



I'll begin with an example of coding a real procedural content generator in 2D from scratch, before circling back to talk about the why, what, and how of "procgen" at a high level.

From there we'll look at ray tracing and implicit surfaces and end up by creating a 3D model of a cute little planet.

At the end you'll have resources and ideas to start your own experiments in this space.

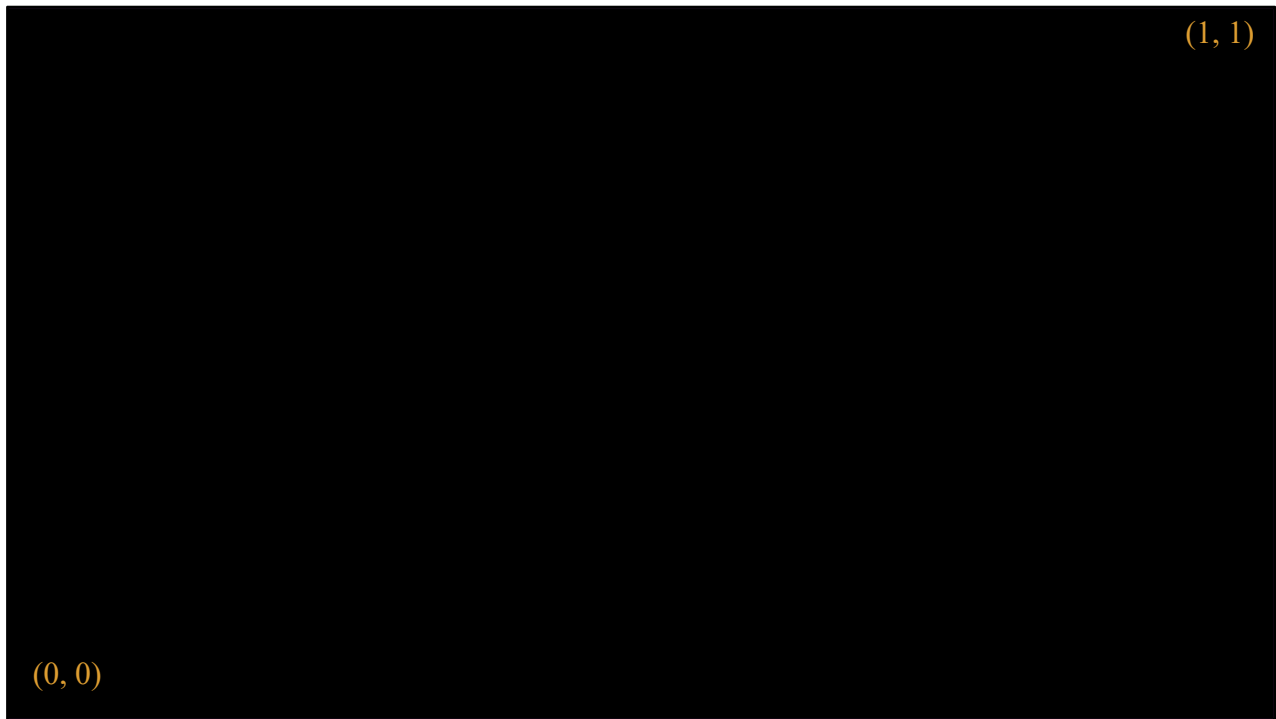


CREATING A UNIVERSE FROM RULES

A lot of graphics content development is done by artists, programmers, and writers placing each piece of their virtual universe and painting, animating, scripting, etc. it by hand.

The alternative is to not create anything except the laws of physics for your universe, and then letting the laws and some initial conditions create all of the richness. That's procedural generation.

Watch me create such a universe right now:



In the beginning, we have formless darkness. That's how every graphics program starts, with this black screen.

In my coordinate system, the origin is the lower-left and the upper right corner is (1, 1)

```
void mainImage(out Color color, in Point coord) {  
    float x = pixel.x / iResolution.x;  
    float y = pixel.y / iResolution.y;  
    color = Color(0.0);  
}
```

Here's the code for a GPU pixel program that draws the black screen. If you're not used to writing graphics code, then I just have to tell you a few simple things to understand this:

- "float" is a 32-bit real number. As in, a value with a decimal point.
- "Color" has red, green, and fields
- "Point" is a 2D point class with X and Y fields
- X and Y FOR loops run this code at every pixel to produce an image. Those are implicit and handled by the basic full-screen shader setup.

This program creates a universe. It is all mine. But I'll share it with you.

Unfortunately, it is really, really boring.

```
void getSkyColor(float x, float y, inout Color color) {
    float h = max(0.0, 1.4 - y - pow(abs(x - 0.5), 3.0))
    color.r = pow(h, 3.0);
    color.g = pow(h, 7.0);
    color.b = 0.2 + pow(max(0.0, h - 0.1), 10.0);
}

void mainImage(out Color color, in Point coord) {
    ...
    getSkyColor(x, y, color);
}
```

Here's some code to compute a gradient color. Don't worry about exactly how it works. I basically fiddled around making the red, green, and blue fields decrease with the Y coordinate value until it looked pretty.

I'll run it. Let there be light...



Yeah! Now we have some computer graphics. Our universe has a rule for how bright every point is.

It is pretty. But still kind of boring. So, let's divide the earth from the sky...

```
float terrain(float x) {  
    return 0.45;  
}  
  
void mainImage(out vec4 color, in Point coord) {  
    ...  
    if (y < terrain(x)) { color = color(0.0); }  
}
```



What I'll do is say that there is a line of terrain at $y=0.45$, just a little under halfway up the screen.

Everything below the terrain turns black, and the rest has the pretty sky gradient.

```
float terrain(float x) {  
    return 0.45;  
}  
  
void mainImage(out vec4 color, in Point coord) {  
    ...  
    if (y < terrain(x)) { color = color(0.0); }  
}
```

Now we have a silhouette of the terrain. Which is a flat line, but that's a start.


```
float terrain(float x) {  
    return noise(x) * 1.2 + 0.36;  
}
```

```
float noise(float x) {  
    float i = floor(x), f = fract(x);  
    float u = 0.5;  
    return 2.0 * mix(hash(i), hash(i + 1.0), u) - 1.0;  
}
```

I'm going to change the elevation by just computing a hash function on the horizontal position so that there is variation.


A hash function just turns each X value into an effectively random Y value that will be the height of the terrain, except that it is stable: if you give it the same input, then you get the same output.

So that this doesn't zig and zag at each pixel like crazy, I'll change the value in big horizontal blocks of X. I can do this by using the FLOOR function to round down the x position before I take the hash.

There's some scaling and offset in the terrain() function to keep the line from going offscreen. Here's what we get:

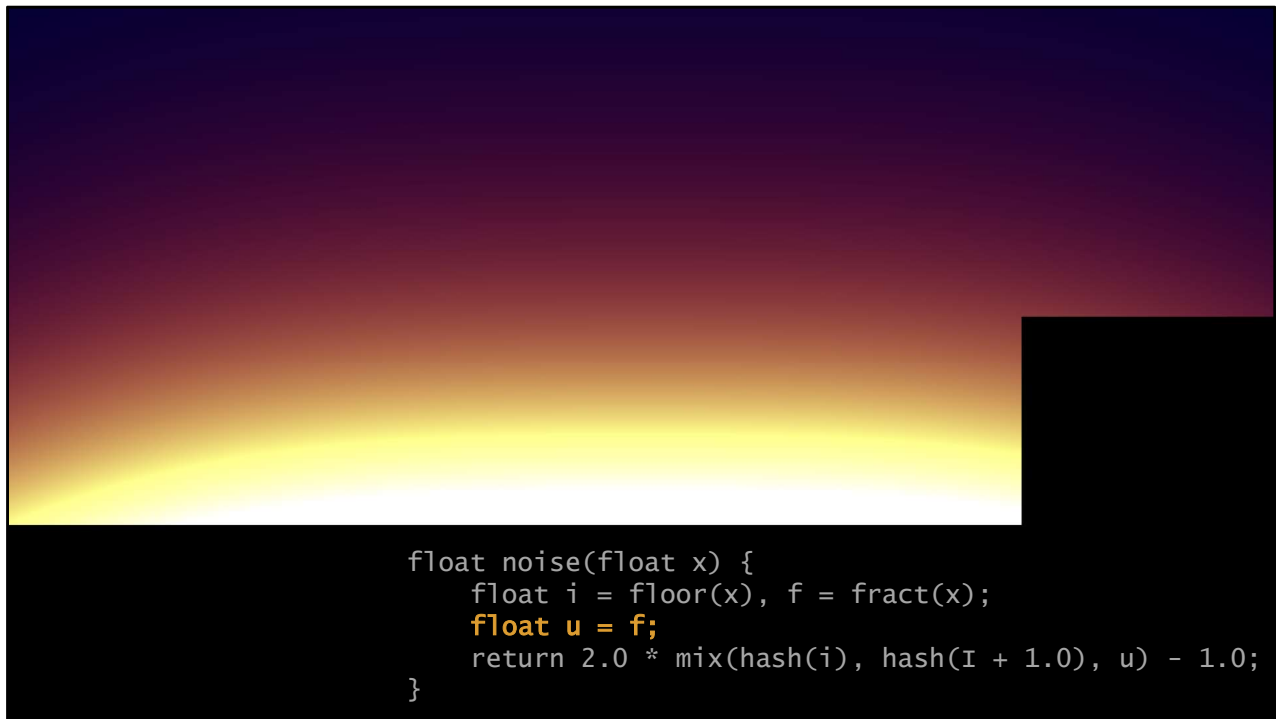
```
float terrain(float x) {
    return noise(x) * 1.2 + 0.36;
}

float noise(float x) {
    float i = floor(x), f = fract(x);
    float u = 0.5;
    return 2.0 * mix(hash(i), hash(i + 1.0), u) - 1.0;
}
```

