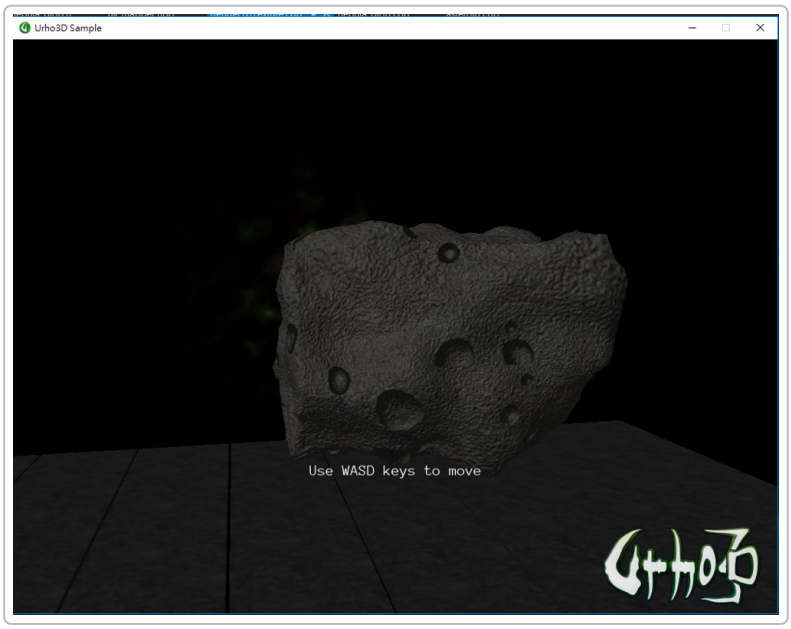
ChatGPT

# Criando Asteroides Procedurais Realistas em WebGL2

**Visão Geral:** Asteroides procedurais permitem gerar formas únicas de forma automática – ideal para telas de abertura impressionantes em jogos. Neste guia, vamos abordar duas frentes principais: **(1)** a geração da malha 3D deformada do asteroide (sua forma irregular, com relevos e crateras) usando ruído procedural (Perlin, Simplex, fractal etc.), e **(2)** a criação de **texturas procedurais** (mapas de altura, normais e difusos) que conferem ao asteroide aparência realista de rocha cheia de crateras. Veremos técnicas tanto em WebGL2 "puro" (GLSL customizado) quanto usando bibliotecas de alto nível como Three.js, sempre com foco em exemplos práticos, shaders GLSL e boas práticas de desempenho/qualidade visual.

**Nota:** WebGL2 traz melhorias (como texturas 3D, render targets avançados, etc.) que facilitam a geração procedural, mas requer a implementação manual de funções de ruído (GLSL não inclui ruído por padrão). Felizmente, existem bibliotecas e *snippets* prontos (por exemplo, a biblioteca de ruído de Stefan Gustavson com funções Perlin/Simplex/Worley em GLSL [1]). A seguir, organizamos o conteúdo em seções claras para guiá-lo passo a passo na criação dos asteroides procedurais.

## 1. Gerando a Malha Procedural do Asteroide




*Exemplo de asteroide gerado proceduralmente com superfície irregular e crateras (captura de um protótipo usando Urho3D).*

### 1.1 Base Esférica vs. Esfera de Cubo para o Asteroide

Para começar a geometria do asteroide, geralmente parte-se de uma forma aproximadamente esférica. Duas abordagens comuns:

- **Esfera convencional:** usar uma esfera pronta (ex: `THREE.SphereGeometry` no Three.js) como base. Entretanto, mapeamento de textura em uma esfera pode causar distorções indesejadas nos polos (o chamado problema do *"spherical texture mapping butthole"*, ou seja, a distorção nos polos da esfera) [2]. Para nossa aplicação com texturas procedurais, essas distorções podem complicar a distribuição uniforme de detalhes.

- **Cubo esferificado (Quadrilateralized Spherical Cube):** uma técnica preferida para planetas/asteroides é usar um cubo subdividido projetado em esfera. Ou seja, comece com um cubo subdividido e **normalize os vértices para formar uma esfera** [3]. No Three.js, por exemplo, pode-se usar `BoxGeometry` subdividido e então empurrar cada vértice para a distância do raio desejado. Essa "esfera de cubo" resulta em distribuição mais uniforme de vértices e facilita o mapeamento cúbico de texturas sem costuras evidentes [4]. (A referência [4] descreve um asteroide gerado como *spherical cube* com textura em forma de cubemap, evitando distorções de mapeamento.)

**Implementação (Three.js):** Você pode criar uma esfera facetada a partir de um cubo assim:

```
const geometry = new THREE.BoxGeometry(1, 1, 1, 32, 32, 32); // cubo
subdividido
geometry.vertices.forEach(v => v.normalize().multiplyScalar(raio));
```

Isso produz uma malha esférica base similar a `SphereGeometry`, porém com mapeamento mais amigável a texturas procedurais [3]. Alternativamente, o Three.js possui `IcosahedronGeometry(raio, subdivisoes)` que também gera uma esfera aproximada subdividida de forma uniforme (20 faces base icosaédricas) – útil como ponto de partida para deformar com ruído [5].

