Fachhochschule Wedel
Fachbereich Informatik

Seminarausarbeitung

Animiertes fraktales Rendering in Echtzeit

Alexander Treptow

Abgegeben am:
10. August 2009

Autor:
Bsc. Alexander Treptow
Tannenallee 43
22844 Norderstedt
Tel: (040) 522 57 51

Referent:
Prof. Dr. Christian-Arved Bohn
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Tel: (04103) 80480

Inhaltsverzeichnis

1 Einleitung

1.1 Kurzbeschreibung

Diese Ausarbeitung ist im Rahmen des Seminarleistung im Masterstudiengang Informatik entstanden. In dieser Ausarbeitung zum Thema 'Animiertes fraktales Rendering in Echtzeit' werde ich erläutern, wie Fraktale bzw. Noise-Funktionen in Echtzeit gerendert werden können. (Noch mehr Text?)

1.2 Motivation

Motiviert wird 'Animiertes fraktales Rendering in Echtzeit' durch die Möglichkeit der Noise-Funktionen die Natur annährend gut abzubilden. Es wird also erreicht, dass natürliche Phänomene, wie Landschaften, Wolken, Planeten usw., dargestellt werden können ohne diese von Hand modellieren zu müssen. Sowie die Tatsache, dass diese Eigenschaft auch gerne in interaktiven Anwendungen erwünscht ist, da sie auf sehr einfache Weise eine recht starke Immersion erzeugt.

Dieser Ansatz mit Noise-Funktionen bietet grundsätzlich die weiteren Vorteile, dass die virtuelle Umgebung natürlich erscheint. Sie ist dabei selbstähnlich ohne ein sichtbares Muster vorzuweisen. Zudem werden endlose Strukturen ermöglicht, die nur durch die Zahlendarstellung der Maschine begrenzt werden. Und es ist möglich beliebig viele Detailierungsgrade (Level-Of-Detail - LOD) darzustellen.

Die Tatsache, dass Funktionen verwendet werden ermöglicht es Strukturen zu erzeugen, die beliebig in ihrer Art Variieren. Wolken zum Beispiel existieren in viele verschiedenen Größen und Formen, diese lassen sich allein schon durch die Funktionsparameter beeinflussen.

Besonder geeignet wäre dieser Ansatz aus einem Grund auch für browser-basierte Spiele: Es sind extrem wenig Daten nötig um ein Objekt inklusive dessen Textur zu erzeugen. Damit können Ladezeiten verringert und Kosten gesenkt werden. Für andere Anwendungen bedeutet dies, dass z.B. beim Wechsel zwischen Bereichen in der virtuellen Umgebung nicht Massen von Texturen in den Speicher der Grafikkarte geladen werden müssen. Dies verringert ebenfalls die Wartezeiten für den Benutzer.

Nachdem die Vorzüge von Noise vorgestellt wurden stellt sich wohl jeder die Frage, warum diese Technik in aktuellen virtuellen Umgebungen so gut wie nie verwendet wird.

Das Problem dieser Technik ist, dass die Noise-Funktion in jedem Punkt bestimmt werden muss und zwar in jedem Renderingdurchgang neu. Zudem waren diese Funktionen bis vor wenigen Jahren nicht auf der Grafikhardware berechenbar.

Typische Anwendungen an Strukturen für Noise-Funktionen wären:

- 1D → Handschriften, Zeichnungen
- 2D → Landschaften, Wolken, Texturen
- 3D → 3D Wolken, Planeten, Animierte Texturen
- 4D → Animierte Planeten, -Wolken, -Solid Textures

1.3 Definitionen

Erreichen wollen wir im folgenden ein Kombination aus der Berechnung von Fraktalen- bzw. Noise-Funktionen und Echtzeit, also interaktive Anwendungen.

1.3.1 Fraktale - Noise

Fraktale sind natürliche oder künstliche Gebilde oder geometrische Muster, die einen hohen Grad von Selbstähnlichkeit besitzen und durch eine Funktion oder Regel beschreibbar sind.

Fraktale an sich eignen sich eher selten um natürliche Gebilde darzustellen, da die Selbstähnlichkeit in der Regel so hoch ist, dass Muster erkennbar sind. Dies ist in der Natur typischerweise nicht der Fall.