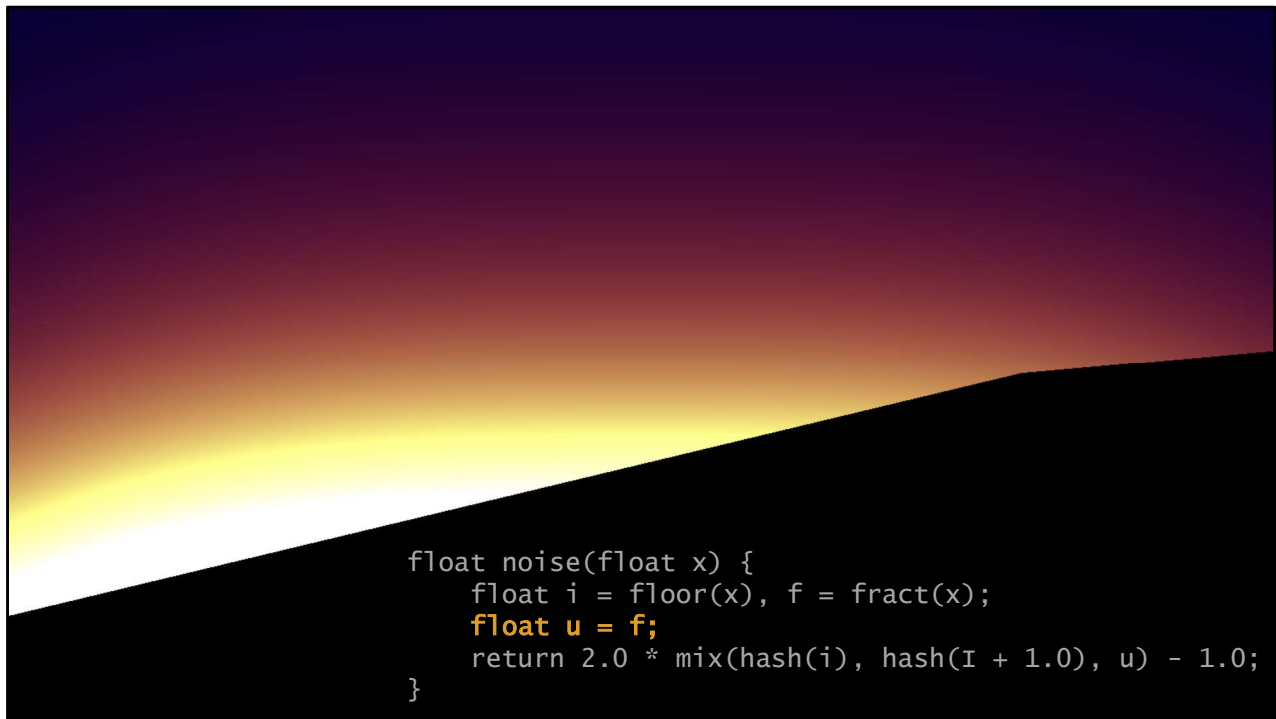


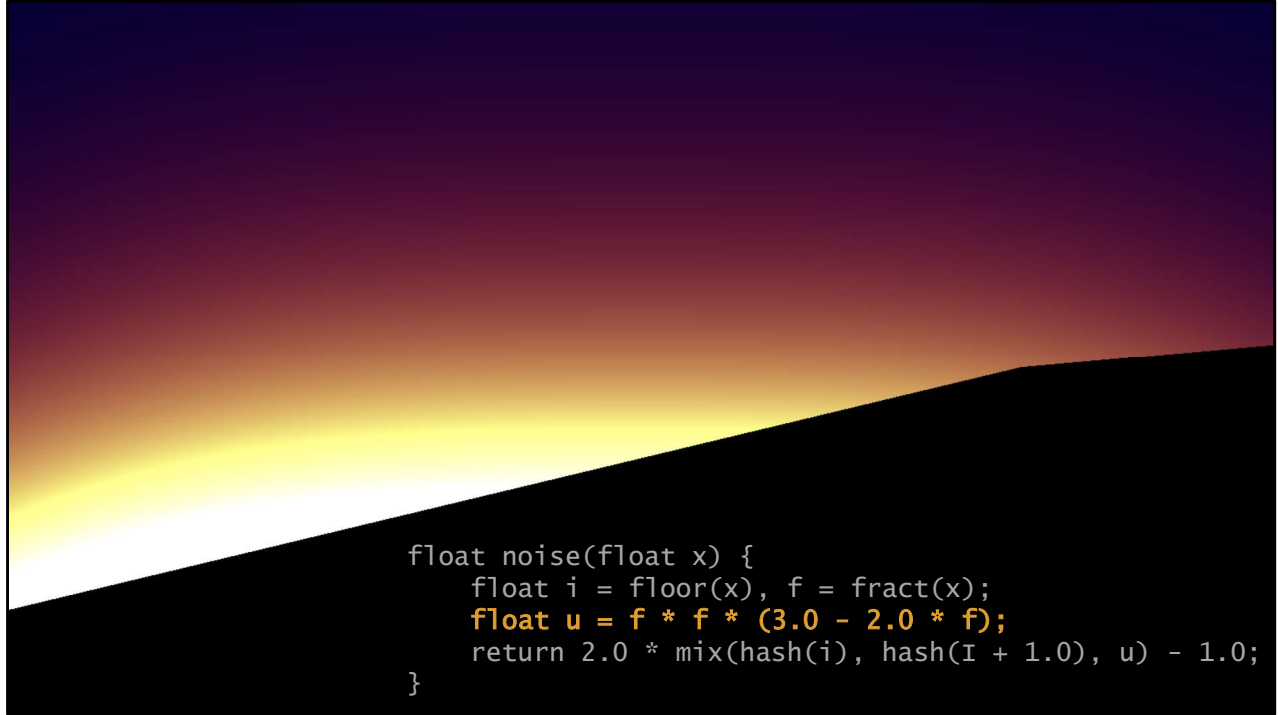
Now there is variation. Well, since I rounded off the horizontal value to the nearest integer and the entire screen just goes from 0 to 1 horizontally, there is only one transition.

We've come a long way. But this is where the fun really starts.



This doesn't look very natural. It is maybe a bad approximation of a city. Let's smooth out the jump in the elevation by using a linear interpolation between two hash values to produce a hill instead of a cityscape.

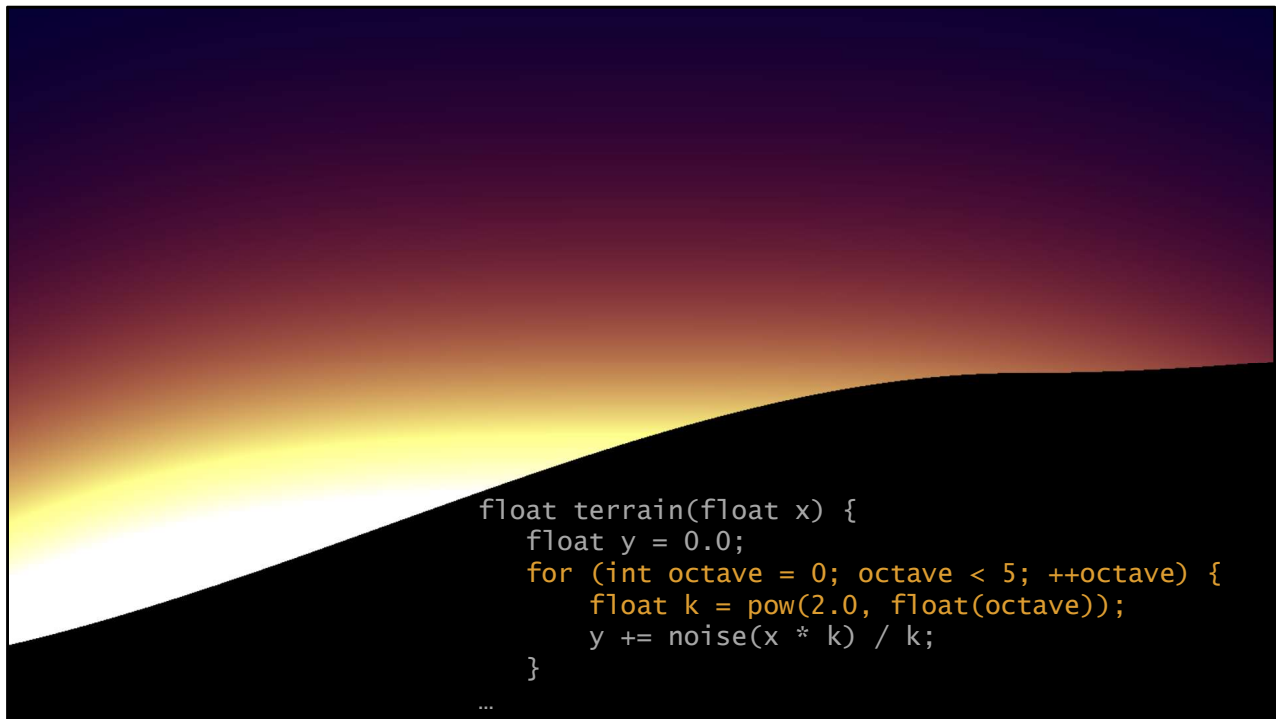




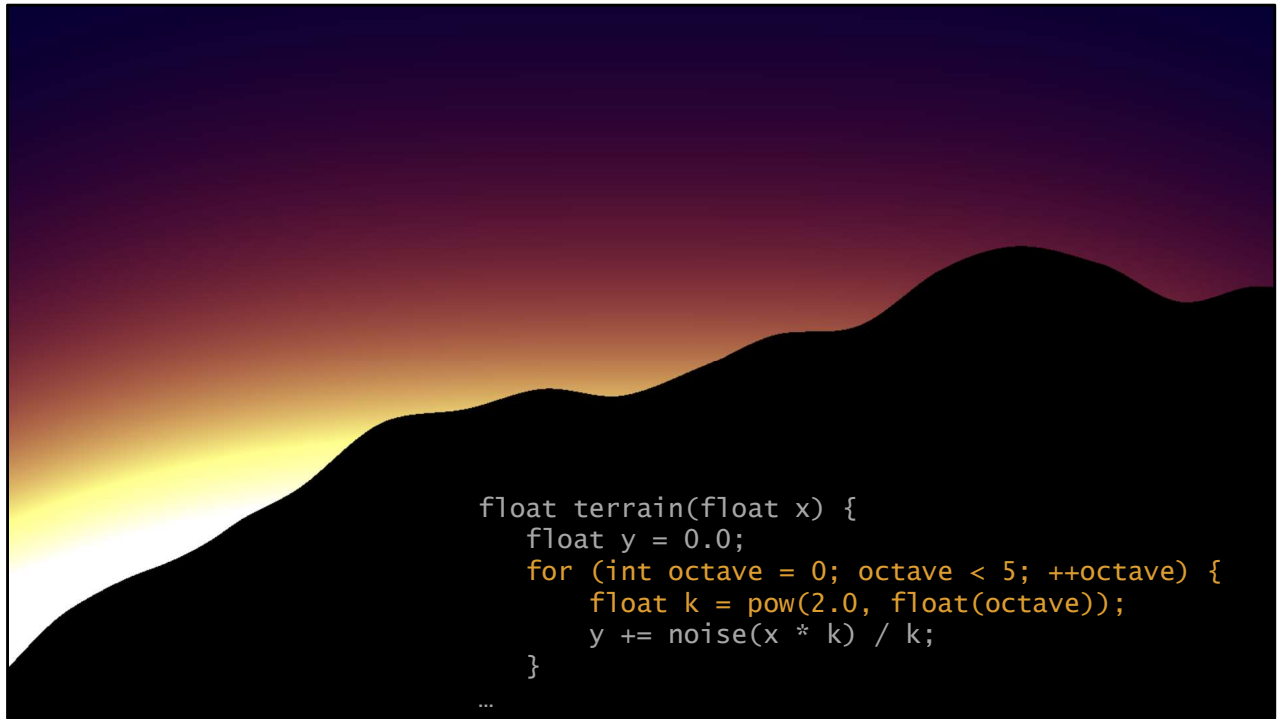
This linear terrain looks very much like low-polygon computer graphics. You can see the sharp corner and unnaturally straight line. So, instead of linear, let's make it a 3rd order curve...



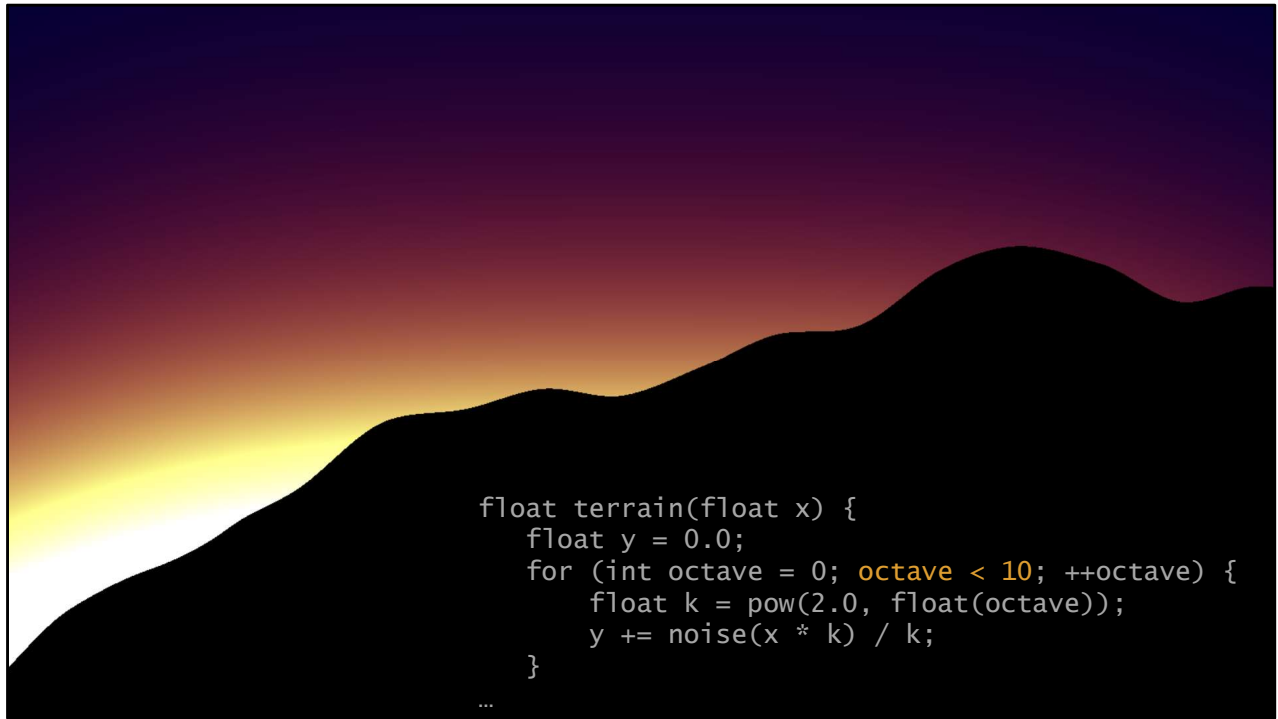
That is much more natural! A gently sloping hill.



Now, let's not only use the noise function, but instead use the sum of five noise functions that have exponentially decreasing period and amplitude, like octave harmonics in music.



Much more detail. We are really getting somewhere.



