v2f o;
            o.wpos=mul(unity_ObjectToWorld,v.vertex);
            o.vertex = UnityObjectToClipPos(v.vertex);
            o.uv = TRANSFORM_TEX(v.uv, _MainTex);
            return o;
        }

        fixed4 frag (v2f i) : SV_Target
        {

           fixed4 col =tex2D(_MainTex,i.uv);
            i.render=distance(i.wpos.xyz,_position_boat.xyz);
            i.render/=_slider;
            i.render=saturate(1-i.render);
            col.a=i.render;
            return col;
        }
        ENDCG
    }
  }
}
```

Para finalizar ao Alpha da textura principal(col.a) iguala-se o valor dado pelo os cálculos anteriormente feitos.

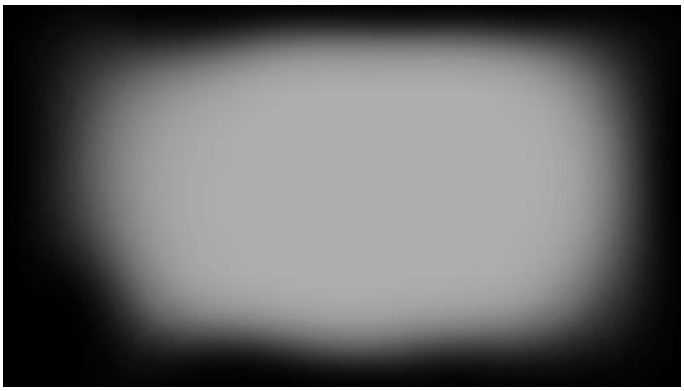
Neste shader apos aplicá-lo existe um problema que ao por o objeto dentro da agua criada ele não se consegue notar que esta submerso, isto por ser um unlit shader, por isso eu ao reparar nisto tentei replicar este shader no surface shader, só que este não deu que era pretendido, acabando assim por ficar o unlit shader.

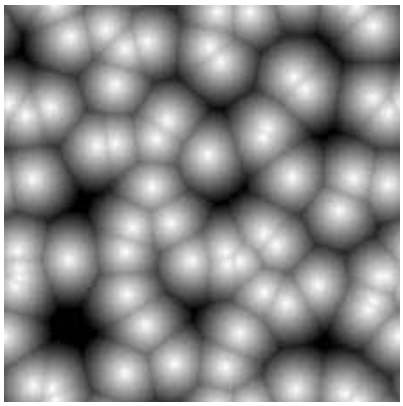


1.8: SHADER DAS COLISOES-POST PROCESING

Com este shader pretende-se aplicar um efeito quando o barco colide contra os obstáculos presentes no mapa, aparecendo no momento da colisão uma distorção na camara e a visão fica afunilada sendo que ao fim de um tempo definido este efeito ira desaparecer aos poucos.