### 1.2 Deformando a Forma com Ruído Fractal (Perlin/Simplex)

Com a malha base pronta, o próximo passo é deformá-la para simular a irregularidade de um asteroide. Aqui entram os **ruídos procedurais**:

- **Ruído Perlin Clássico:** Gera padrões suaves pseudo-aleatórios. Útil para relevo orgânico. Entretanto, em 3D o Perlin clássico pode apresentar artefatos lineares, então muitos preferem o simplex.

- **Ruído Simplex:** Variante do Perlin introduzida por Ken Perlin em 2001, com menos direcionalidade e cálculo eficiente em GPU. Ideal para WebGL – muitas implementações GLSL usam simplex por ser mais rápido e menos propenso a artefatos [1].

- **Ruído fractal (fBM - *fractal Brownian motion*):** Combina múltiplas camadas (octaves) de ruído Perlin/Simplex em diferentes escalas e amplitudes, resultando em detalhes tanto de grande escala quanto finos. Essa técnica de somar ruídos de alta frequência com menor amplitude sobre ruídos de baixa frequência dá uma aparência natural e complexa [6] [7]. Em essência, o

> ruído fractal nos permite simular relevos grandes (colinas, saliências maiores) junto com rugosidades pequenas (rochas, saliência miúdas) no mesmo asteroide.

**Como aplicar à malha:** O conceito é deslocar cada vértice ao longo de sua normal por uma magnitude dada pela função de ruído. Em outras palavras, para cada vértice da esfera, calculamos `altura = ruído3D(posição_original * frequência) * amplitude`, então ajustamos `vértice_novo = vértice_original + normal * altura`. Isso cria relevos na superfície.

No **vertex shader GLSL**, podemos fazer isso diretamente, passando por uniforme um "tempo" ou "seed" se quisermos aleatorizar ou animar. Exemplo simplificado de shader (GLSL):

```
// pseudocódigo GLSL
attribute vec3 position;
attribute vec3 normal;
uniform float amplitude;
uniform float frequency;
varying vec3 vNormal;

float noiseVal = snoise(position * frequency); // snoise: função simplex 3D
float displacement = noiseVal * amplitude;
vec3 newPosition = position + normal * displacement;
...
gl_Position = projectionMatrix * modelViewMatrix * vec4(newPosition, 1.0);
vNormal = normalize(normalMatrix * (normal + derivaNormalDoRuido));
```

*(Nota: `snoise` seria uma função de ruído simplex implementada ou incluída da biblioteca. Também poderia-se usar uma função de turbulência/fractal que soma várias frequências.)*

No clássico tutorial de *vertex displacement* de *@alteredq* (2012), isso é demonstrado usando um icosaedro e ruído Perlin: o shader desloca vértices ao longo da normal por `displacement = -10.0 * noise + base` [8]. O resultado são formas orgânicas "derretidas" ou rochosas a partir de uma esfera [9] [8]. Abaixo, um trecho ilustrativo desse tutorial:

> "...usando uma esfera básica e perturbando os vértices com ruído Perlin... movemos a posição do vértice ao longo da normal em uma quantidade proporcional ao valor do ruído" [9] [10].

**Dicas para ruído no shader:**
- Utilize ruído **3D** (entrada tridimensional) para evitar padrões que se repetem em lat/long. Ou seja, passe a posição 3D do vértice (ou posição mapeada em esfera) à função de ruído. Assim, o ruído é avaliado no espaço 3D do asteroide, dando continuidade ao redor de toda a superfície.
- Para um efeito fractal, combine múltiplas amostras: e.g. `noiseTotal = noise(p*1) * 0.5 + noise(p*2) * 0.25 + ...` ou use uma função pronta de turbulência/**fBM**. No exemplo citado, eles usam uma função `turbulence(vec3 p)` que soma ruídos em diferentes escalas para produzir formas mais interessantes [11] [12].
- **Simplex vs Perlin:** Com poucas octaves, é possível notar diferença (Simplex tende a ser menos "gridado" que Perlin). Mas com várias octaves a diferença visual diminui [7]. Simplex 3D é uma boa escolha em GLSL pela eficiência.
- **Normal recálculo:** Após deslocar vértices, recompute as normais (no caso de fazer na CPU) ou calcule

no shader a nova normal derivada do mapa de altura procedural para iluminação correta. No Three.js, se você modifica vértices em JavaScript, chame `geometry.computeVertexNormals()` depois.

**Exemplo (Three.js ShaderMaterial):** No Three.js, podemos usar um **ShaderMaterial** para escrever nossos shaders de vértice e fragmento manualmente. O tutorial de AlteredQualia linkado mostra como configurar um ShaderMaterial com strings de shader embutidas [13]. Nele, definimos uniforms como `amplitude`, `frequency` e possivelmente passamos a função de ruído GLSL (pode ser colada do repositório do Stefan Gustavson [1]). Isso nos dá controle total para deformar a malha no GPU.

**Exemplo (Three.js CPU):** Alternativamente, podemos aplicar ruído via JavaScript: iterar sobre `geometry.vertices` e deslocá-los usando uma função de ruído (por exemplo, usando uma biblioteca JS de Perlin/Simplex). Essa abordagem CPU é simples e faz sentido se for um asteroide estático (pode ser feito uma vez na inicialização). Porém, para malhas de alta resolução, isso pode ser lento. O usuário XenoverseUp notou que gerar malha procedural no CPU era lento (vários segundos) para alta resolução, por isso migrou para um cálculo no fragment shader (GPGPU) para acelerar [14] [15]. Em suma: para asteroides únicos ou poucos, CPU pode bastar; para gerar muitos ou em tempo real, prefira deslocamento no vertex shader ou técnicas GPGPU.