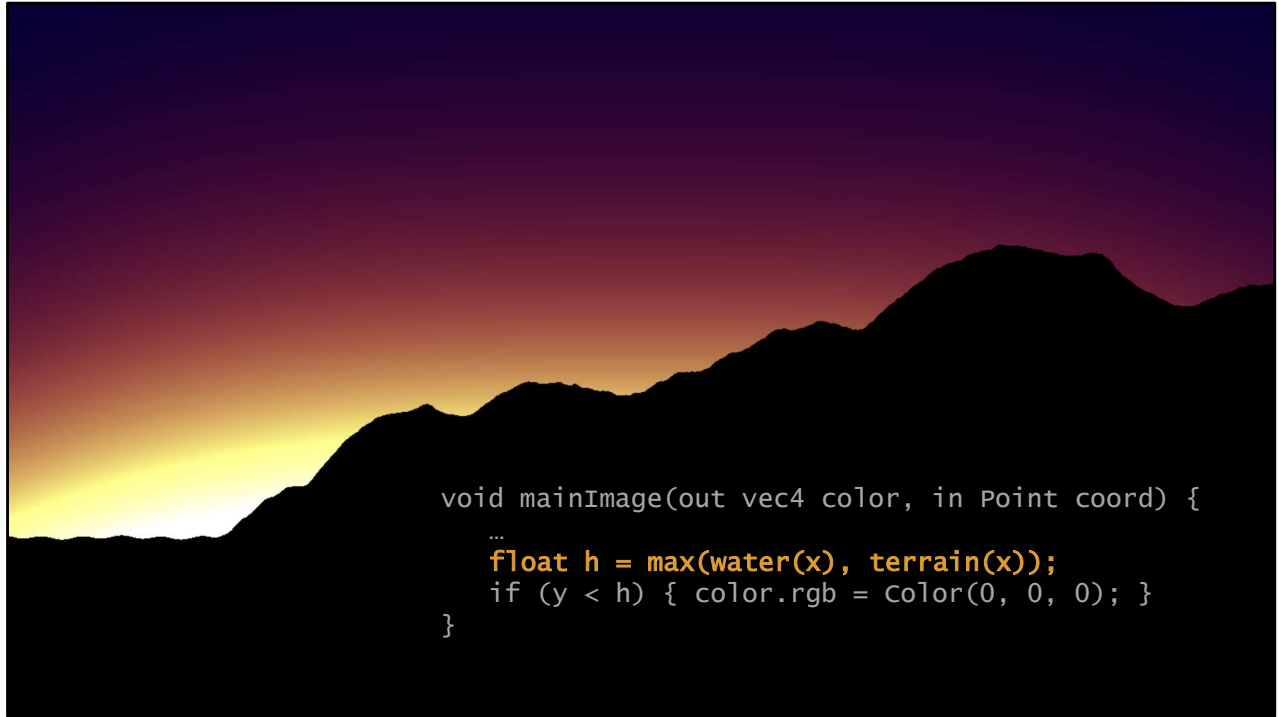
And if we increase the number of octaves some more...



Now there is a lot of detail. This looks like a hill or mountain range at sunset.



Let's go back to the main function. If instead of treating the silhouette line as the terrain height, I instead make it whichever is larger: some sine waves or that terrain height...



Then I get animated ocean waves rolling in to the shore.



And if I add this line of code adjusting the silhouette line again, then the terrain rises up by another noise function in some sharp peaks...



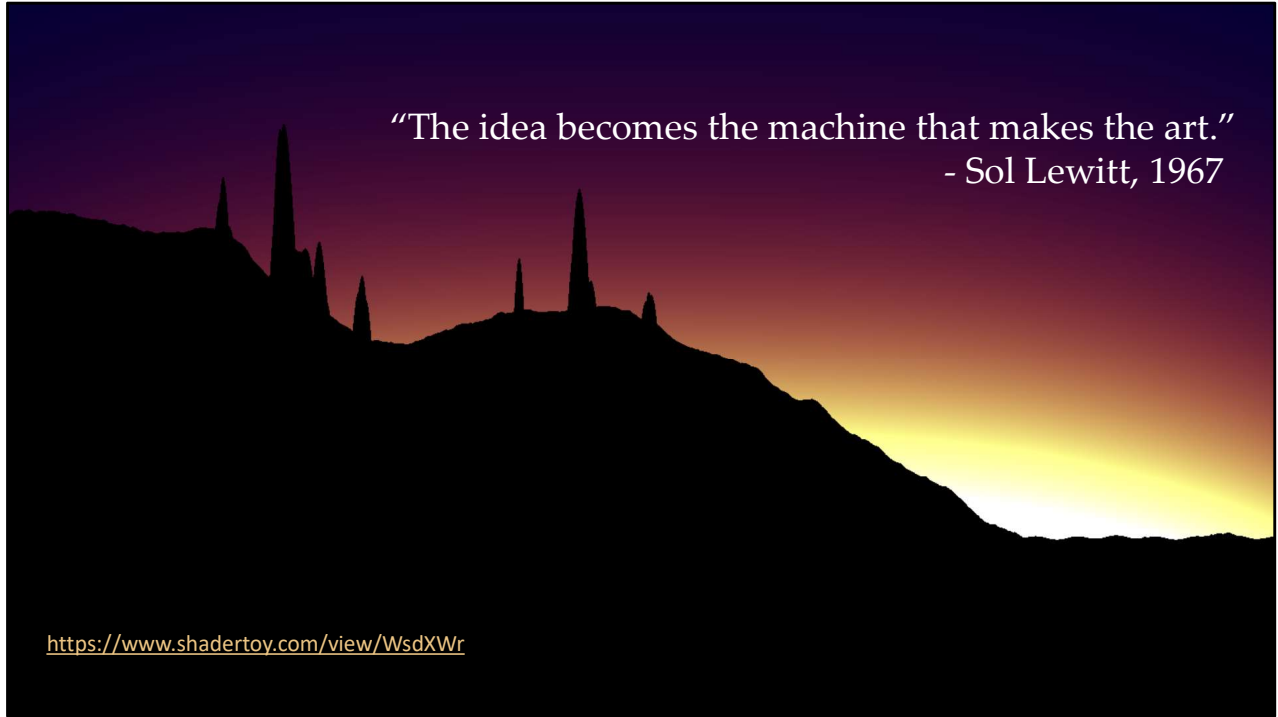
And we get trees!

```
void mainImage(out vec4 color, in Point coord) {  
    ...  
    float shift = 0.09 * iTime + 0.2;  
    x += shift;  
    ...  
}
```

Now, here's the *real* payoff. If I just start scrolling the viewpoint by making the horizontal coordinate increase with time, we can travel around and view this world that we've built. And discover new things in it, because it is only bounded by numerical precision...and we've never seen anything except the one starting location.

Think about this for a moment. I did NOT create the specific islands, ocean, and trees. I created the rules of the universe, and those rules created the scene.

In the words of my favorite conceptual artist, Sol Lewitt, "The idea becomes the machine that makes the art".



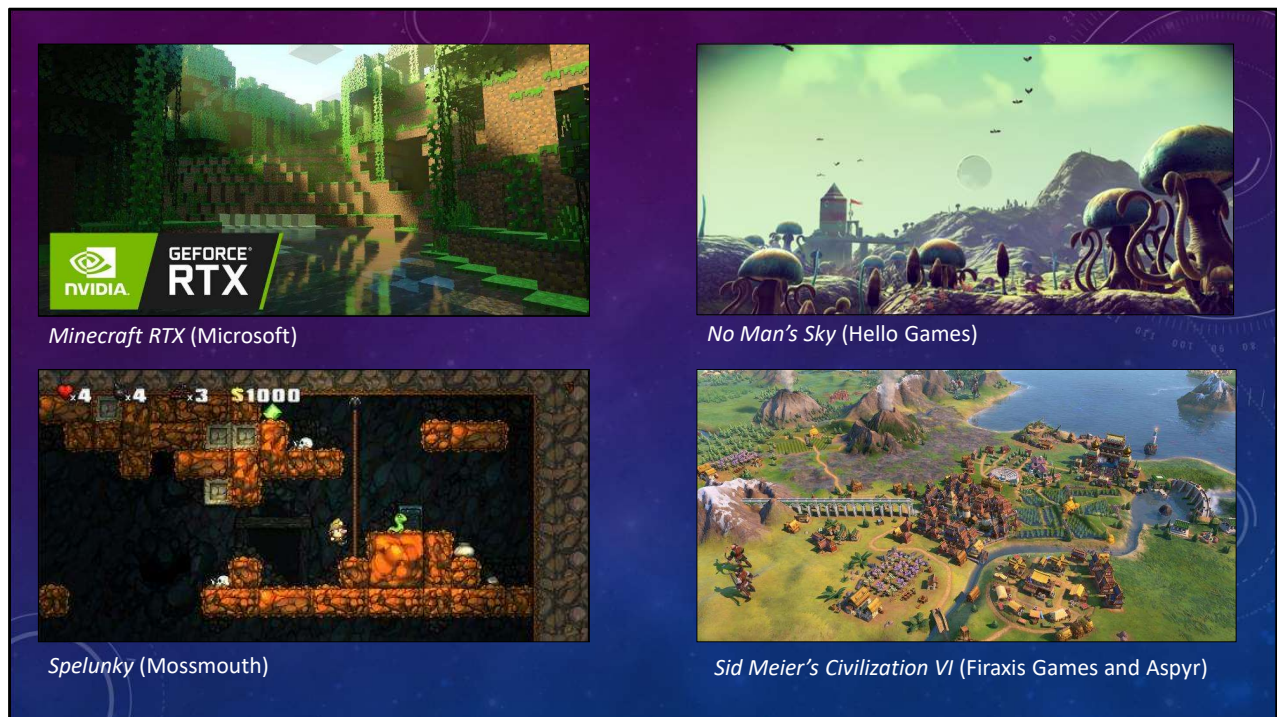
All of my code for this little demo is at that URL, and you can edit it and run it yourself in a web browser.



PROCEDURAL WORLDS

You just saw procedural content.

Now I'll take a step back and talk about the big ideas of procedural content generation in games to generalize from this example.



This is the kind of result we want to achieve. Rich content with unlimited variations and scope created automatically by code. There's also varying amounts of manual artwork injected as primitives into these systems, but for the purpose of creating the "world" or the "level"/"map", that's all coming from the code without anyone touching each forest or planet.

I'm pointing out these games specifically because you're probably familiar with them and they're all similar to our goal today: creating a procedural LANDSCAPE.

Akalabeth: World of Doom (1979) was probably of the first to create most aspects of the world from a single random number seed; Rogue was developed contemporaneously and similarly relied on procgen, and led to the "Roguelike" genre. The modern version of those is Dwarf Fortress, which is THE most aggressive and deepest procgen/world simulator today. But you'll find procedural generation in lots of indie titles, such as 20XX to Terraria.

Because of the rising expectations of players for both fine scale detail and giant worlds, we're starting to see procedural content generation go mainstream for all aspects of games: **modeling, texture, geometry, weather, audio, narrative,**

conversation, text, lighting, animation, character design, and full level design with gameplay implications. This is sometimes “offline” (or maybe triggered when the player first sees an area) and sometimes happening at runtime continuously as a reaction to game events.

I think that with machine learning as a viable new computer science tool, procedural content should be the common case instead of an exception in the future, and is an important area for research and development.

PROCEDURAL CONTENT STRATEGY



Here's a general strategy for procedural content.

Take some small or noisy INPUT signal

EXPAND it by the procedural core

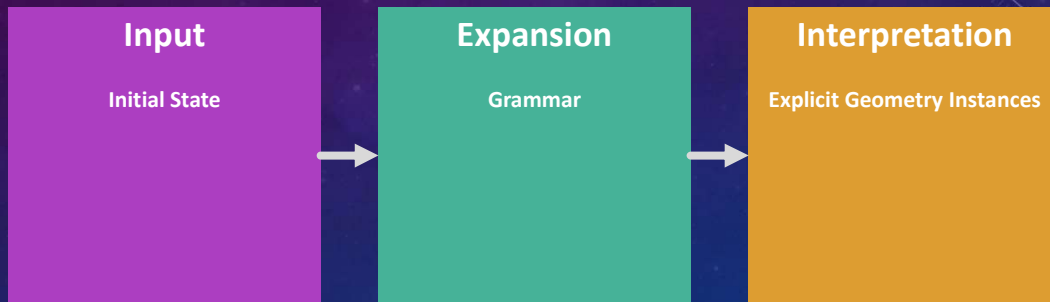
And then INTERPRET the result in a meaningful way for the game



You should be further inspired that one of the leading tools in procedural content generation is Houdini by SideFX, which was founded by University of Waterloo alumni and is located nearby in Toronto.

The key to most procedural content is combining the creation of detail by an algorithm with a tool that allows control of the content at a high level. Unreal and Unity provide tools for this. Houdini is one of the industry standards for both real-time game procedural content and offline. For example, here this city is almost entirely created procedurally in Houdini.