Para fazer este efeito primeiro e preciso ter uma textura em que o Alpha dela desce constantemente, para fazer o efeito de a camara estar afunilada e também iremos precisar de um noise texture para fazer a distorção da camara



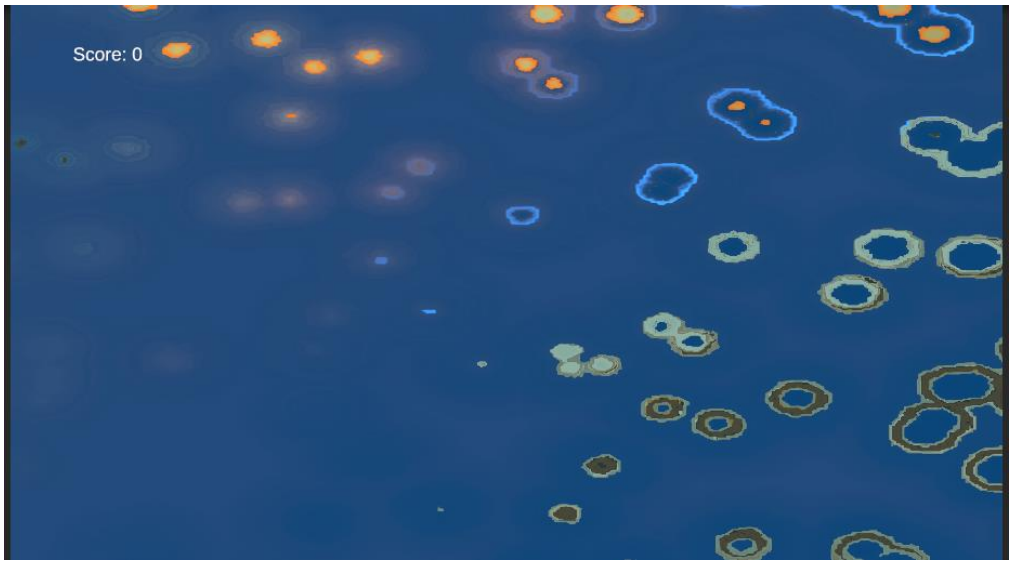


(imagem com o Alpha a descer)(Fundo preto meramento ilustrativo) (Noise Texture)

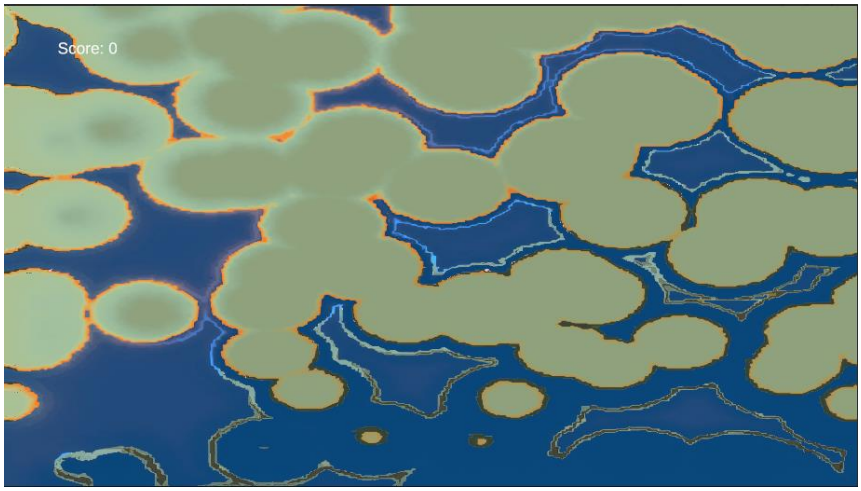
Com isto agora cria-se, dois fixed4 para guardar cada uma destas texturas, para que depois estas sejam usadas.

1.8.1: DISTORÇAO DA CAMERA

Primeiro para que consiga aplicar a distorção da camara simplesmente vai ser preciso dentro do frag, multiplicar o fixed4 que guarda a Noise Texture as uvs da textura principal da camara(col), ao aplicar simplesmente isto vai-se verificar que a imagem mostrada no ecrã modificou exponencialmente, mas não e o que pretendemos.



Por isso agora iremos criar um slider e iremos multiplicar esse slider pelo valor do fixed4 dado pelo noise texture, com isto já iremos ter mais liberdade em relação de como podemos mexer na distorção da camara, mas de qualquer das formas verificamos que ainda não possui o tal efeito pretendido.



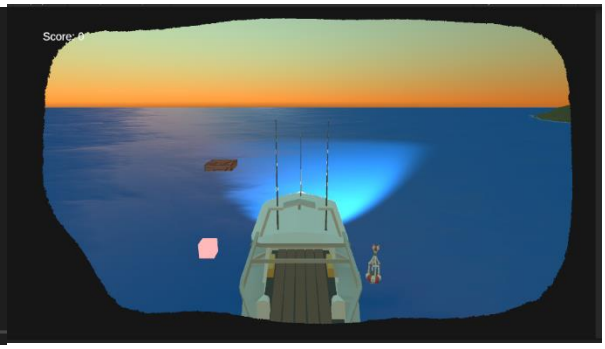
Para que o tal efeito fique o que nos esperamos basta só subtrair por 1 a multiplicação feita anteriormente dando assim a tal distorção da camara.