### 1.3 Formas Irregulares Adicionais (Cortes, Escalas Aleatórias)

Um asteroide realista nem sempre é perfeitamente arredondado – pode ter formatos mais alongados ou facetados. Além do ruído fractal, podemos aplicar transformações aleatórias simples na malha base antes do deslocamento:

- **Escala não-uniforme:** Escalar a malha diferentemente em cada eixo (ex.: alongar em X, achatar em Z) para produzir um asteroide oval ou achatado. Faça isso aplicando um fator aleatório por eixo nos vértices antes de normalizá-los no raio desejado [16]. Isso gera variações macro na silhueta (por exemplo, um asteroide mais comprido).

- **Cortes por planos aleatórios:** Técnica interessante descrita em um projeto Urho3D: cortar pedaços do asteroide com planos randômicos [16]. Funciona assim: defina um plano (com posição/orientação aleatória) passando pelo modelo e "corte" empurrando vértices que ficam de um lado do plano em direção ao plano. Isso cria uma face meio chata em uma região, simulando uma fratura. Repetir com 2-3 planos randômicos dá um aspecto mais "astilhaçado" e não tão esférico. Essa técnica adiciona faces planas irregulares ao objeto, enriquecendo o formato. (Após os cortes, pode-se normalizar ligeiramente a malha ou suavizar transições.)

- **Facetamento deliberado:** Dependendo do estilo desejado, manter o asteroide meio facetado (não suavizar completamente as normais) pode dar um visual "low-poly" estilizado. Há projetos que deliberadamente usam asteroides facetados [17]. Mas assumindo que buscamos realismo, manter as normais suavizadas é melhor.

Combine essas transformações com o deslocamento de ruído: por exemplo, primeiro aplique escalas/cortes no cubo esferificado, depois execute o deslocamento fractal de vértices. Assim obtemos um asteroide de formato único, porém com superfície rugosa natural.

### 1.4 Simulando Crateras na Geometria

Crateras de impacto são características marcantes de asteroides. Gerar crateras procedurais pode ser desafiante apenas com ruído fractal puro (que tende a produzir bumps aleatórios, não círculos definidos). A seguir, apresentamos abordagens para adicionar crateras:

- **"Stamping" de Hemisférios:** Uma forma intuitiva é subtrair porções esféricas da superfície – essencialmente "imprimir" depressões esféricas (hemisférios invertidos) em pontos aleatórios. Por exemplo, o usuário PurpleCat descreveu gerar crateras como depressões hemisféricas aleatórias adicionadas à esfera [4]. Implementação: escolha N posições aleatórias na superfície (por exemplo, escolhe N vértices aleatórios como centro de crateras, ou amostre pontos aleatórios na esfera). Para cada ponto, defina um raio de cratera e para cada vértice dentro desse raio (ou distância angular no globo), desloque-o **para dentro** seguindo o perfil de uma calota esférica. O perfil de crateras reais costuma ter bordas levantadas e fundo em forma de tigela. Você pode modelar isso usando uma fórmula de esfera: por ex, a depressão pode seguir $z = -\sqrt{r^2 - d^2}$ (parte inferior de esfera) para um vértice a distância $d$ do centro da cratera, ajustando para que na borda ($d = r$) o deslocamento seja zero ou ligeiramente positivo (borda levantada). Esse método é mais fácil se trabalharmos em um **mapa de altura 2D** (veja seção de texturas), mas também pode ser feito iterando vértices em 3D.

- **Ruído Worley (Voronoi) como distribuição de crateras:** Outra técnica é usar ruído de Voronoi para marcar locais de crateras. Worley noise gera um padrão de "células" aleatórias. Podemos pegar o mapa de distância do Worley e extrair regiões circulares: basicamente cada "célula" do Worley vira a região de uma cratera. Um artista procedural relata: *"usei Worley noise para espalhar 'círculos' na superfície e então um ramp de cores para moldá-los como crateras"* [18]. Em shader, isso pode ser: `dist = worley3D(point)`, então aplique uma curva (por exemplo, um step/ suavização) onde se $dist < limiar$ formamos uma depressão. Esse método gera crateras de tamanhos aleatórios distribuídas naturalmente, sem precisar iterar crateras manualmente. Podemos depois deformar o fundo/forma com uma curva (como o *float curve* mencionado para dar o perfil côncavo realista) [19]. Inclusive, dá para adicionar detalhes como o pico central em crateras maiores – como discutido num fórum, isso exige passos extras (ex: adicionar um pequeno cone no centro após formar a cratera) [20].

- **Mistura de ruídos para crateras:** Uma dica prática de desenvolvedores é combinar diferentes ruídos para gerar um padrão craterado. Por exemplo, multiplicar duas camadas de ruído de forma a criar "manchas" isoladas. Um relato menciona que *multiplicar dois ruídos e ajustar com curvas resultou em um padrão de manchas na superfície*, ideal para base de crateras (fonte: usuário no Reddit, ver referência) [21]. Em outras palavras, um noise de baixa frequência define a distribuição geral e outro de alta frequência acrescenta granularidade dentro das manchas.