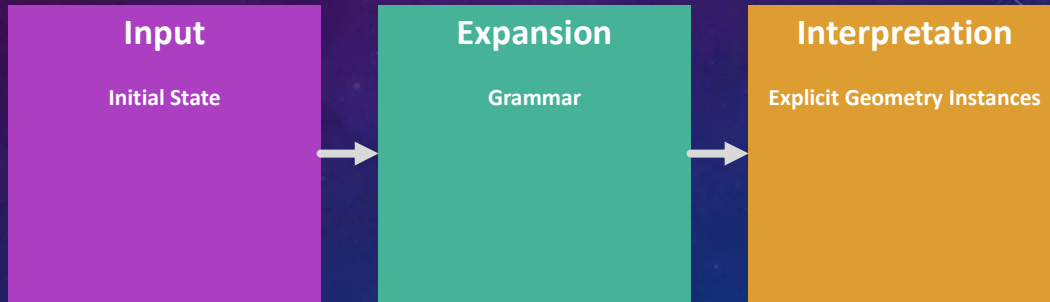
PROCEDURAL CONTENT STRATEGY



Houdini SideFX via PaQ WaK tutorial

The Houdini city example used an initial state, expanded by a GRAMMAR, and interpreted as explicit GEOMETRY instances

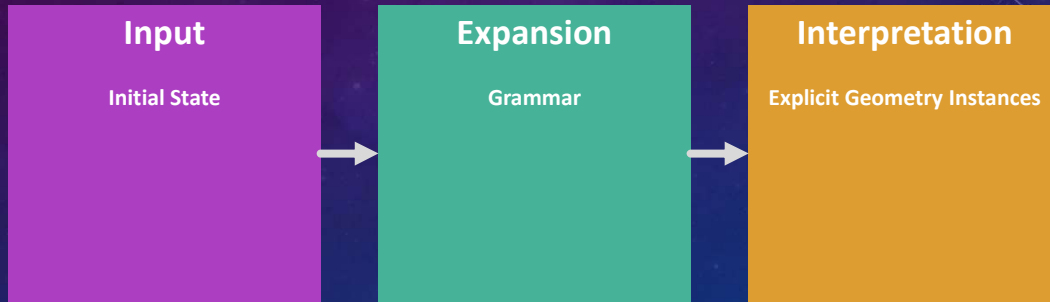
PROCEDURAL CONTENT STRATEGY



Transformations in architectural scenes, Muller et al. 2005

This is a very popular technique for buildings and trees...anything with self similarity.

PROCEDURAL CONTENT STRATEGY



Speed Tree by Interactive Data Visualization, Inc.

NVIDIA GAUGAN DEMO



Try it live online at <http://nvidia-research-mingyuliu.com/gaugan/>

Details: Park et al., Semantic Image Synthesis with Spatially-Adaptive Normalization, CVPR 2019

The cutting edge of procedural content creation today is using machine learning/AI for the expansion. Here's an example from NVIDIA that works entirely in 2D which won the Best in Show at SIGGRAPH 2019 real-time live.

You paint big regions as shown on the left, and a Generative Adversarial Network algorithm produces a photorealistic 2D image of a 3D scene on the right. The 3D scene never actually exists in this case. Notice that all of the detail is "hallucinated" by the algorithm...you say where the trees and water are but don't have to specify the details.

Tools are absolutely the right solution for production work on a team with artists. But today we're going to focus on the code itself for procedural content. The idea here is to create art using only code, where we don't even have to specify regions for trees and water but let them emerge themselves. What is cool about the 100% code approach is that it can create scenes that look good but where you as the creator can be surprised by what you discover within them, since you didn't steer it at all at a low level.

That is, you create the rules of the universe, but let those rules create everything

within it for you.

PROCEDURAL CONTENT STRATEGY



Semantic Image Synthesis with Spatially-Adaptive Normalization Park et al. 2019

GAUGAN used a painted mask, expanded by ML inference, and interpreted the output as pixel colors



The rest of the examples I'm going to show you today are more like the planet part of No Man's Sky. They use:

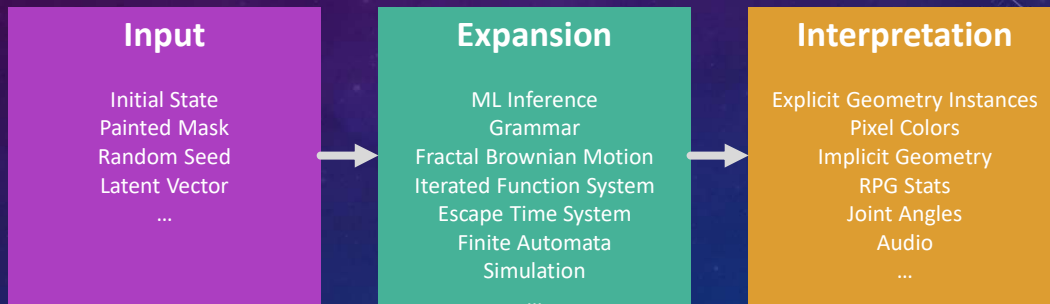
Random Seed input

Which is expanded by multiple octaves of pseudorandom noise

And interpreted as implicit 3D geometry

There are plenty of other alternatives...

PROCEDURAL CONTENT STRATEGY



And here are some other tools for each of these boxes

SOME PROCEDURAL CONTENT KEYWORDS

- Wave Function Collapse (<https://github.com/mxgmn/WaveFunctionCollapse>)
- Boids – Flocking Simulation
- Value Noise, Perlin Noise, Worley Noise
- L-System
- Fractal Brownian Motion (FBM)
- Voronoi
- Cellular Automata, Reaction-Diffusion
- Rogue-like
- Kruskal's Algorithm, Prim's Algorithm
- Fractals: Mandelbulb/Mandelbrot Set, Apollonian Cube/Gasket, Menger Sponge, Iterated Function Systems, Escape Time Algorithm, Limit Set

Each section of this slide deck has some keywords and links for you to look at afterwards to follow up for your own exploration.

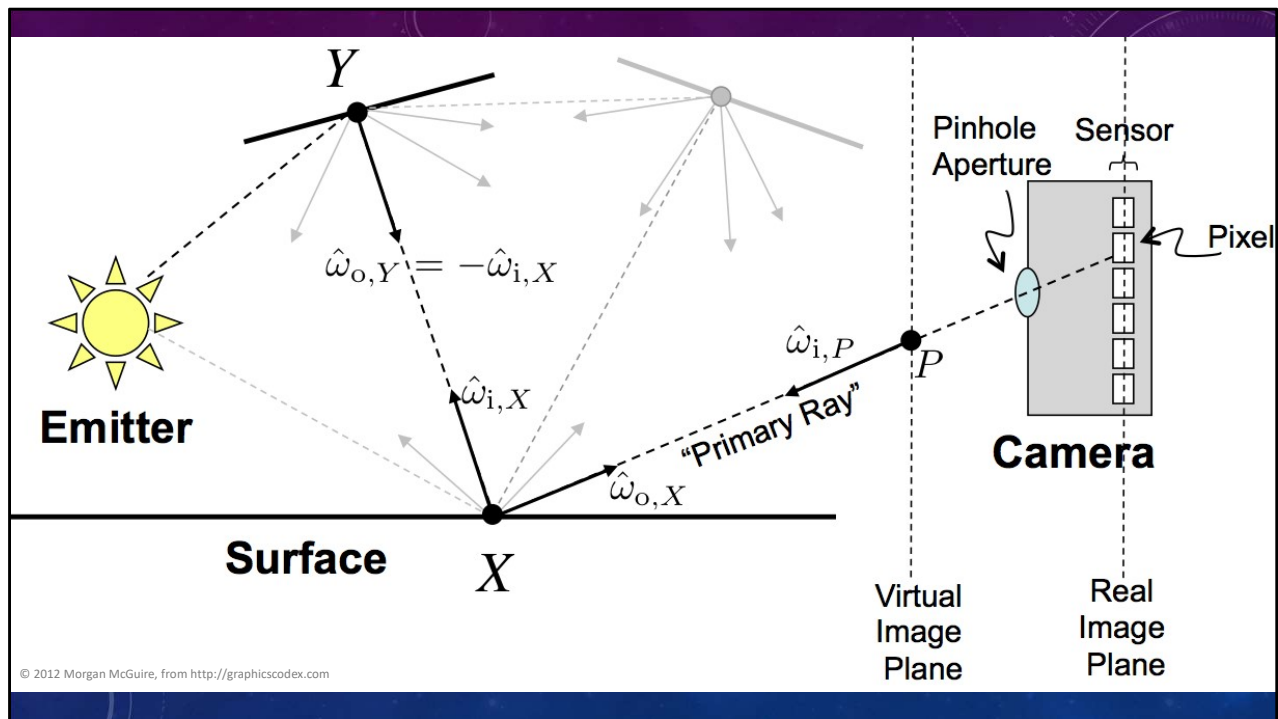
SOME PROCEDURAL CONTENT RESOURCES

- PROCJAM (Nov 2019) <http://www.proccjam.com/>
- Spelunky, Derek Yu 2016 (the book) <https://bossfightbooks.com/products/spelunky-by-derek-yu>
- Procedural Generation in Game Design, Short & Adams 2017 <https://amzn.to/33WMJNV>
- Texturing and Modeling, Ebert et al. 2002 <https://amzn.to/32u8IRI>
- Amit Patel <https://www.redblobgames.com/>
- Galaxy Kate <http://www.galaxykate.com/>
- Inigo Quilez <https://www.iquilezles.org/www/index.htm>
- Peter Wonka <http://peterwonka.net/Publications/publications.html>
- Pascal Muller <https://scholar.google.com/citations?user=Mkf-G5wAAAAJ&hl=en>
- Craig Reynolds <https://www.red3d.com/cwr/>
- Ken Perlin <https://mrl.nyu.edu/~perlin/>
- Daniel Shiffman <https://shiffman.net/>
- Mike Bostock <https://bost.ocks.org/mike/algorithms/>
- Dwarf Fortress World Gen https://dwarffortresswiki.org/index.php/v0.34:Advanced_world_generation
- Mazes for Programmers, Jamis Buck 2015 <https://pragprog.com/book/jbmaze/mazes-for-programmers>
- AI for Games, Ian Millington 2019 <https://amzn.to/32t2HPR>
- Melodrive Blog <http://melodrive.com/blog/>
- Spider Man city <https://www.youtube.com/watch?v=4aw9uyj9MAE>
- Horizon Zero Dawn clouds <https://bit.ly/1K1VrK3>



RAY TRACING

We first need to know how to draw procedural model before we can generate interesting models; otherwise we can't see what we're creating and debugging. So, let's talk about ray tracing for a moment.



Almost all rendering works by putting a virtual camera in a 3D scene and then tracing a light ray back through each pixel to find out what colored the pixel. It then traces some additional light rays back to find out where the light originally came from, which is ultimately going to be something like the sun or a light bulb.

There are a lot of different algorithms for rendering. The core operation for all of them is tracing a single ray of light until it hits something in the scene. This is called "ray casting" or "RAY TRACING" in common graphics jargon, (although we also use that phrase to refer to entire rendering algorithm sometimes!)

Here are two ways of implementing the RAY TRACING operation for surfaces:

EXPLICIT SURFACE RAY TRACING



Image from "The Amazing Wireframe Shader" by Arkhivrag, <http://u3d.as/86j>

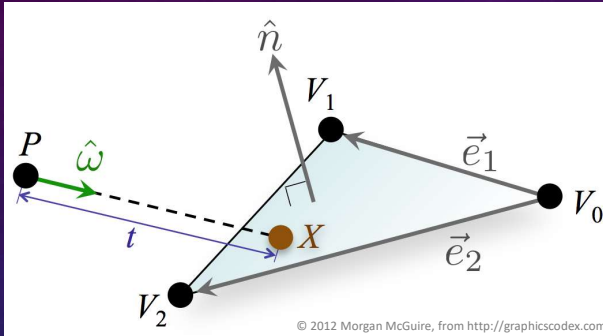
```
class Point { x, y, z }  
class Vector { x, y, z }  
  
class Ray { Point origin,  
           Vector direction }  
  
class Triangle { Point vertex[3] }  
class Mesh { Triangle[] }
```

If the 3D scene is made out of triangles or subdivision patches, then the algorithm for solving this is explicit surface ray tracing. A special case of this is rasterization, but I'm not going into that today.

This is how most rendering is done, both real-time and offline.

This image reveals its underlying triangle mesh representation as it fades in on the right as a visualization. I put some simple code for how the world is represented for this kind of explicit surface triangle mesh.

EXPLICIT SURFACE RAY TRACING



```
class Point { x, y, z }  
class Vector { x, y, z }
```

```
class Ray { Point origin,  
            Vector direction }
```

```
class Triangle { Point vertex[3] }  
class Mesh { Triangle[] }
```

Foundational research:

- **Some techniques for shading machine renderings of solids**, Appel 1968
- **An improved illumination model for shaded display**, Whitted 1980
- **A 3-dimensional representation for fast rendering of complex scenes**, Rubin and Whitted, 1980
- **The rendering equation**, Kajiya 1986

The actual rendering algorithm involves doing some geometry work to compute where the light ray hits one triangle, and then applying that to all of the triangles in the mesh.

On the bottom I've highlighted some key research papers that developed us towards the state of the art for this kind of rendering.