```
fixed4 col_2 = tex2D(_MainTex_2, i.uv);
fixed4 col_3 = tex2D(_MainTex_3, i.uv);
fixed4 col = tex2D(_MainTex, i.uv);

col = tex2D(_MainTex, i.uv*(1-col_3*_Controlador));
```

1.8.2: VISAO AFUNILADA

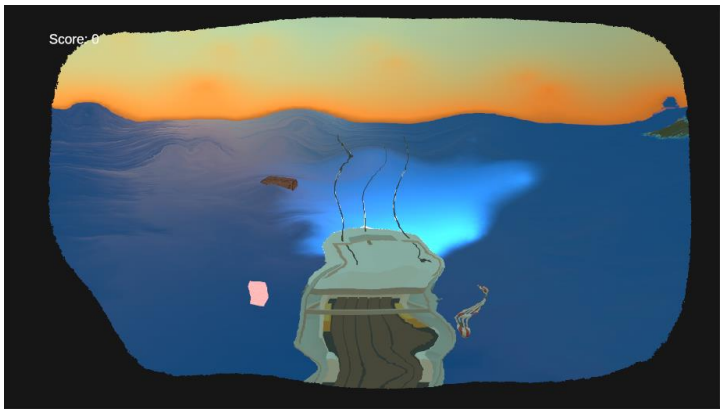
Para obter este efeito, usamos o fixed4 que esta a guardar a imagem que possui o Alpha a descer continuadamente, e dentro do frag iremos criar um if que estará a comparar o valor do Alpha do fixed4 da imagem e o valor do slider(col_2.a<_Controlador),faz-se nesta comparação com que o Alpha da imagem seja menor que o slider para que o que seja mostrado no ecrã seja o inverso da imagem dada(a parte branca da imagem fica invisível,e a parte invisível branca),este slider vai ser mesmo usado na distorção da camera porque vamos querer que ao bater ambos os efeitos atuem ao mesmo tempo e desapareçam ao mesmo tempo.

```
fixed4 col_2 = tex2D(_MainTex_2, i.uv);
fixed4 col_3 = tex2D(_MainTex_3, i.uv);
fixed4 col = tex2D(_MainTex, i.uv);


col = tex2D(_MainTex, i.uv*(1-col_3*_Controlador));