**Escolha da abordagem:** Para simplificar, muitos projetos optam por **não modelar crateras como geometria real**, mas sim incorporar as crateras na textura (altura/normal map). Modelar grandes crateras via deslocamento de vértice é viável se a malha tiver subdivisão suficiente naquela região. Já crateras pequenas demais exigiriam muitos polígonos – melhor tratá-las via **normal map** (ver próxima seção). Por exemplo, o projeto *Asteroid OpenGL* usa Perlin noise para relevo geral e **bump mapping para detalhes**, adicionando crateras proceduralmente sem alterar tanto a malha base [22]. Da mesma forma, um experimento em Urho3D gerou crateras no **mapa de altura** e depois calculou a normal map, mantendo a malha relativamente simples [23].

Em resumo, para crateras grandes ou médias, você pode combinar métodos: aplique alguns “stamps” para crateras maiores diretamente na malha, e deixe crateras menores por conta da textura normal. Isso equilibra realismo com performance.

## 2. Texturas Procedurais de Superfície (Altura, Normal, Difusa)

Além da forma 3D, a aparência visual do asteroide depende de suas texturas de superfície – principalmente o **mapa difuso** (cores), **mapa de altura/normal** (relevo fino) e possivelmente **mapa de oclusão/roughness** se usar um material PBR. Vamos focar nos principais.

### 2.1 Mapas de Altura e Normais Procedurais (Relevo fino)

Um mapa de altura (*heightmap*) é essencialmente uma imagem em tons de cinza que representa elevação da superfície. Podemos utilizá-lo de duas formas: - Como **displacement map** – se aplicado no vertex shader (ou via `MeshStandardMaterial.displacementMap` no Three.js) para deslocar vértices conforme os níveis de cinza. - Como base para **normal map** – calculando as derivadas (gradientes) do mapa de altura para gerar um mapa de normais que simule relevo na iluminação sem realmente deslocar geometria.

**Gerando o mapa de altura:** Podemos criar um heightmap proceduralmente combinando elementos: - **Ruído fractal base:** similar ao que usamos na malha, mas possivelmente em resolução maior para capturar detalhes finos. - **Crateras:** podemos literalmente desenhar crateras no heightmap. Por exemplo, start com uma imagem toda cinza médio; para cada cratera, desenhe um círculo escuro (depressão) com borda mais clara (rebordo da cratera). Programaticamente, isso pode ser feito iterando coordenadas ou usando funções de distância. - **Mistura com ruído de alta frequência:** adicione um pouco de “rugosidade” aleatória (por exemplo, noise de pequena amplitude) no heightmap para simular irregularidades de rocha dentro e fora das crateras [23].

Um procedimento relatado: 1. *“Gerar height map colocando algumas crateras aleatórias e ruído branco”* [24]. 2. Então converter esse heightmap em normal map.

Isso é exatamente o que o projeto Urho3D fez: colocou crateras + noise no mapa de altura e depois utilizou um shader para computar a normal map [23]. Existem algoritmos simples para converter heightmap em normal map (calculando o vetor normal via diferenças entre pixel vizinhos). Há até ferramentas online (ex: NormalMap-Online) – no caso, eles portaram o shader do NormalMap-Online para C++ para fazer isso em runtime [23].

**No contexto WebGL/Three.js:** Você pode gerar o heightmap no CPU (como um canvas 2D ou array) ou usar a GPU: - *CPU approach:* criar um Canvas ou ImageData do tamanho desejado (ex: 512x512 px), iterar preenchendo com noise e crateras. Em JavaScript, bibliotecas como perlin-noise ou simplex-noise podem ajudar a gerar ruído. Para crateras, você pode iterar algumas posições e desenhar formas (ex: usando CanvasRenderingContext2D radial gradients ou manipulando pixel arrays). - *GPU approach:* usar um **render target** com um shader que escreve a altura em gl_FragColor. O truque é renderizar um quad cobrindo uma render target 2D, onde o fragment shader calcula: `altura = noiseFractal(x, y) + craterFunction(x,y)`. Foi assim que Holger Ludvigsen acelerou a geração de texturas planetárias: em vez de gerar textura no CPU, ele escreveu um shader e renderizou em um `THREE.WebGLRenderTarget`, obtendo a textura procedural direto na GPU [25] [26]. Depois, essa textura pode ser usada normalmente como mapa. Esse método “GPGPU” tira proveito do paralelismo do fragment shader (cada pixel calcula seu valor de altura independentemente) [27] [15]. Com WebGL2, fica fácil renderizar num framebuffer offscreen e usar o resultado como textura sem cópia para CPU.

Uma vez obtido o heightmap: - **Gerar Normal Map:** Você pode fazer no próprio shader final (calculando derivadas do heightmap dentro do shader de fragmento via amostras offset de textura) ou pré-calcular. O Three.js suporta fornecer um normal map diretamente. Ferramentas: ou use a técnica de GPU (um shader que leia heightmap e calcule normal – já existem muitos exemplos de shaders triplanar que fazem isso), ou faça em JS (menos recomendado se imagem grande, mas possível). No caso do planet demo, Holger menciona que converter bump para normal no GPU facilitou adicionar detalhes lindos sem sobrecarregar geometria [28] [29].

**Aplicação:** No Three.js, para materiais standard ou Phong, você pode aplicar:

```
material.displacementMap = heightTexture;
material.displacementScale = escala; // quão profundo será o deslocamento
material.normalMap = normalTexture;
material.map = diffuseTexture;
```

Isso fará com que a própria pipeline de shaders do Three.js desloque vértices (no vertex shader) e use a normal map (no fragment shader) automaticamente. Tenha certeza de que a geometria tem subdivisão suficiente para a displacementMap surtir efeito – caso contrário, depende só da normal map.