The great news is that while this is something you can implement yourself (and will in the intro graphics course!), today GPUs and high-level APIs like Vulkan...



have it all built in, so you get a *really* fast and already debugged implementation. By really fast, I mean processing billions of rays per second against millions of triangles.

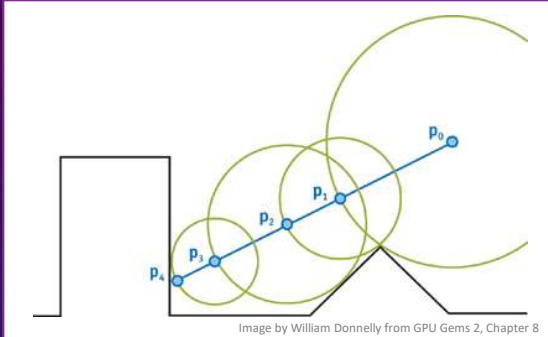
EXPLICIT SURFACE RAY TRACING RESOURCES

- Ray Tracing in One Weekend, Shirley 2018 <http://www.realtimerendering.com/raytracing/Ray%20Tracing%20in%20a%20Weekend.pdf>
- The Graphics Codex, McGuire 2019 <http://graphicscodex.com/>
- Introduction to DXR by Adam Marrs <http://www.visualextract.com/posts/introduction-to-dxr/>

- Computer Graphics: Principles & Practice, Hughes et al. 2013 <https://amzn.to/2pLKKgB>
Free chapter on ray tracing http://cgpp.net/file/cgpp3e_ch15.pdf
- Physically-Based Rendering, Pharr et al. 2016 <http://www.pbr-book.org/>
- Fundamentals of Computer Graphics, Marschner and Shirley 2015 <https://amzn.to/2ocrqZK>
- Real-Time Rendering, Akenine-Möller et al. 2018 <https://amzn.to/2P7nynI> ch. 19, 22, 24
Free chapter on ray tracing APIs: http://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf

You can look at some of these resources after the talk from my online slides to learn more about explicit surface ray tracing on CPUs and GPUs.

IMPLICIT SURFACE AND VOLUMETRIC RAY TRACING



function `distanceToSurface(Point p)`

or

function `densityOfVolume(Point p)`

Foundational research:

- A generalization of algebraic surface drawing, Blinn 1982
- Marching cubes: a high resolution 3D surface construction algorithm, Lorensen and Cline 1987
- Ray tracing deterministic 3-D fractals, Hart et al. 1989
- Sphere tracing: a geometric method for antialiased ray tracing of implicit surfaces, Hart 1996
- Enhanced sphere tracing, Keinert et al. 2014

Sometimes it is better to represent the scene using an implicit volumetric representation. This is how fluids, smoke, ...and certain kinds of procedural content are handled.

You can render these by *converting* the implicit surface to an explicit surface (or the volume to a whole lot of particle surfaces) and then running an explicit surface ray tracer. That is how most mainstream graphics works today because it can take advantage of highly optimized explicit surface ray tracing implementations.

OR, you can directly render the distance or density function by “marching” along a ray through 3D space. That is what we’re going to do. The reason we’ll use this is that it is extremely elegant to not make an intermediate representation, and when the model is incredibly detailed or constantly changing, it is also faster to work with the implicit form instead of expanding it into triangles.

Let me show you a simple example of defining geometry for this kind of renderer...

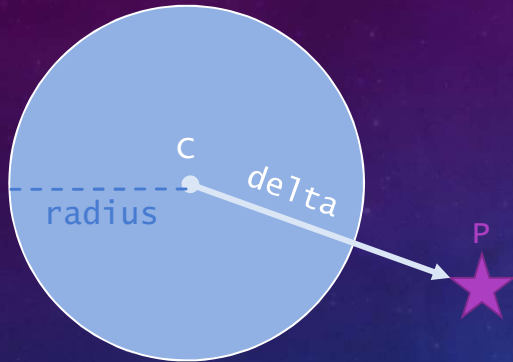
(This figure is by Donnelly, who was a Waterloo alumnus)

IMPLICIT SURFACE RAY TRACING RESOURCES

- The Graphics Codex, McGuire 2019 <http://graphicscodex.com/> “Ray Marching” chapter
- Modeling with Distance Functions by Inigo Quilez <http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm>
- Ray marching and signed distance functions by Jamie Wong <http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/>
- Implicit Curves and Surfaces, Wyvill 2009
- Level Set Methods and Dynamic Implicit Surfaces, Osher and Fedkiw 2003
- Implicit surfaces by Paul Bourke <http://paulbourke.net/geometry/implicitsurf/>
- Inigo Quilez <https://www.iquilezles.org/>
- Brian Wyvill <http://webhome.cs.uvic.ca/~blob/>

MODELING IMPLICIT SURFACES

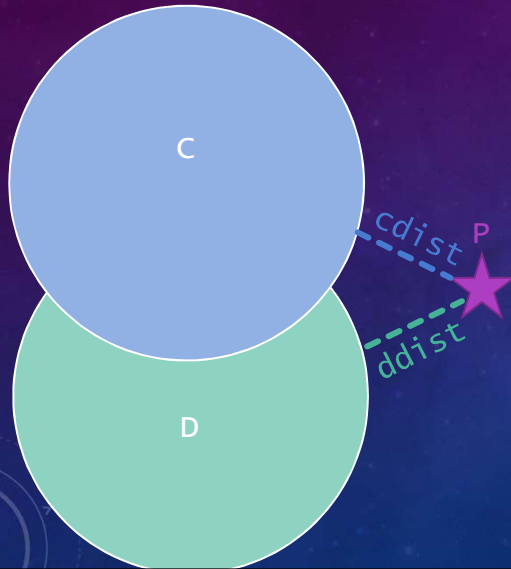
SPHERE SIGNED DISTANCE FUNCTION



```
function distanceToSurface(Point P)  
  Vector delta = P - C  
  return length(delta) - radius
```

```
function length(Vector v)  
  return sqrt(v.x2 + v.y2 + v.z2)
```

UNION OF TWO SPHERES



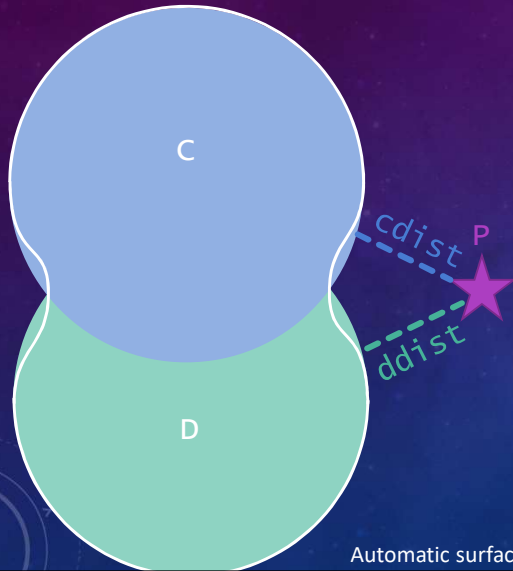
```
function distanceToSurface(Point P)
  Vector cdist = length(P - C) - radius
  Vector ddist = length(P - D) - radius
  return min(cdist, ddist)
```

```
function length(Vector v)
  return sqrt(v.x2 + v.y2 + v.z2)
```

That's a really easy way to compose shapes. This is the basic idea we'll use to create complicated geometry from simple pieces.

Even better, to get a more natural form, you can simply tweak the operation used to combine the two underlying distance functions

BLENDING TWO SPHERES



```
function distanceToSurface(Point P)
  Vector cdist = length(P - C) - radius
  Vector ddist = length(P - D) - radius
  return smoothmin(cdist, ddist)
```

```
function smoothmin(a, b)
  c = max(0, min(1, ½ + b - a))
  return c * b + (a - ½) * c * (1 - c)
```

Automatic surface generation in computer aided design, Hoffman and Hopcroft, 1985

Here I'm using a SMOOTHMIN instead of a strict minimum. It blends the two shapes together instead of taking the union to create a sharp boundary.

With distance function representations, there are a lot of elegant ways to combine shapes: intersection, union, subtraction, reflections, repetition, twists, rigid transformations, skew, etc., and smooth versions of all of those. Here are some references that you can use as a kind of implicit surface cookbook.



THE SHADERTOY DEMOSCENE

The “shadertoy” website is a simple code editor for writing GPU programs in a web browser. Its best feature is that you can see the source code for everyone else’s programs, so you can learn by modifying and copying their code. There is also a really supportive community on the site that will cheer you on and help debug and optimize your programs.