Noise in diesem Sinne ist eine Funktion die Zufallswerte generiert. Noise ist kein 'weißes Rauschen' sondern ein Fraktal mit statistischer Selbstähnlichkeit.

Die Zufallswerte, die eine Noise-Funktion erzeugt sind nicht wirklich zufällig, was zu 'weißem Rauschen' führen würde. Sie werden durch einen Pseudo-Zufallsgenerator erzeugt und sind somit reproduzierbar. Das ist ein wesentliches Kriterium wenn virtuellen Umgebungen generiert werden sollen, da diese bei wiederholtem Laden oder auch nur beim nächsten Renderingdurchgang identisch sein müssen.

Ebenso genügen Noise-Funktionen allen anderen Anforderungen:

- Natürlichkeit - natürliche Strukturen nachbildend

- Unendlichkeit - Strukturgröße wird nicht durch Speicher begrenzt
- Modifizierbarkeit - sich über die Zeit ändernd
- Reproduzierbarkeit - gleich Parameter erzeugen gleichen Wert
- Variabilität - verschiedene Parameter erzeugen verschiedene Werte bzw. es existieren mehrere Funktionen die ähnliche Ergebnisse erzeugen

1.3.2 Echtzeit

Realtime-Rendering bezeichnet das schnelle Erstellen synthetischer Bilder, so dass der Betrachter direkt mit der virtuellen Umgebung interagieren kann. Als echtzeitfähig bezeichnet man ein Applikation die > 15 *FPS erzeugt, obwohl der Mensch schon Bildfolgen ab 8 FPS als 'flüssig' wahrnimmt.*

2 Noise

2.1 Arten

Es existieren sehr viele verschiedene Arten von Noise-Algorithmen, die über unterschiedliche Eigenschaften wie Visualisierbarkeit und Rechenaufwand verfügen. Die meisten dieser Algorithmen sind sich dennoch sehr ähnlich, so dass es ausreicht hier exemplarisch den meist verwendeten Algorithmus vorzustellen.

Die meisten Noise-Algorithmen werden den Bereichen Gradient- oder Value-Noise zugeordnet. Der bekannteste und im folgenden beschriebene Algorithmus ist Perlin-Noise und wird dem Gradient-Noise zugeordnet. Gradient-Noise spannt seine Funktion über mit festem Abstand gewählte Punkte auf und berechnet für jeden dieser Punkte je nach Dimension einen oder mehrere Gradienten. Diese Gradienten werden ja nach Funktion unterschiedlich interpretiert. Oft wird auch zwischen ihnen interpoliert. Value-Noise hingegen erzeugt keine Gradíenten sondern einfache Werte und interpoliert zwischen diesen.

Als bekannte Noise-Funktionen sind aufzuführen:

Perlin Noise, Simplex Noise, Wavelet Noise, Lattice Convolution Noise, Sparse Convolution Noise, Worley Noise, Cell Noise, Voronoy Noise, Simulation Noise, ...

2.2 Erzeugen von Noise

Perlin Noise wurde 1983 von Ken Perlin entwickelt. Ich werde hier den Basisalgorithmus skizzieren und erst im Kapitel Echtzeit auf mögliche Erweiterungen oder Verbesserungen eingehen.

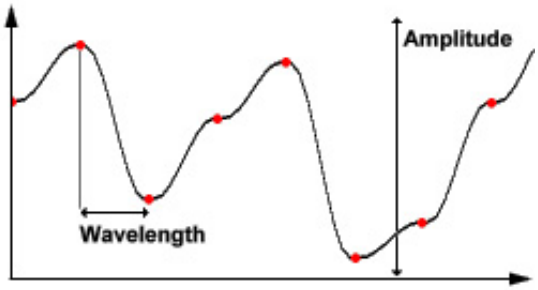
Perlin Noise eignet sich sehr gut zum Rendern von natürlichen Strukturen, da relativ Wenig Rechenzeit pro Berechnung der Noise-Funktion benötigt wird. Abgesehen davon ist es der meist verwendete Noise-Algorithmus und relativ gut verständlich. Das einzige Problem des Basisalgorithmus ist, dass dieser nicht divergenzfrei ist. Wodurch er sich nicht eignet um Pfade für Objektbewegungen oder Simulationen zu erstellen.