**LOD/Resolução:** Vale notar que usar um heightmap 2D fixo mapeado numa esfera pode reintroduzir costuras ou distorções se não for feito cuidadosamente. Por isso, algumas abordagens preferem **texturização triplanar** em vez de UV tradicional. Triplanar mapping aplica a textura projetando de 3 eixos e mesclando, evitando costuras. O projeto Urho3D testou ambos: UV (mas com seams) vs triplanar (sem seams porém normal map fica menos nítido) [30]. Para um único asteroide, você pode escolher UV unwrap manual (tentando esconder costuras) ou até gerar a textura como um cubemap (6 faces) para mapear perfeitamente na esfera (como PurpleCat fez com diamond-square noise em cada face do cubemap) [4].

## 2.2 Textura Difusa (Coloração de Rocha)

A textura difusa define a cor base da superfície. Mesmo que asteroides sejam geralmente acinzentados ou amarronzados e pouco vivos em cor, adicionar variações sutil pode melhorar o realismo. Abordagens: - **Derivar do próprio heightmap:** áreas elevadas e baixas podem ter tonalidades ligeiramente diferentes. Por exemplo, crateras podem parecer mais escuras no fundo devido à sombra permanente ou composição diferente. Você pode fazer o fragment shader pintar pixel com base no valor do heightmap: ex: um gradiente de cinza para marrom conforme altura, ou usar o ângulo da normal para escurecer depressões. - **Ruído de cor:** use um ruído de baixa frequência para variar a cor em manchas suaves – simula heterogeneidade do material (como regiões com composição mineral diferente). Multiplique isso pela cor base do asteroide. Por exemplo, um noise 2D pode modular entre um tom #909090 e #6f6050. - **Detalhes de crateras:** Poderíamos gerar um mapa de *occlusion* simples da cratera – e.g., escurecer ligeiramente dentro das crateras para dar profundidade. Se já temos o heightmap, podemos derivar a oclusão aproximada (pixels muito "côncavos" ficam mais escuros). - **Mapeamento procedural no shader:** Em vez de usar uma textura 2D para cor, podemos escrever lógica no fragment shader: ex: `if(mod(floor(pos*(freq)),2)==0) {color = ...}` (só exemplo de listras), mas no caso de rocha, ruídos suaves são melhores. No Shadertoy existem materiais procedurais de rocha que combinam vários noise para base color.

Uma ideia interessante é utilizar **bi-color noise**: pegar dois noise de escalas diferentes e combiná-los para obter manchas. O reddit de "zero artistic skill" menciona usar duas noises multiplicadas e ajustadas para criar spots [21]. Em termos práticos:

```
float n1 = snoise(p * freq1);
float n2 = snoise(p * freq2);
float spots = smoothstep(threshold1, threshold2, n1 * n2);
```

Isso poderia gerar pontos esparsos (onde ambos noises estão altos simultaneamente). Esses pontos podem indicar manchas de cor ou pequenas crateras.

**Exemplos Realistas:** Asteroides tendem a ter cores bem sutis. Você pode basear a cor em fotos de asteroides reais (geralmente cinza escuro, às vezes com toques avermelhados ou áreas mais claras devido a composição). Um toque de marrom-avermelhado pode ser adicionado (simulando óxido de ferro?).

### 2.3 Shaders GLSL Personalizados vs. Materiais Prontos

Se você optar por um ShaderMaterial ou escrever shaders WebGL puros, pode integrar tudo no código GLSL: - **Vertex shader:** calcula posição deslocada e também pode passar informações ao fragment (ex: altura, posição original, etc. via varyings). - **Fragment shader:** combina as várias componentes – pode calcular a cor difusa proceduralmente e aplicar iluminação, pode aplicar normal mapping calculando derivadas de height procedurais etc. Por exemplo, um fragment shader pode amostrar um normal map textura ou computar normal via função que lê height nos arredores.

No caso de usar Three.js com materiais prontos, você pode usar extensões como: - `MeshStandardMaterial` com mapas (difuso, normal, displacement, roughness). Isso te poupa de escrever equações de iluminação física, pois o Three.js cuida disso. Você apenas fornece as textures procedurais geradas. - *NodeMaterial/ShaderNode (Three.js)*: Three.js mais recente tem um sistema de nós (TSL) que facilita criar materiais procedurais sem escrever GLSL manual, porém isso foge do escopo aqui; citamos apenas se quiser explorar.

## 3. Implementação: WebGL2 Puro vs. Three.js

Tanto usar WebGL2 diretamente quanto utilizar Three.js (ou similares, ex: Babylon.js) são opções válidas para nosso objetivo. Vamos comparar brevemente abordagens e destacar boas práticas em cada caso:

### 3.1 WebGL2/GLSL Puro

Trabalhar direto com WebGL2 + GLSL dá controle máximo, ao custo de mais código “boilerplate”. Algumas considerações: - **Setup manual:** você terá que criar o contexto WebGL, compilar shaders, criar buffers de geometria (por ex, carregar vertices/indices da esfera), enviar uniforms etc. É verboso, mas existem exemplos de código para base (por ex, um simples cubo rotacionando em WebGL precisa de muitas linhas de código OpenGL) [31] [32] . - **Shaders customizados:** Aqui é onde brilha – você escreve o vertex shader com deslocamento procedural e o fragment shader com texturas procedurais. Pode integrar bibliotecas GLSL de noise facilmente (copie o código no shader). Certifique-se de definir precisão alta (`precision highp float;`) se for precisar de detalhes no noise (WebGL2 costuma suportar highp em fragment). - **Recursos do WebGL2:** WebGL2 adiciona algumas coisas úteis como: texturas 3D (pode armazenar um volume de noise precomputado e amostrar no shader), múltiplos render targets (pode gerar diferentes mapas em um pass), melhor suporte a floating point textures etc. Aproveite isso se fizer sentido – ex: você pode pré-computar uma textura 3D de noise ($32^3$) no CPU e mandar para GPU, então no shader basta amostrar `noiseTex3D(coord)` ao invés de calcular função noise. Isso pode ser mais rápido dependendo do caso. - **Compute shaders?** Não, WebGL2 não tem

compute shaders (esses só no WebGPU ou OpenGL 4.3+). Mas conforme mostrado antes, podemos simular "compute" usando fragment shader render-to-texture (a técnica usada no Procedural Planet Generator [27] [33]). Basicamente, usamos um fragment shader para calcular dados e escrever num texture, depois usamos essa texture em outro pass. - **Desempenho:** GLSL permite grande paralelismo, mas cuidado com loops não necessários. Gerar 1 octave de noise é barato, mas se você faz 6~8 octaves + muitas crateras + outras operações *por vértice*, pode pesar se sua malha tiver milhares de vértices. Considere equilibrar: talvez aplicar ruído grosseiro no vértice e deixar detalhes para o fragment via normal map. Além disso, use **LOD** se aplicável – ex: um asteroide distante não precisa 32k polígonos. Sem tessellation shader no WebGL2, você teria que trocar manualmente de geometria ou usar técnica de malha multi-res. - **Instanciamento:** Se sua cena tem campo de asteroides (muitos asteroides), prefira instanced drawing em vez de draws separados. Você pode usar `ANGLE_instanced_arrays` (WebGL1) ou instancing nativo (WebGL2). Cada instância poderia usar um uniform diferente seed para noise, gerando asteroides variados a partir de um único shader/programa. O projeto AsteroidOpenGL cita uso de *instanced rendering* para um cinturão com 250k asteroides! Cada instância recebia um *noise vector* e escala diferente para variar a forma gerada [34]. WebGL2 certamente pode instanciar milhares de asteroides mais simples.

Em resumo, WebGL2 puro é poderoso mas requer manejo de muitos detalhes. A vantagem é conseguir otimizações sob medida e integrar tudo num único shader se quiser.

### 3.2 Three.js (ou Bibliotecas 3D)

Three.js simplifica imensamente a criação de cenas WebGL. Pontos relevantes para nosso caso: - **Facilidade de cena e câmera:** Com poucas linhas você cria uma cena, câmera e renderer WebGL [35] [36], sem precisar mexer diretamente com contextos ou loop manual (basta usar `renderer.render(scene, camera)` dentro de `requestAnimationFrame`). - **Geometrias e utilitários:** Three.js fornece geometria esférica, icosaédrica, Box etc. para começar. Você pode modificar os vértices da geometria facilmente: `geometry.attributes.position.array` ou antigo `geometry.vertices`. Lembre de atualizar as normais. Three.js também tem métodos como `applyMatrix4` para escalar/rotacionar malha facilmente (útil para aquela etapa de escala aleatória). - **Materiais de alto nível:** Como mencionado, materiais padrão do Three.js suportam mapas de texturas variados. Você pode gerar suas texturas procedurais (por exemplo, via canvas ou via render target) e apenas atribuí-las. O Three cuida de carregar para GPU e usar no shader dele. Isso é conveniente e garante integração com iluminação física (PBR) facilmente. Exemplo: aplicar `mesh.material.normalMap = generatedNormalMapTexture;` e pronto – o asteroide exibirá os relevos nas interações de luz. - **ShaderMaterial e exemplos:** Se quiser controle de shader, Three.js permite usar ShaderMaterial ou RawShaderMaterial. O primeiro fornece ainda algumas uniforms automáticas (como matrices) e sintaxe amigável (insere common includes), enquanto RawShaderMaterial você escreve 100% do código GLSL. O tutorial do AlteredQualia demonstra o uso de ShaderMaterial para rodar um vertex shader custom (ruído Perlin) e fragment basicão [13]. Existem também **exemplos e repositórios**: por exemplo, o repositório `XenoverseUp/procedural-planets` no GitHub usa Three.js e implementa geração de terreno com múltiplos noise simplex em camadas [37] – vale conferir para ver como integraram fragment shader para gerar malha (eles inclusive exportam a malha gerada como OBJ). Outro exemplo é o blog do Holger Ludvigsen [38], que mostra passo a passo gerar planeta no Three.js, incluindo usar um shader custom para a atmosfera e texturas procedurais para o terreno. - **Extensões e comunidade:** Three.js tem uma comunidade grande, então você encontra *plugins* ou snippets – por ex, **three-noise** (uma lib de ruído para three), ou posts no forum sobre procedural generation [27]. No forum oficial há showcases de planetas e asteroides. Também existe o Three.js Journey e outros tutoriais que, embora foquem em terreno, as técnicas de

deslocamento valem para asteroides (e.g. Bruno Simon's Three.js Journey tem uma seção sobre terreno procedural com ruído e shaders).