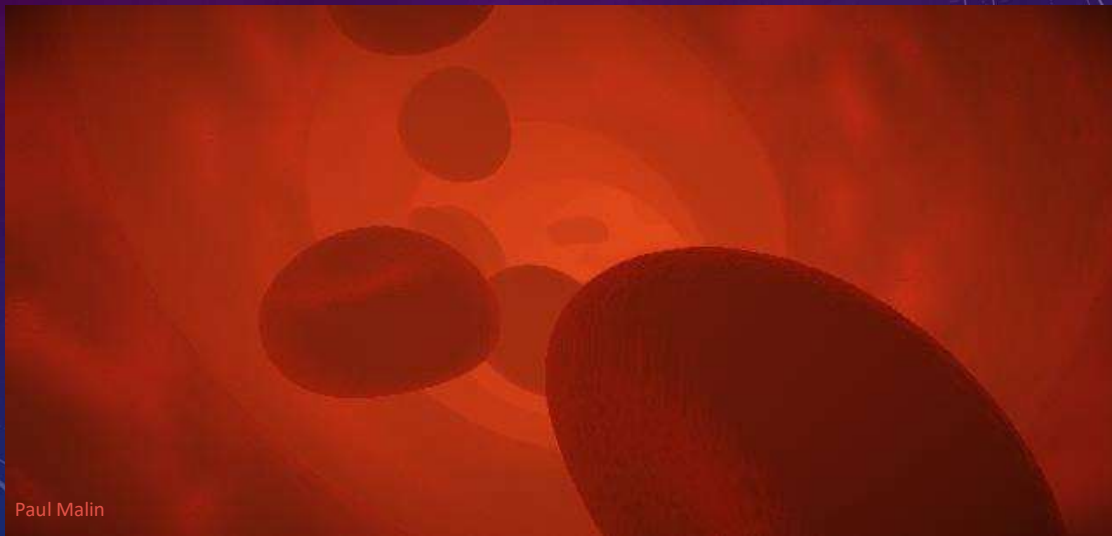
SNAIL

<https://www.shadertoy.com/view/ld3Gz2>



RED CELLS

<https://www.shadertoy.com/view/MsXXWH>



Paul Malin

HOLY GRAIL QUEST II

<https://www.shadertoy.com/view/MtfGWM>



eiffie

VENICE

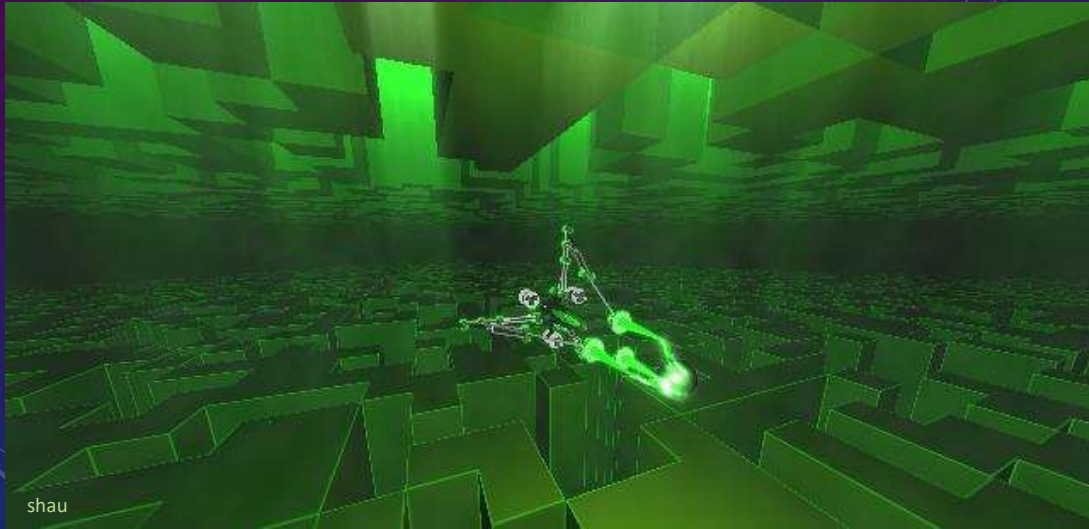
<https://www.shadertoy.com/view/MdXGW2>



Reinder Nijhoff

ETHICS GRADIENT

<https://www.shadertoy.com/view/tdGGRh>



shau

GENERATORS

<https://www.shadertoy.com/view/Xtf3Rn>



GETTING STARTED WITH SHADERTOY



Video tutorial by Quilez

<https://www.youtube.com/watch?v=0ifChJ0nJfM>



Blog tutorial by Neyret

<https://shadertoyunofficial.wordpress.com/>



Example Implicit Surface Ray Tracer

<https://www.shadertoy.com/view/Ms2SWw>

Example Explicit Surface Ray Tracer

<https://www.shadertoy.com/view/XdsGWS>

Mandelbulb Fractal Explained

<https://www.shadertoy.com/view/XsXXWS>



BUILDING A TINY PLANET

I ♥ ABSTRACTION

```
#define Vector2      vec2
#define Point3      vec3
#define Vector3      vec3
#define Color3      vec3
#define Radiance3   vec3
#define Irradiance3 vec3
#define Power3      vec3
#define Biradiance3 vec3

const float pi      = 3.1415926535;
const float degrees = pi / 180.0;
const float inf     = 1.0 / 1e-10;

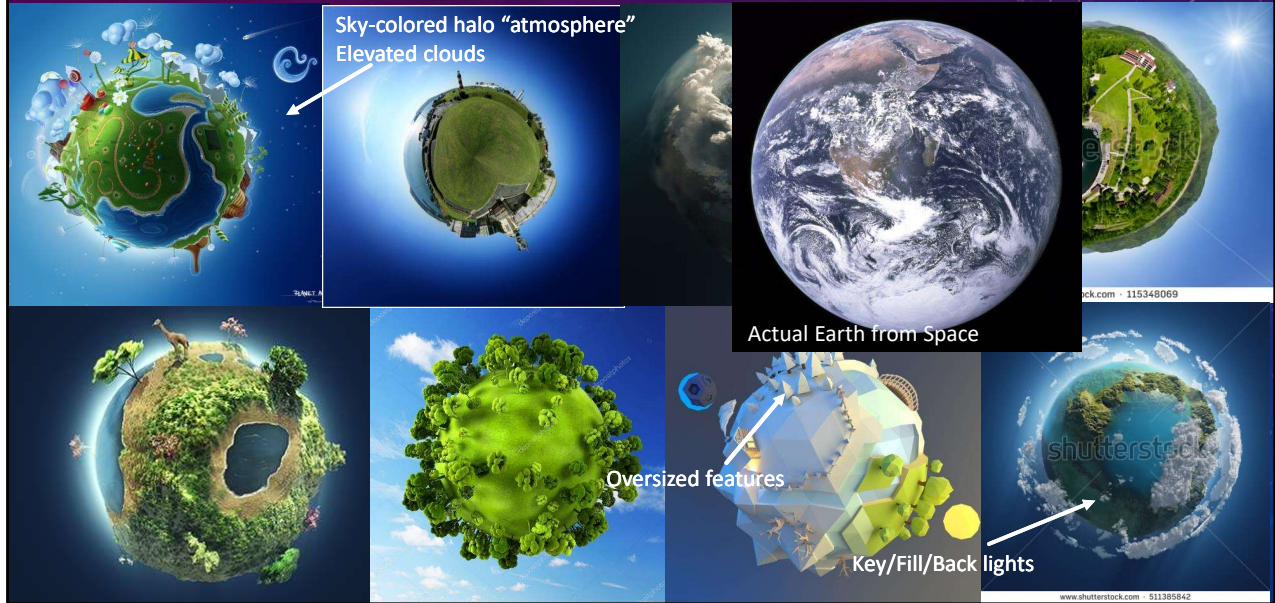
struct Ray    { Point3 origin;   Vector3 direction; };
struct Material { Color3 color;   float metal;   float smoothness; };
struct Surfel { Point3 position; Vector3 normal;   Material material; };
struct Sphere { Point3 center;   float radius;   Material material; };
```

One of the great things about shadertoy is that anybody can look at your code. So, we all get to learn from each other and new coders can quickly get up to speed.

The clearer that you make your shaders, the more that you'll help advance the field and bring new people into graphics. I try to always make a cleanup pass with extra comments and abstractions...in fact, most of my shadertoy use has just been showing how pieces of other people's demos work.

OK, so here's what I wanted to do:

CONCEPT ART (FROM OTHERS)



Here's the kind of little prince tiny planet visual I wanted to hit.

First, you'll notice that this looks nothing like actual photos of the Earth from outer space.

So, some of the things we want are:

- The atmosphere is radically expanded so that the planet is silhouetted against a halo of "terrestrial sky" color instead of outer space
- Clouds are expanded outward and cast large shadows
- Although the real earth experiences strong directional lighting with no bounce, these planets have at least 3-point lighting
- If you don't add the lighting and the sky background, the planet looks cold and sad.

ATMOSPHERE

Here we go again. We start with the black screen.

ATMOSPHERE

```
vec2 delta = (fragCoord.xy - iResolution.xy * 0.5) *  
  invResolution.y * 1.1;  
float atmosphereRadialAttenuation =  
  min(1.0, 0.06 * pow8(max(0.0, 1.0 - (length(delta) - 0.9) / 0.9)));
```

I add a turquoise radial gradient

ATMOSPHERE

```
float radialNoise = mix(1.0, noise(normalize(delta) * 40.0 + iTime * 0.5), 0.14);
```

Make some radial streaks (I don't care about the center, which will be covered by the planet)

ATMOSPHERE

```
float galaxyClump = (pow(noise(fragCoord.xy * (30.0 * invResolution.x)), 3.0) * 0.5 +  
                    pow(noise(100.0 + fragCoord.xy * (15.0 * invResolution.x)), 5.0)) / 1.5;  
L_o = color3(galaxyClump * pow(hash(fragCoord.xy), 1500.0) * 80.0);  
  
// Color stars  
L_o.r *= sqrt(noise(fragCoord.xy) * 1.2);  
L_o.g *= sqrt(noise(fragCoord.xy * 4.0));  
  
// Twinkle  
L_o *= noise(time * 0.5 + fragCoord.yx * 10.0);
```

Add some stars

ATMOSPHERE

```
vec2 re1 = 0.65 * (fragCoord.xy - iResolution.xy * 0.5) / iResolution.y;  
float a = min(1.0,  
             pow(max(0.0, 1.0 - dot(re1, re1) * 6.5), 2.4) +  
             max(abs(re1.x - re1.y) - 0.35, 0.0) * 12.0 +  
             max(0.0, 0.2 + dot(re1, vec2(2.75))));  
float planetShadow = mix(minVal, maxVal, a);
```

Ad the cut out what will be the shadow of the planet on its own “atmosphere”

MOUNTAINS



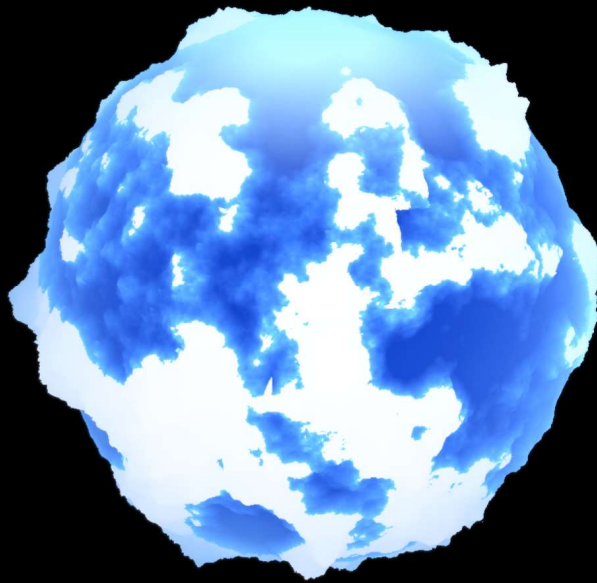
For the terrain, we start with a sphere distance function.

MOUNTAINS



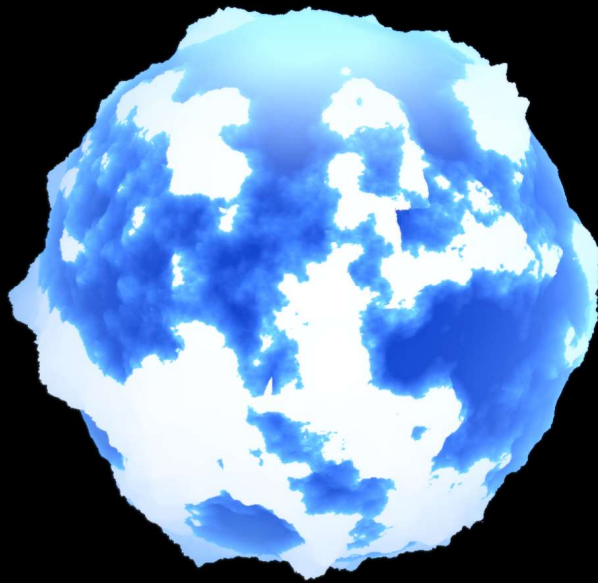
Then we distort the elevation by a noise function over the surface of the sphere. This is 6th order fractal Brownian motion, raised to the third power so that the mountains are more rare and steep.

OCEANS



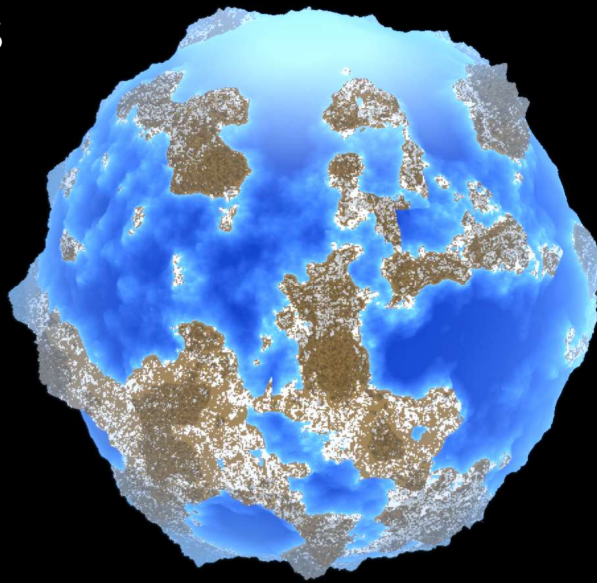
If the elevation is below sea level, we code that as water and pull the surface up to be at sea level (but keep track of how deep the water is).

OCEANS



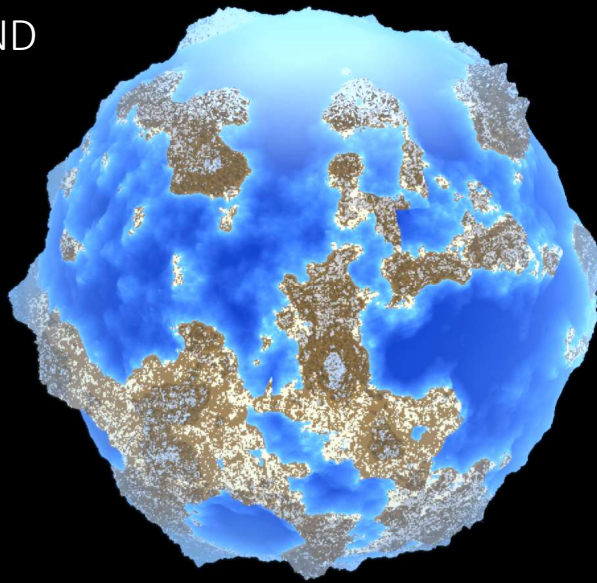
If the elevation is below sea level, we code that as water and pull the surface up to be at sea level (but keep track of how deep the water is).