2.2.1 Die Noise-Funktion

Die folgende C-Funktion stellt eine vereinfachte Noise-Funktion dar, die einen beliebigen Integer auf einen Floatingpoint-Wert abbildet. Die Gradientenberechnung ist in den drei 'x' enthalten. Die Formel für das verwendete Polynom ist $3t^2 - 2t^3$.

```
float noise(x: int) {
        x = (x<<13) ^ x;
        return ( 1.0 - ( (x * (x * x * 15731 + 789221) +
                1376312589) & 0x7fffffff) / 1073741824.0);
}
```

Hier wurde nur ein Wert berechnet, dieser muss dann auf das zu erzeugende Array angewendet werden und dort zwischen zwei diskreten Eingangswerten interpoliert werden. Typischerweise wird in 1D linear, in 2D bilinear und in 3D trikubisch interpoliert.



Die Frequenz der Pseudozufallswerte wird dabei beschrieben durch:
$frequency = \frac{1}{wavelength}$

2.2.2 Berechnung von 1D-Perlin Noise

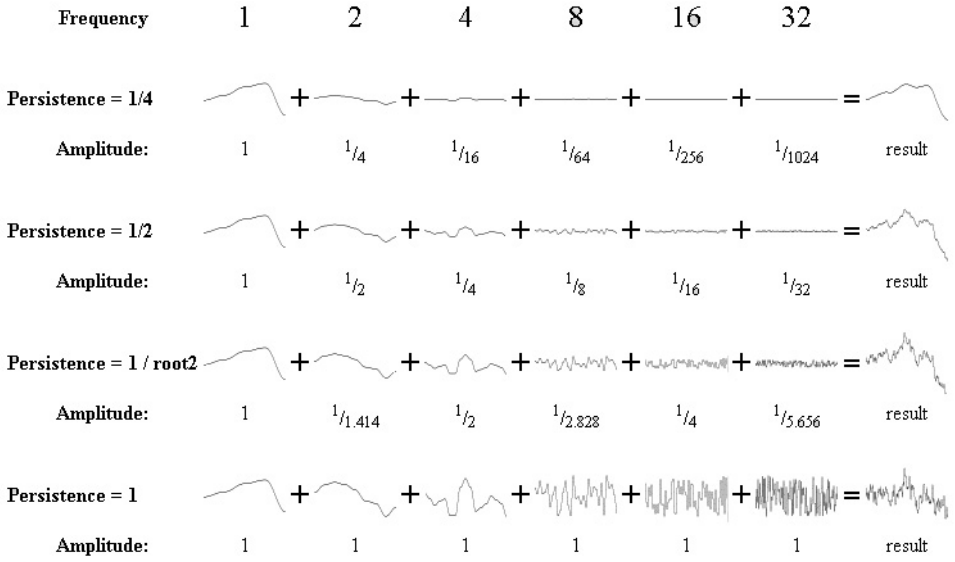
In der folgenden Abbildung ist in der letzten Spalte jeweils das Ergebnis zusehen. Die Noise-Funktion wird um das Ergebnis zu erzeugen mehrfach (hier sechs Mal) berechnet. Die einzelnen berechneten Funktionen werden dabei aufsummiert.

i läuft im Beispiel von 0 bis 5

$fequency = 2^i$

$amplitude = persistence^i$

Die Persistenz wird dabei festgesetzt, je nachdem wie 'zerklüftet' oder 'weich' die Ergebniskurve sein soll.



2.2.3 Erweiterung auf mehrere Dimensionen

Zur Erweiterung von 1D-Perlin Noise auf mehrere Dimensionen werden für die n-te Dimension 2^n Gradienten je Punkt im nD-Raum benötigt. Daraus ergibt sich für Perlin-Noise eine Komplexität von $O(2^n)$ bezüglich der Dimension. Was in der ersten Dimension noch keine große Herausforderung darstellt wird in der für interaktive Anwendungen typischen vierten Dimension ein ernst zu nehmendes Problem.

Dort sollen normalerweise 3D-Objekte wie Planeten oder Wolken auftreten, die um die vierte Dimension erweitert sind um diese zu animieren. Die Noise-Funktion besitzt dafür vier Parameter, von denen der letzte innerhalb eines Renderingdurchganges statisch ist. Die restlichen Parameter beschreiben einen Eckpunkt eines Kubus im 3D-Raum. Sind werden in allen drei Koordinaten Interpoliert.