- **Performance e boas práticas:** Three.js, sendo uma camada a mais, pode ser ligeiramente menos performático que WebGL puro, mas geralmente o gargalo está nos cálculos GPU de qualquer forma. Para otimizar:
- Use BufferGeometries (padrão no Three r125+), evite geometries legadas. BufferGeometry permite usar `mesh.geometry.setAttribute('position', ...)` etc., e é mais eficiente.
- Se gerar geometria no CPU, faça-o uma vez e em seguida use `.dispose()` em buffers se for descartá-los, para liberar memória GPU.
- Para muitos asteroides: use **InstancedMesh** do Three.js (permite desenhar milhares de cópias de uma malha com diferenças em transform e até em atributos via *shader attributes* or vertex texture fetch).
- Limite o tamanho de texturas procedurais ao necessário – 512px ou 1024px de mapa de normal já pode ser suficiente para detalhes, não precisa 4K a menos que a câmera chegue muito perto.
- Three.js também possui **LOD** utilities se quiser trocar modelo conforme distância.

Em termos de fidelidade visual, Three.js lhe permite facilmente adicionar uma iluminação direcional ou ambiental, sombras (shadow maps) e até pós-processamento (bloom, etc.) para caprichar na apresentação da tela de abertura. Por exemplo, no AsteroidOpenGL original eles usaram um skybox estrelado e uma fonte de luz representando o Sol, com sombras projetadas [39] – tudo isso você reproduz em Three.js definindo um `DirectionalLight` e `Scene.background` com textura de estrelas, etc.

## 4. Dicas de Performance e Estética

Para consolidar, aqui vão **algumas boas práticas** ao criar asteroides procedurais:

- **Use detalhamento hierárquico:** Combine baixa geometria + normal map para detalhes. Não exagere nos polígonos para tentar modelar cada pedrinha – pequenas irregularidades ficam mais eficientes em mapas de textura (normal map) do que em geometria. Uma malha base com alguns milhares de triângulos pode parecer incrível se tiver um bom normal map simulando crateras miúdas e rugosidade [22].

- **Balanceie octaves de ruído:** Para aparência natural, use 2-4 octaves de ruído fractal no shape. Muitas octaves de amplitude alta tornam o asteroide espinhoso demais (ruído de alta frequência exagerado). Em asteroides reais, vemos superfícies relativamente lisas com crateras sobrepostas. Uma configuração comum: 1ª octave: grandes variações (~10-20% do raio), 2ª octave: médio (~5%), 3ª: pequenos (~1-2%), octaves superiores bem sutis. Ajuste o ganho (amplitude decrescente) e lacunarity (aumento de frequência) para obter um relevo convincente.

- **Crateras de vários tamanhos:** Asteroides apresentam crateras de diâmetros variados. Simule isso adicionando crateras grandes, médias e pequenas. Se for algoritmo (como Worley), tente repetir o processo em diferentes escalas [19]. Se for manual, sorteie raios diferentes e quantidades proporcionais (muitas pequenas, poucas grandes).

- **Iluminação e material:** Utilize um modelo de iluminação que destaque relevo. Um `MeshStandardMaterial` com mapa normal e uma luz direcional cria sombras nas crateras e saliências. Ajuste a intensidade da luz e posição para um ângulo rasante – isso realça as irregularidades (sombra projetada). Considere ativar sombras no renderer para que o asteroide

projete sombra em si mesmo nos relevos mais altos (auto-sombra via normal map não ocorre, então sombra de verdade dá mais contraste nas crateras profundas).

- **Textura de alta definição:** Se o asteroide ocupar grande parte da tela, garanta que a resolução do normal map/diffuse seja suficiente para não pixelar. Talvez 1024x1024 ou 2048x2048 se houver muitos detalhes finos. Caso contrário, texturas procedurais de 512px podem bastar e ainda economizar GPU.

- **Evite costuras visíveis:** Se usar UV mapping, tente que a borda de UV fique em áreas menos visíveis (ex: mapeamento cúbico minimiza, mas pode ainda ter seams entre faces se textura não casar exatamente). Em último caso, triplanar mapping (sample difusa/normal 3x e misturar) remove costuras – útil se você não quer se preocupar com UVs, embora possa suavizar um pouco os detalhes.

- **Animação opcional:** Para uma tela de abertura dinâmica, você pode rotacionar lentamente o asteroide (ex.: `mesh.rotation.y += 0.005`). Como toda geração é procedural mas fixa no modelo, rotacionar não muda o shape, apenas a vista. Se quiser um efeito de asteroide "vivo", poderia animar ligeiramente os parâmetros de ruído (e.g., offset no ruído ao longo do tempo) para criar um asteroide que se deforma sutilmente – mas isso não é realista fisicamente; então a menos que seja um "asteroide alienígena maleável", mantenha estático.

- **Testes de performance:** Procedural costuma ser pesado no início. Faça testes em diferentes hardware. Se a geração estiver lenta, considere pré-computar algumas coisas (por ex, gerar texturas uma vez e reutilizar). Se usar Three.js, use o DevTools (Stats.js) para monitorar FPS. Um asteroide único não deve ser um problema para GPUs modernas, mas se for povoar a cena com muitos, tenha atenção ao orçamento de polígonos e preenchimento de pixels (talvez reduza resolução de texturas para asteroides distantes, etc.).