MOUNTAINS



Above a certain elevation, we'll classify the surface as rock

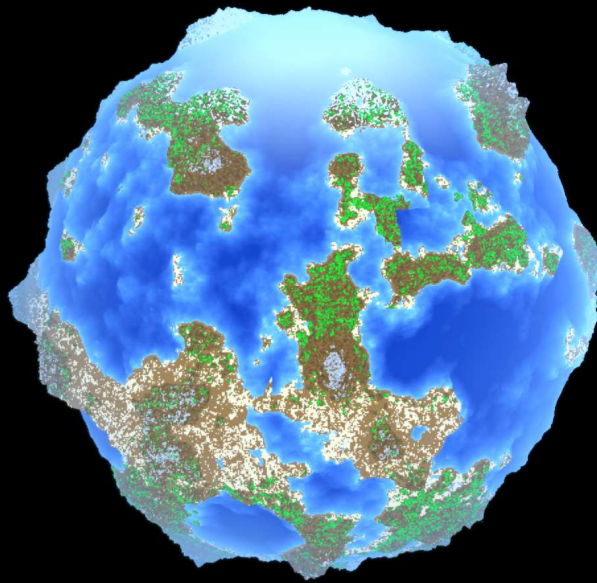
ICE AND SAND



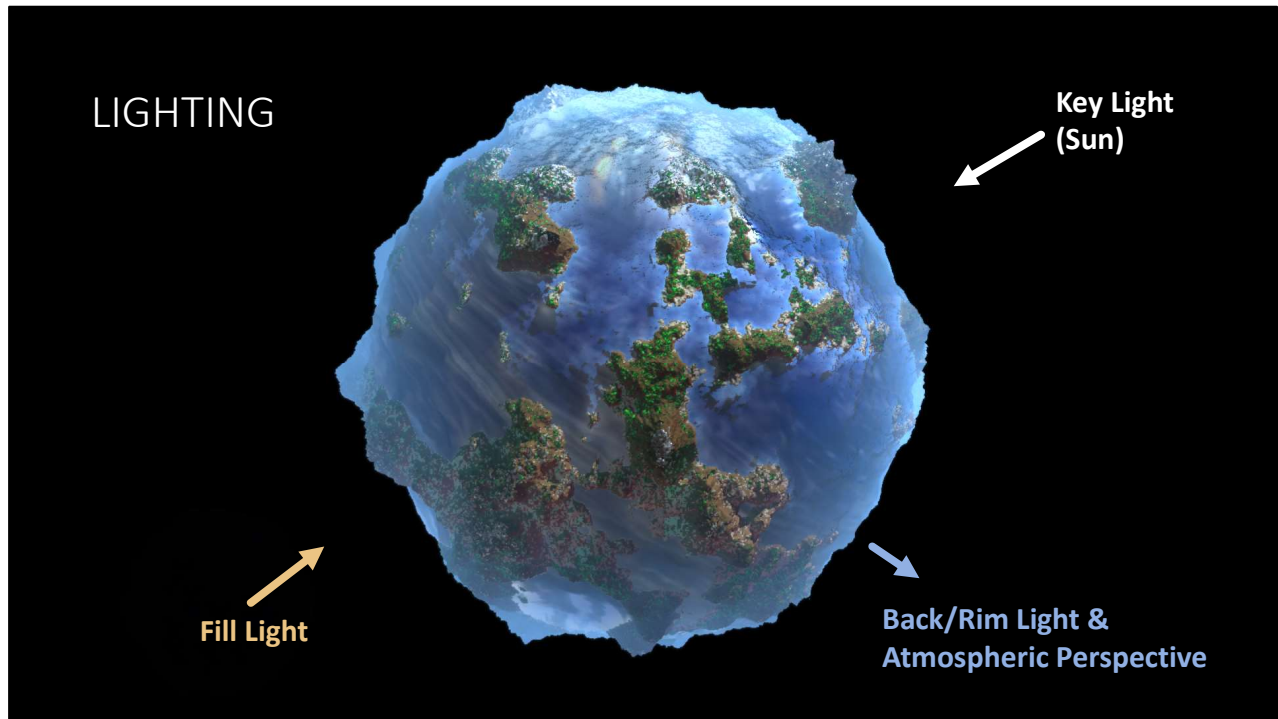
With ice at the poles and really high elevations

And sand along beaches and for some equatorial deserts

VEGETATION



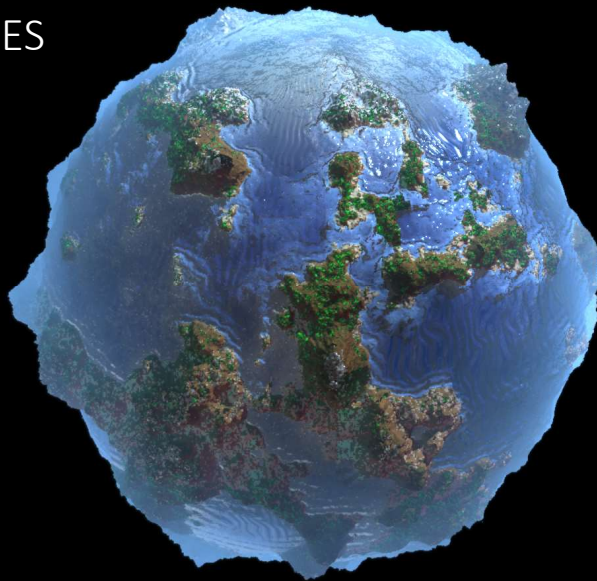
Let's grow some trees wherever the climate is good and the terrain is not too steep.



I effectively lit the scene with standard three-point lighting, although that “fill” light is really a bit more complicated and includes a reflection environment from all directions.

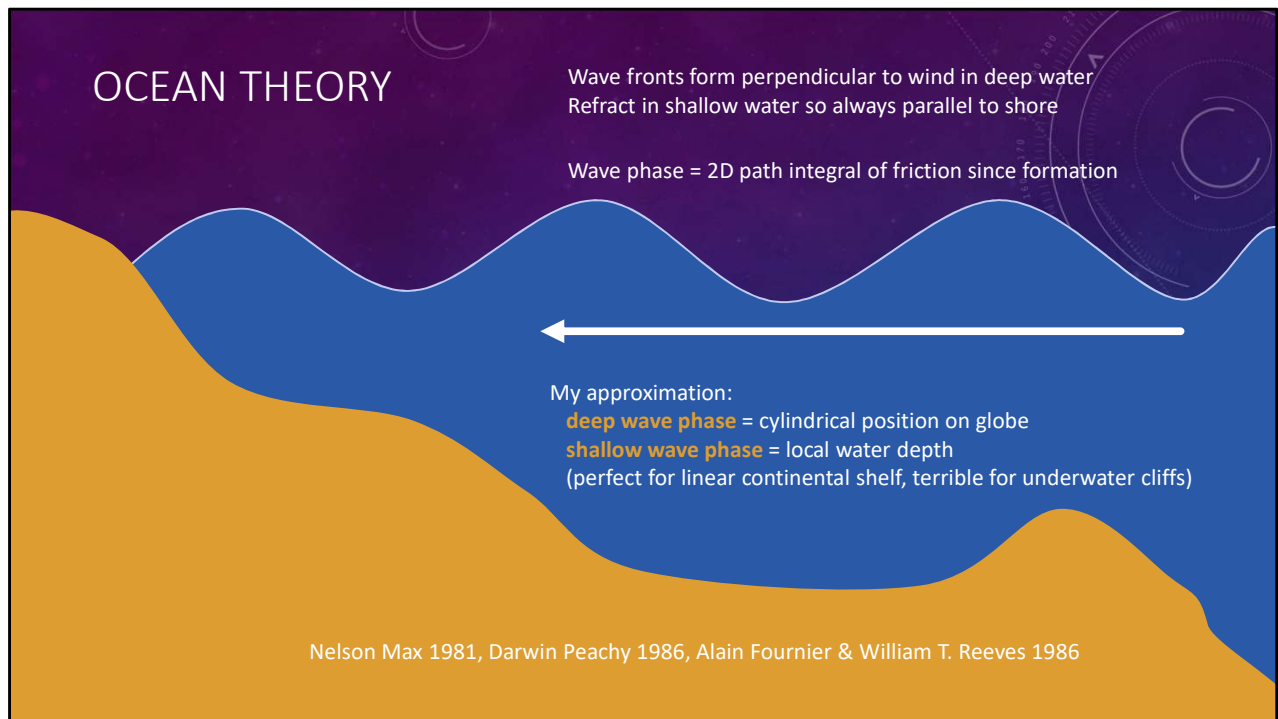
The key light is the only one that is shadowed.

OCEAN WAVES



I effectively lit the scene with standard three-point lighting, although that “fill” light is really a bit more complicated and includes a reflection environment from all directions.

The key light is the only one that is shadowed.



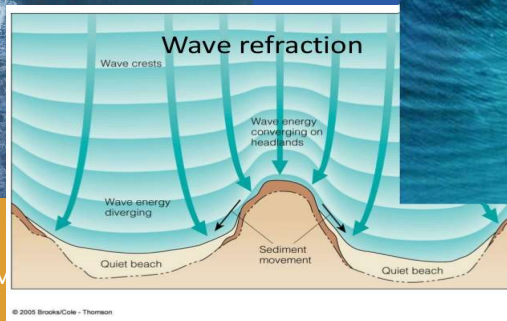
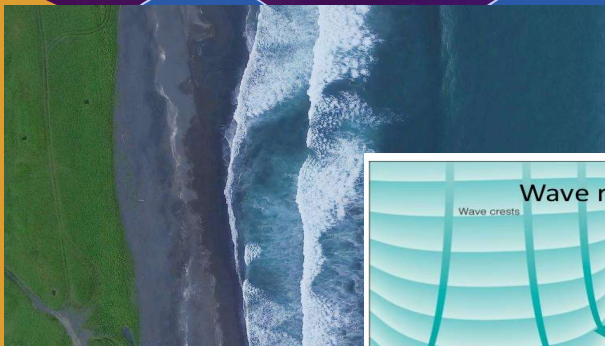
Waves refract. I'm not talking about light, I'm talking about the shape of the geometry...it bends so that waves always come in to shore almost parallel.

You can simulate this with an expensive path integral over the entire distance travelled since the wave formed, as described in these three papers.

OCEAN THEORY

Wave fronts form perpendicular to wind in deep water
Refract in shallow water so always parallel to shore

Wave phase = 2D path integral of friction since formation



Nelson M

am T. Reeves 1986

What this means is that in the real world, waves far from shore have a direction based on the wind and waves coming in to shore are always parallel to the shoreline.

In this Google Maps photograph, you can see the waves parallel to the shore. Theory tells us how they'll refract if the shore has a complex shape...

And indeed, that is what you'll see in this other photograph. The path integral is too expensive for use in the shader.

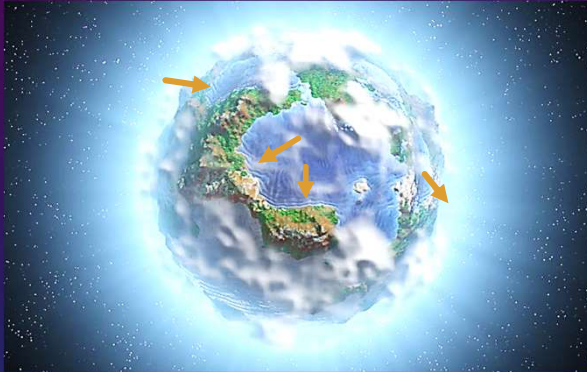
I use a simple approximation, instead: If we assume that the underwater shore falls off with a constant slope (which is somewhat reasonable), then the path integral is proportional to the distance between sea level and the ocean floor.

We can just substitute the elevation as the phase for a sine wave and get proper wave refraction phenomena.

OCEAN THEORY

Wave fronts form perpendicular to wind in deep water
Refract in shallow water so always parallel to shore

Wave phase = 2D path integral of friction since formation



You can see that in this debug view...the waves are always parallel to the shoreline in shallow water in the tiny planet.

OCEAN IMPLEMENTATION

```
float relativeWaterDepth = min(1.0, (water - mountain) * 30.0);
const float waveMagnitude = 0.0008;
const float waveLength = 0.008;

// Create waves. Shallow-water waves conform to coasts. Deep-water waves follow global wind patterns.
const Color3 shallowWaterColor = Color3(0.4, 1.0, 1.9);
// How much the waves conform to beaches
const float shallowWaveRefraction = 1.0;
float shallowWavePhase = (surfaceLocation.y - mountain * shallowWaveRefraction) * (1.0 / waveLength);

const Color3 deepWaterColor = Color3(0, 0.1, 0.7);
float deepWavePhase = (atan(surfaceLocation.z, surfaceLocation.x) + noise(surfaceLocation * 15.0) * 0.075) * (1.5 / waveLength);

// This is like a lerp, but it gives a large middle region in which both wave types are active at nearly full magnitude
float wave = (cos(shallowWavePhase + iGlobalTime * 1.5) * sqrt(1.0 - relativeWaterDepth) +
             cos(deepWavePhase + iGlobalTime * 2.0) * 2.5 * (1.0 - abs(surfaceLocation.y)) * square(relativeWaterDepth)) *
             waveMagnitude;

elevation = water + wave;

// Set material, making deep water darker
planetMaterial = Material(mix(shallowWaterColor, deepWaterColor, pow(relativeWaterDepth, 0.4)), 0.5 * relativeWaterDepth, 0.7);
// Lighten polar water color
planetMaterial.color = mix(planetMaterial.color, Color3(0.6, 1.0, 0.9) * 0.75,
                          square(clamp((abs(surfaceLocation.y) - 0.667) * 3.0, 0.0, 1.0)));
```


CLOUDS

```
float cloudDensity(Point3 x, float t) {  
    return 0.1;  
}
```

Now I'll make some clouds. They aren't a surface, so instead of a distance-to-surface function, I define a density function for every point in space.

If I make the density uniform, then it just looks like the planet is covered in fog.

You can see a few mountain peaks sticking up. I've turned off the planet shading, so the mountains are just black right now.

CLOUDS

```
float cloudDensity(Point3 X, float t) {  
    Point3 p = X * vec3(1.5, 2.5, 2.0);  
    return fbm5(p) - 0.42;  
}
```

If I use a 3D noise function for the cloud density, then we get nice variation creating weather patterns. They are too regular, however.