3 Rendering

3.1 1D-/2D-Noise

1D- oder 2D-Noise wird typischerweise nicht für interaktive Anwendungen verwendet, sondern eher zum erzeugen statischer Bilder. Bei diesen stellt die benötigte Rechenzeit kein Problem dar, da meist mehr als der Bruchteil einer Sekunde zur Verfügung steht und die Komplexität des Algorithmus hier nicht so sehr wirkt.

Ein Beispiel von 2D-Perlin Noise kann in diversen Grafikbearbeitungsanwendungen betrachtet werden. In einer namenhaften Anwendung ist es als Filter unter dem Namen 'Wolken' verfügbar.

3.2 Verfahren

Alle folgenden Renderingverfahren haben gemeinsam, dass für unendliche Strukturen, wie Funktionen es sind, ein adaptives LOD notwendig ist.

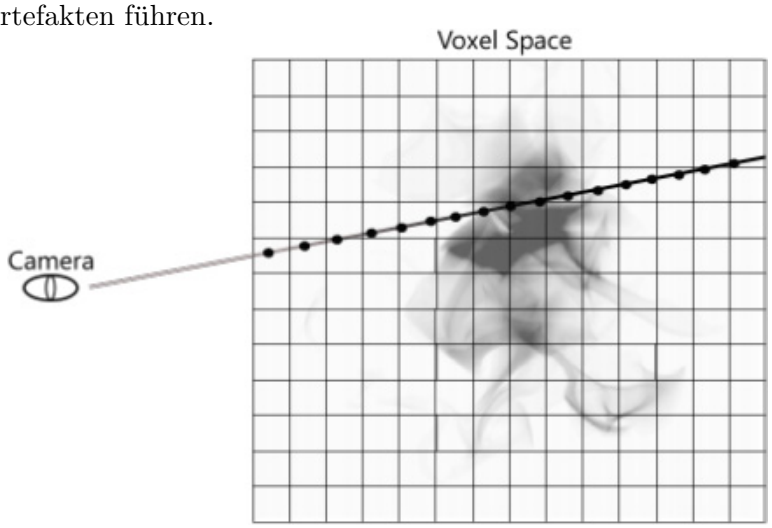
Direktes Raytracing zu verwenden um eine Noise-Funktion zu berechnen gestaltet sich sehr schwierig, da es nicht möglich ist schnell den exakten Schnittpunkt einer Graden mit einer Pseudo-Zufallsfunktion zu berechnen. Dies ist typischerweise nur durch Annährung möglich.

3.2.1 Volumenrendering

Volumenrendering bezeichnet einen Raytracingansatz, der darauf basiert die zu rendernde Szene in gleichgroße Kuben zu unterteilen. Diese Kuben werden als Voxel bezeichnet. Die zu rendernden Objekte oder Strukturen liegen dabei als Dichtefunktion vor, so dass für jeden Voxel angegeben werden kann welche Dichte dieser hat. Also wie viel des einfallenden Lichtes er absorbiert.

Zum Rendern wird nun ein Strahl durch das Voxel-Volumen simuliert und die Beleuchtung jedes Voxel wird akkumuliert. Durch die Akkumulation steht der Beitrag des Volumens zur Intensität des Pixels fest.

Dieses Verfahren eignet sich besonders um gasartige Strukturen zu rendern. Viele kleine Voxel machen das Ergebnis diese Verfahrens sehr genau, allerdings benötigt es auch entsprechenden Rechenaufwand. Zu große Voxel können zu harten Kanten und Artefakten führen.