 if(col_2.a<_Controlador)
 {
  return _border_color;
 }
```



Para finalizar dentro deste if vai-se por que vai retornar uma cor que neste caso a cor que retorna e o preto, assim agora com isto através do slider conseguimos controlar manualmente o quanto distorcido e afunilado fica a visão da camara, mas o que nos queremos agora e fazer com que esta seja automatizada.

(Normal)



(distorcido e afunilado)

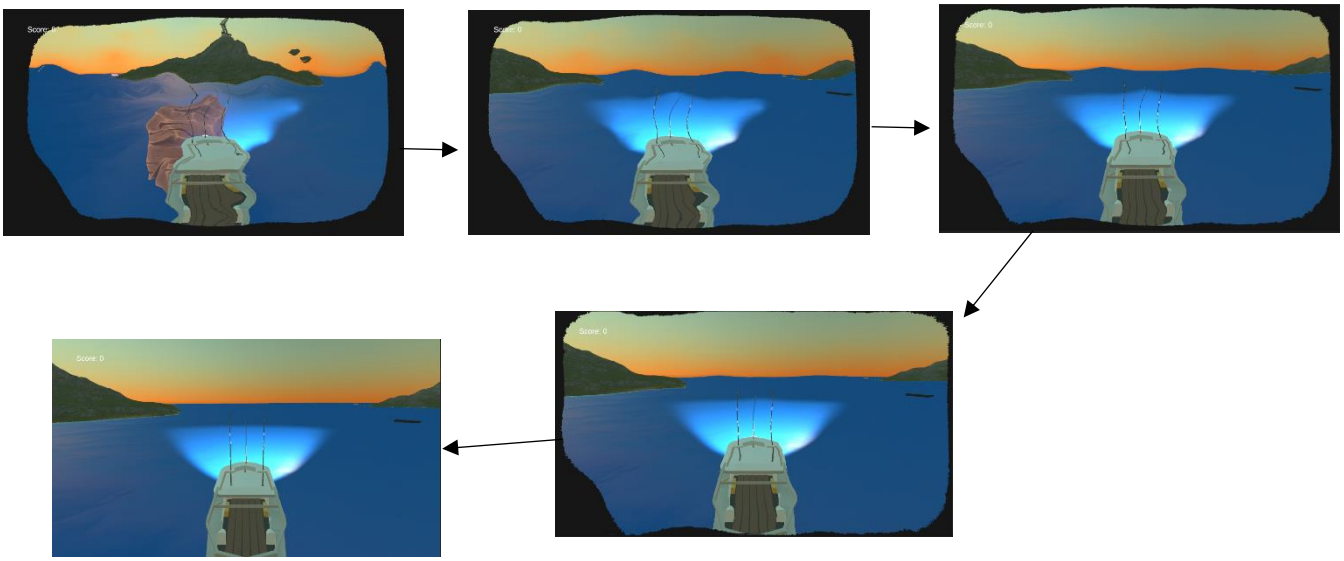
1.8.3: DETETAR AS COLISOES

Para que o efeito anterior fique automatizado iremos ter de fazer um script a parte para detetar as colisões com os obstáculos, para isso dentro de um script vamos criar as seguintes variáveis (bool colided) ;(controlador);A primeira variável criada vai servir para detetar quando o barco colidiu com os obstáculos sendo verdadeiro quando colide e vice-versa, para que o barco detete quando colidiu dentro do script criamos uma função(private void OnTriggerEnter(Collider other)) em que dentro desta iremos criar um if em que quando o barco colide com um objeto que possui uma tag("Obstacle") este ira chamar um IEnumerator, este IEnumerator vai servir para que a fim de um tempo o bool que fica verdadeiro passe para falso para que o efeito seja aplicado durante um tempo.

```
Mensagem do Unity | 0 referências
void Update()
{
    Shader.SetGlobalVector("_position_boat",transform.position);
    controlador = material.GetFloat("_Controlador");
    if (colided == true)
    {
        material.SetFloat("_Controlador", 0.1f);
    }
    if (colided == false)
    {
        if (controlador > 0)
        {
            controlador -= 0.02f * Time.deltaTime;
        }

        if(controlador < 0)
        {
            controlador = 0;
        }
        material.SetFloat("_Controlador", controlador);
    }
}
Mensagem do Unity | 0 referências
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Obstacle"))
    {
        StartCoroutine(dizy());
    }
}
1 referência
public IEnumerator dizy()
{

    colided = true;
    yield return new WaitForSeconds(2f);
    colided = false;

}
```

Tedo isto agora com a segunda variável iremos guardar o valor atual do controlador dentro do shader através do método(material.GetFloat).

Depois iremos criar um if em que quando o bool for true este ira dizer que o valor do controlador dentro do shader e igual ao máximo possível que neste caso e 0.1 e quando e falso a valor atual de 0.1 vai ser lentamente subtraído ate este valor ser igual a zero, nestes dois if's para que o valor do controlador seja sempre atualizado existe um método sempre no fim de cada if que é (material.SetFloat("_Controlador", controlador)).



2: CONCLUSÃO

Ao longo deste trabalho podemos ver vários usos de diferentes shaders a um nível pratico através do nosso jogo. Este jogo também demonstra a utilidade e a potencialidade do uso de shaders, não só a nível estico, mas também a nível funcional.

Vimos diversas funções de shaders como Post Processing, Depth Texture, Noises, alteração de vértices, entre vários outros, tendo sempre em foco como se integrariam no nosso jogo. Assim, diria que fizemos um projeto bem elaborado e complexo, onde todas as partes têm um propósito, permitindo-nos também explorar o uso mais pratico de shaders, implementado num projeto próprio.