CLOUDS

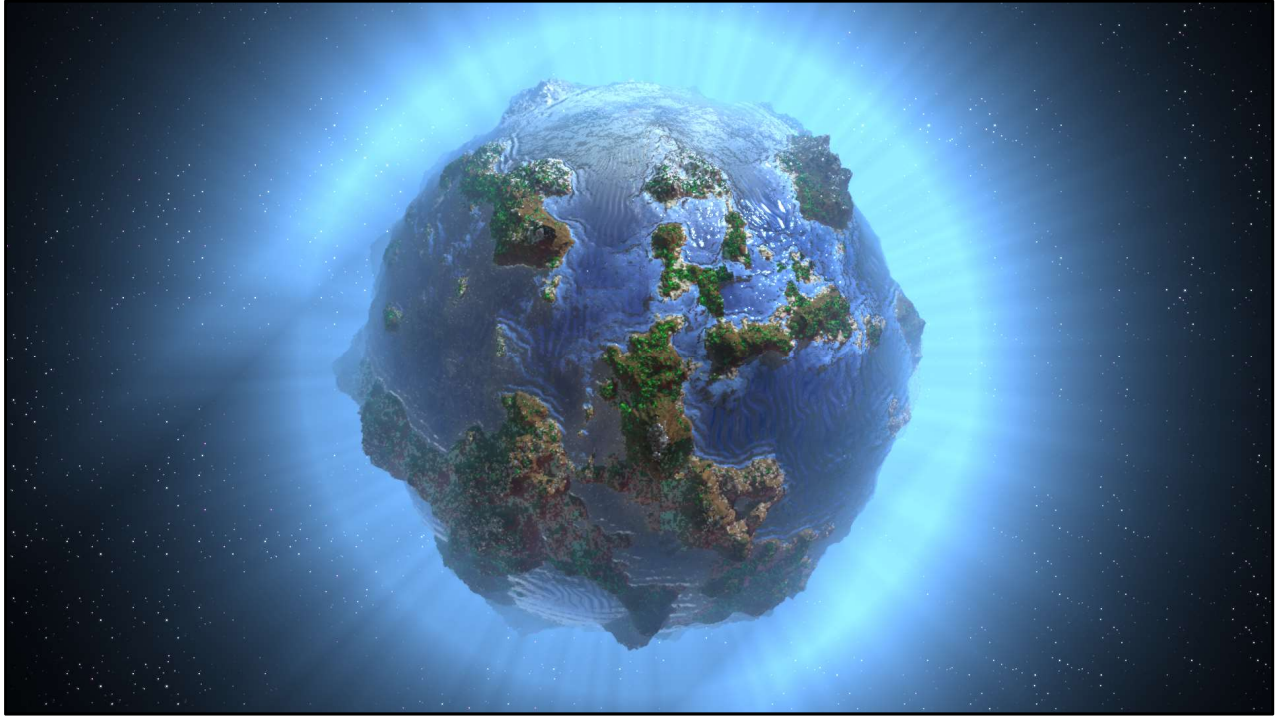
```
float cloudDensity(Point3 X, float t) {  
    Point3 p = X * vec3(1.5, 2.5, 2.0);  
    return fbm5(p + 1.5 * fbm3(p)) - 0.42;  
}
```

Iterating on the noise function by feeding it back into itself makes the clouds start to swirl.

I can also make them develop over time by making the argument depend on the current time, which is “t” here.



Now, let's put it all together. Here's the atmosphere (without the planet shadow)



With the planet on top



And the clouds on top.

There are some other things I'm doing here, like post-process color grading, antialiasing, and variable resolution rendering which you can see in the full shader.

SUMMARY 1/2: PROCGEN & RAY TRACING

Procedural content

- You create the rules, the rules create the art
- Signal → Expansion → Interpretation
- Geometry, texture, narrative, music, stats, animation...
- Iterative process

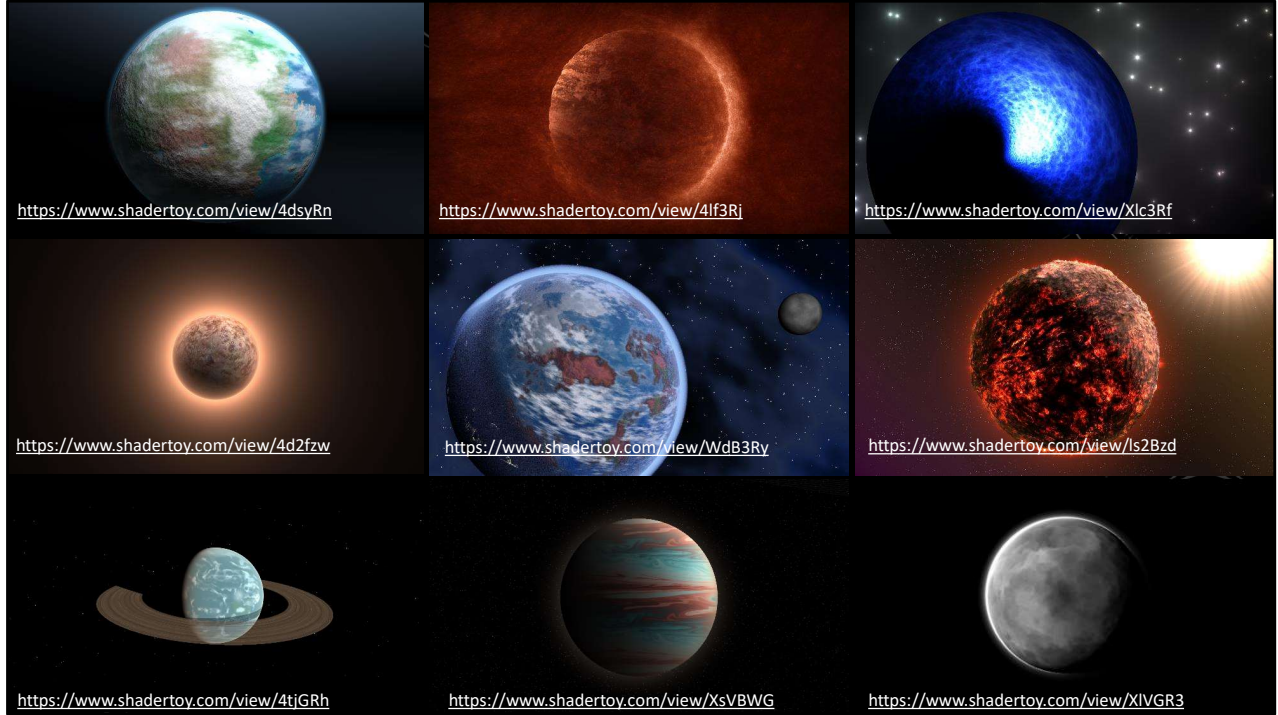
Ray tracing for implicit and explicit geometry

- Implicit geometry is often easier to synthesize
- Explicit geometry has hardware rendering support
- Can convert between them

Shadertoy for experimentation with ray tracing and procedural content

SUMMARY 2/2: THE TINY PLANET

- **Art direction!**
 - Reference & concept art
 - Material, geometry, animation, and lighting
- **Value noise with fractal brownian motion** for terrain
- Elevation, slope, and latitude determine **biome**
- Approximated path integral creates refracting ocean waves
- Volumetric **ray marching** for clouds
- Performance and post-processing polish



Here are some variations on this theme by others.

I hope that you'll download these slides to get the code links, and then take some time tonight or this weekend to start your own experiments. Here's my email and Twitter. Please share what you create!

Thank you.

THANK YOU

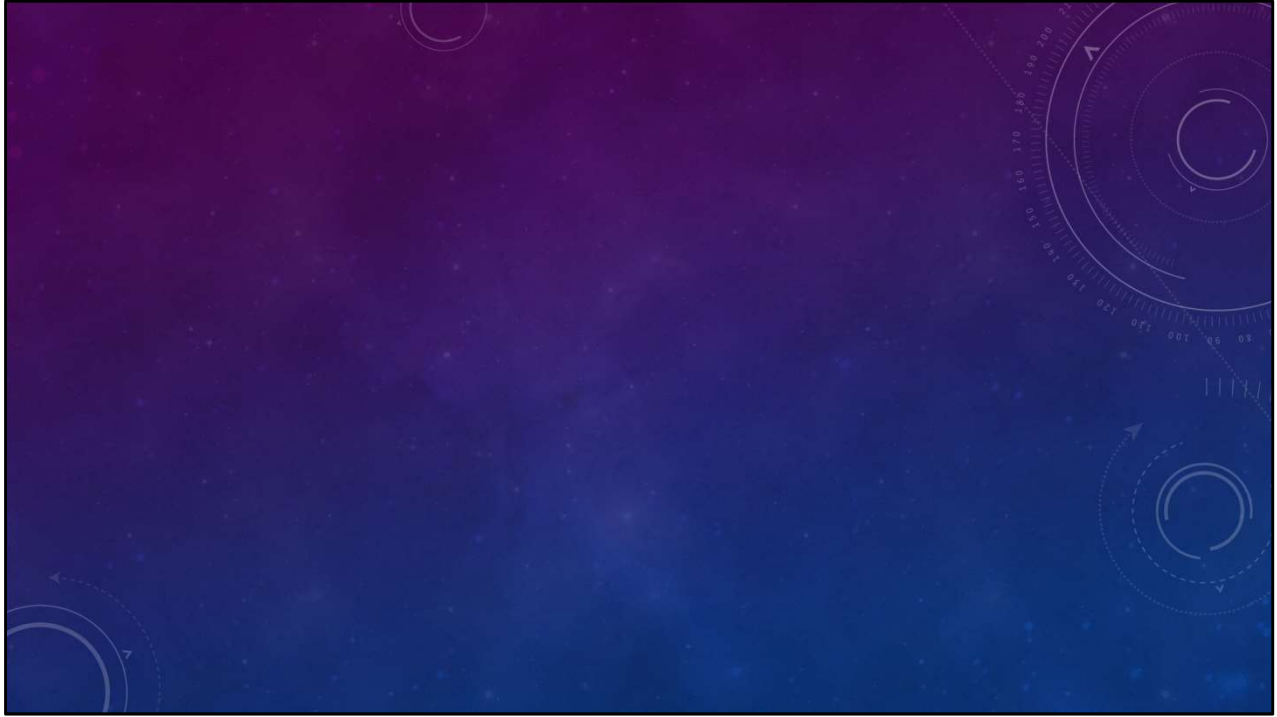
Contact: @CasualEffects on Twitter | morgan@casual-effects.com

Slides hosted at <https://casual-effects.com>

More details in the Graphics Codex <http://graphicscodex.com>

PROCIAM November 2019 <http://www.prociam.com/>

Thanks to Peter Shirley, Josef Spjut, Sina Nabizadeh, and Inigo Quilez for help preparing this talk.



RECIPE FOR A PLANET

- Happy 2D halo + star field
- Sphere trace a noisy sphere
 - Only trace 2D planetary disk
 - Sea level threshold
 - Ocean wave refraction physics
 - Material from latitude, elevation, and slope
 - Shadowed key light + unshadowed fill + environment map reflection
 - Distance fog
- Ray march $1/3^2$ resolution clouds, stopping at depth buffer
- Bloom and tone map post FX
- Simple temporal antialiasing when not moving too fast

If I just render what I described, it flickers and aliases terribly. So, I apply very simple temporal antialiasing, since I know the motion. When you use the mouse to rotate quickly, I lower the AA amount to avoid ghosting.

So, this is basically a classic heightfield scene like I learned to draw from Inigo, but wrapped onto a sphere. I'm a sailor, though, and the trick I'm most proud of is the ocean waves....

RAY TRACING A TINY PROCEDURAL PLANET

Dynamic procedural worlds are magical. They have limitless detail to explore and can surprise even their creators, who work with small amounts of elegant code. Minecraft and No Man's Sky are two examples of great games built entirely on procedural generation and detailed simulation. In this talk I first describe real-time ray tracing of implicit surfaces and give a tour of the Shadertoy online playground for real-time graphics exploration. I then show how I art directed and implemented the popular "Tiny Planet" real-time demo using procedural generation, dynamic simulation, and ray tracing, all implemented as 600 lines of GPU shader code. Attendees with a year of programming experience will be able to immediately begin their own experiments in procedural content and ray tracing online after the talk, supported by links from the slides and follow-up details in the Graphics Codex.

Dr. Morgan McGuire is a Distinguished Research Scientist at NVIDIA and adjunct professor at the University of Waterloo and McGill University. Morgan is the coauthor of The Graphics Codex, Computer Graphics: Principles and Practice, and Creating Games books and many research papers; contributed to the Skylanders, ROBLOX, Marvel Ultimate Alliance, Call of Duty, and Titan Quest games; and taught for 12 years at Williams College.