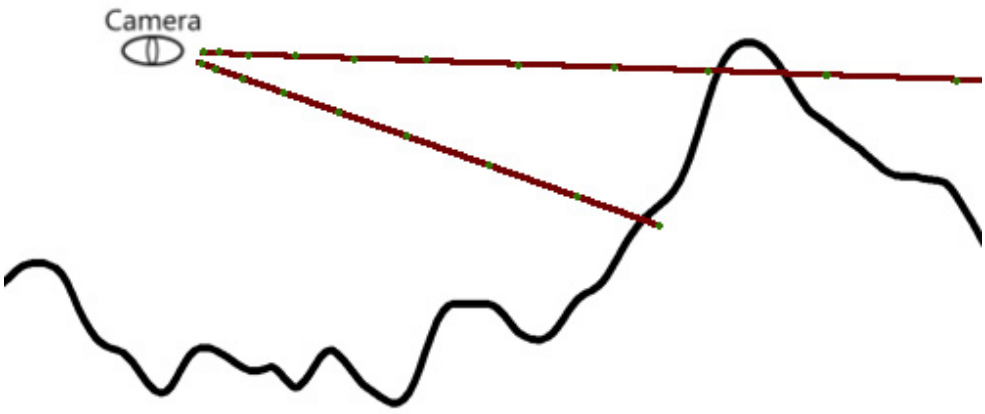


3.2.2 QAEB-Rendering

QAEB-Rendering ist wie Volumenrendering ein Raytracingansatz und steht für 'Quasi-Analytic Error-Bounded'. Dieser Ansatz simuliert ebenfalls einen Strahl, benötigt aber keine vorher generierten Voxel, sondern führt in diskreten Abständen Schnitttests mit der Noise-Funktion durch. Diese Abstände werden vom Augpunkt ausgehend größer, werden aber nur soweit erhöht, dass ein bestimmter Fehler nicht überschritten wird.

Bei der Berechnung des Schnitttest wird ermittelt ob die Funktion für den diskreten Wert innerhalb oder außerhalb des Objektes ist. Wenn zwischen zwei Schnittest ein Vorzeichenwechsel des Ergebniswertes stattgefunden hat, liegt zwischen diesen Punkten die Oberfläche.

Der Nachteil dieses Verfahrens ist, dass sie sehr langsam ist, wenn der Fehler klein gewählt ist. Wird der Fehler größer gewählt, wird das Ergebnis ungenauer und es können Artefakte auftreten, wie z.B. löcher in Bergspitzen oder freischwebende Bergspitzen.



3.2.3 Polygonalisieren

Dies ist das am meisten verwendete Verfahren, da die genannten Raytracingansätze zu Aufwändig sind und Standardcomputer aktuell über kein hardwarebasiertes Raytracing verfügen.

Hauptsächlich werden zwei Verfahren zusammen verwendet um eine Szene, Struktur oder ein Objekt zu Polygonalisieren. Dies sind zum Einen die Subdivisions, die im Dreidimensionalen durch Octrees realisiert werden. Und zum Anderen das Marching Cubes Verfahren.

Subdivisions werden wie beim Volumenrendering verwendet um die Szene in Blöcke zu unterteilen. Diese Blöcke sind nun nicht mehr gleich groß sondern können abhängig von der Entfernung des Szenenpunktes zum Betrachter in mehrere oder weniger viele Blöcke unterteilt werden. Ein Block wird dabei immer in acht Teilblöcke zerlegt. Anschaulich kann das gemacht werden, durch einen Kubus durch dessen Mitte im dreidimensionalen drei Ebenen gelegt werden. Diese Ebenen liegen jeweils Orthogonal zu einer Koordinatenachse des Kubus vor.

Durch die Subdivisions wird ein adaptives LOD erzeugt, das eigentliche Polygonalisieren übernimmt dann das Marching Cubes Verfahren. Bevor die Polygonalisierung starten kann müssen für alle Eckpunkte eines zu polygonalisierenden Kubus die Funktionswerte der Noise-Funktion berechnet werden. Dann wird ermittelt an welchen von den zwölf Kanten des Kubus die Oberfläche der Noise-Funktion schneidet. Ist dies geschehen wird zwischen den Punkten linear interpoliert und somit approximativ der Schnittpunkt der Funktion mit der Kante des Kubus bestimmt. Dieser ermittelte Punkt stellt einen Vertex eines Polygons dar.

Das Verfahren ist sehr stabil und da das Rendern der erstellten Vertices auf der Grafikkarte ausgeführt werden kann im Gegensatz zu den anderen Verfahren sehr schnell. Das Problem ist, dass dieses Verfahren immer noch sehr viel Rechenaufwand benötigt und natürlich bei einem zu geringen LOD Kanten sichtbar sein können. Zudem müssen die Normalen noch berechnet werden.