## 5. Recursos Úteis e Exemplos

Para aprofundar e ver códigos de exemplo, recomendamos:

- **Tutorial de Vertex Displacement (GLSL + Three.js):** *"Vertex displacement with a noise function"* por @alteredq [9] [8] – Mostra passo a passo como usar ruído (Perlin) em shader para deformar uma esfera, com código no GitHub. Excelente para entender integração de ruído em shaders.

- **Artigo "Procedural Planet" (Partes 1 e 2):** por Holger Ludvigsen [38] [25] – Embora foque em planetas, cobre conceitos relevantes: usar cubo esferificado, gerar textura de terreno com Perlin fractal, mover cálculo pro GPU, e aplicar bump/normal map [25] [40]. Código fonte disponível no GitHub do autor (holgerl/ procedual planet).

- **Repositório "Procedural Planets" (Three.js + GPGPU):** GitHub: *XenoverseUp/procedural-planets* [37] – Projeto moderno (2024) que gera planeta procedural usando várias camadas de noise Simplex, calculando altura via fragment shader (fake compute). Útil para ver técnicas de desempenho e exportação de malha. Inclui demo web.

- **Asteroides em OpenGL (C++):** GitHub: *rekfuki/asteroidOpenGL* – Cena com asteroide principal e cinturão, usando noise + tessellation. O README destaca uso de Perlin noise para terreno e bump mapping para detalhes, além de instancing [41] [34]. O código é em C++ OpenGL, mas lógica de geração pode inspirar implementação WebGL.

- **Geração de Crateras Procedurais:** Thread no Reddit *"Procedural planetary craters"* – Discussão entre artistas usando Blender/Unity, com dicas como uso de Voronoi noise para crateras, ajuste de curvas para formato, e até adicionar raios de ejecta [42] [20]. Embora seja mais artístico, contém ideias aplicáveis em shaders.

- **Biblioteca GLSL Noise:** *stegu/webgl-noise* [1] – Implementação pronta de ruídos (Perlin, Simplex 2D/3D/4D, Worley etc.) em GLSL, de Stefan Gustavson. Você pode copiar essas funções para seus shaders WebGL2. O repositório inclui exemplos comparativos e é amplamente usado em projetos procedurais.

- **Three.js – Exemplos de Terreno Procedural:** A coleção oficial do Three.js possui exemplos (por enquanto em experimento TSL) como `tsl / procedural / terrain`. Além disso, o course *Three.js Journey* tem uma seção sobre *Procedural Terrain Shader* que, embora paga, ensina conceitos de deslocamento de terreno com noise (analogamente aplicável a asteroides) [43].

Com essas orientações e recursos, você deve conseguir criar asteroides procedurais visualmente ricos. Lembre-se de iterar experimentando parâmetros de noise, quantidade de crateras, etc., até atingir o visual desejado. Boa codificação e bons jogos! [41] [37]

---


[1] [7] GLSL noise library: Simplex and Perlin noise
https://stegu.github.io/webgl-noise/webdemo/

[2] [3] [6] [31] [32] [35] [36] [38] Procedural Planet in WebGL and Three.js | by Holger Ludvigsen | Bekk
https://blogg.bekk.no/procedural-planet-in-webgl-and-three-js-fc77f14f5505?gi=f5fe0a674934

[4] Asteroid with craters : r/proceduralgeneration
https://www.reddit.com/r/proceduralgeneration/comments/1cln4em/asteroid_with_craters/

[5] [8] [9] [10] [11] [12] [13] Vertex displacement with a noise function using GLSL and three.js - Blog - Clicktorelease
https://www.clicktorelease.com/blog/vertex-displacement-noise-3d-webgl-glsl-three-js/

[14] [15] [27] [33] [37] Procedural Planet Mesh Generator (GPGPU) - Showcase - three.js forum
https://discourse.threejs.org/t/procedural-planet-mesh-generator-gpgpu/69389

[16] [23] [30] GitHub - ab4daa/procedural_asteroid: attempt to generate asteroid using Urho3D
https://github.com/ab4daa/procedural_asteroid

[17] WIP - Procedural Asteroid Field (trying to find my style) - Reddit
https://www.reddit.com/r/proceduralgeneration/comments/1g2xk8z/wip_procedural_asteroid_field_trying_to_find_my/

[18] [19] [20] [42] Procedural planetary craters : r/proceduralgeneration
https://www.reddit.com/r/proceduralgeneration/comments/1j5utio/procedural_planetary_craters/

[21] Pre-rendered procedural asteroids with zero artistic skill. - Reddit
https://www.reddit.com/r/gamedev/comments/b7colj/prerendered_procedural_asteroids_with_zero/

[22] [34] [39] [41] GitHub - rekfuki/asteroidOpenGL: A 3D asteroid space scene implemented using OpenGL
https://github.com/rekfuki/asteroidOpenGL

[24] Procedural generate asteroid - Urho3D
https://discourse-urho3d.github.io/t/procedural-generate-asteroid/5055/

[25] [26] [28] [29] [40] Procedural Planet in WebGL and Three.js — Part 2 | by Holger Ludvigsen | Bekk
https://blogg.bekk.no/procedural-planet-in-webgl-and-three-js-part-2-33d99bbb2256?gi=0320f1327f91

43 Procedural Terrain Shader - Three.js Journey
https://threejs-journey.com/lessons/procedural-terrain-shader