4 Echtzeit

4.1 Problem und Einschränkung

Das größte Problem den Noise-Algorithmus für interaktive Anwendungen zu benutzen ist die Komplexität des Algorithmus und damit auch verbunden der benötigte Rechenaufwand. Natürlich könnte der Algorithmus auf einem Supercomputer ausgeführt werden und von Ken Perlin existiert eine Schemaskizze eines Hardwaremoduls, dass in die Renderingpipeline mit eingebunden werden könnte.

Normalerweise wird nur niemand einen Supercomputer zu hause stehen haben oder sich eine Grafikkarte herstellen können.

Ebenso ausgeschlossen werden hier kleine animierte Texturen (z.B. 256x256 Pixel), die mit aktuellen Computern keine Herausforderung mehr darstellen.

4.2

5 Beispielimplementierung aus GPU Gems 3

5.1 Nvidia

6 Fazit

6.1 Möglichkeiten

6.2 Ausblick

möglichkeiten aktueller grafikkarten noch nicht annährend ausgereizt/erforscht
nvidia raytracing für interaktive anwendungen

Literaturverzeichnis

[1] K. Perlin, (2002) *Improving Noise*, New York University

[2] J. C. Hart, (2001) *Perlin Noise Pixel Shaders*, University of Illinois

[3] S. Gustavson, (2005, März) *Simplex noise demystified*, Linköping University, Schweden

[4] K. Perlin, *Noise Hardware, Chapter 2*

[5] N. Brickman, D. Olsen und G. Smith, *Realtime Cloud Simulation and Rendering*, UCSC

[6] M. Olano, (2005, Juli) *Modified Noise for Evaluation on Graphics Hardware*, UMBC

[7] J. R. Frisvad und G. Wyvill, (2007, Dezember) *Fast High-Quality Noise*, TU of Denmark, University of Otago

[8] N. A. Carr und J. C. Hart, (2002, April) *Meshed Atlases for Real-Time Procedural Solid Texturing*, University of Illinois

[9] M. Kameya und J. C. Hart, *Bresenham Noise*, Washington State University

[10] A. Goldberg, M. Zwicker und F. Durand, (2008, August) *Anisotropic Noise*, UCSD

[11] R. L. Cook und T. DeRose, (2005) *Wavelet Noise*, Pixar Animation Studios

[12] J. Laeuchli, (2003) *Graphics Programming Methods*, *Real-Time Generation and Rendering of 3D Planets*

[13] S. Green, (2005) *GPU Gems 2*, *Implementing Improved Perlin Noise*, NVIDIA Corporation

[14] N. Wang, (2004) *Perlin Noise and Cloud Rendering*, Concordia University

[15] K. Perlin, (1999, Dezember) *Making Noise*, [online]
`http://www.noisemachine.com/talk1/`

[16] H. Elias, *Perlin Noise*, [online]
`http://freespace.virgin.net/hugo.elias/models/m_perlin.htm`

[17] K. Perlin, (1985) *Noise and Turbulence*, [online]
`http://mrl.nyu.edu/~perlin/doc/oscar.html#noise`

[18] K. Perlin, (2002) *Improved Noise reference implementation*,
`[online] http://mrl.nyu.edu/~perlin/noise/`

[19] P. Bourke, (2000, Januar) *Perlin Noise And Turbulence*, [online]
`http://local.wasp.uwa.edu.au/~pbourke/texture_colour/perlin/`

[20] D. Shiffman, *Random Nimbers, Probability, Perlin Noise*, [online]
`http://www.shiffman.net/teaching/nature/week-1/`

[21] P. Warden, (2005, Mai) *Animating Worley Noise*, [online]
`http://petewarden.com/notes/archives/2005/05/testing.html`

[22] F. Brebion, *Infinity Universe*, [online]
`http://www.infinity-universe.com/`

[23] H. Elias, *Cloud Cover*, [online]
`http://freespace.virgin.net/hugo.elias/models/m_clouds.htm`

[24] R. Geiss, (2007) *GPU Gems 3*, *Generating Complex Procedural Terrains Using the GPU*, [online]
`http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html`

[25] D. Kristjansson, (1999) *Evolving Planet*, [online]
`http://mrl.nyu.edu/~kristja/planet/`

[26] K. Perlin, *A sheet of simplex noise*, [online]
`http://mrl.nyu.edu/~perlin/homepage2006/simplex_noise/index.html`