I DON'T LIKE PERLIN NOISE There, I feel better already!

This improved feeling comes at the risk of committing procedural blasphemy[1]. You'd be right in saying Perlin noise is very useful. Its also complicated and for some of us opaque.

A multilayered, random-ish waveform, Perlin and other[2] noise systems can be put to use making Waves, Islands, Caves etc. Noise systems generate these structures easily because they produce a repeating output that is never quite the same. Variations in the output are a similar scale and can have a distinctive shape, allowing us to use it as a source for hills or islands etc which are created by the same geological processes.

One way to improve the shape of the landscapes is to gradually increase scale on the y-axis, making the higher points of the landscape more pointy. A profile more closely matching mountains and hills.

Public Domain, via Wikimedia Commons

[1] An 'electric sinner' could use AI, and NLP to codify the rules of religious texts. Build a machine to break those rules perhaps automatically tweeting results. Maybe inciting others to commit sin and finally in action. Somewhat like a morality play (some might say that video games are already doing this better anyway).

[2] Lets call all these value noise systems Perlin noise to avoid sentence mangling.

Webmaster.vinarice CC BY-SA 4.0 via Wikimedia Commons

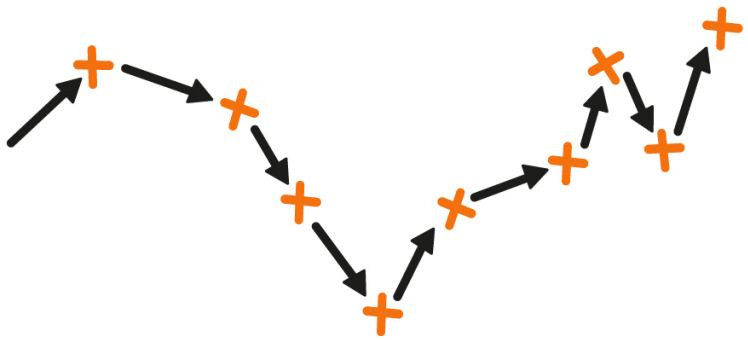
High end procedural generation of sand dunes is complex with authentic outputs that may not appropriate for gameplay. Journey's gameplay relied on hand designed dunes. Extra rendering passes added interest. Conversely in Meteor Storm Escape we included a dune racing level, we compromised all authenticity for a challenging and exhilarating player experience.

We have a choice, from arbitrary algorithms and heuristics which generate forms useful for gameplay, to simulations whose internal factors accurately replicate

the internal states and dynamics of the system found in reality whose aspects we find valuable for our game.

Plate tectonics allows volcanos to form along long wiggly lines. Using a move-turn, loop to manipulate a sequence of vectors we can steer a random walk in a direction. Parameterising the length of the vectors and size of change in orientation, to create linear-ish path as the place where the world's crust is thin.

```
var rotation = constAngle * ((Math.random() + Math.random())-1)
Turn (0.0, 1.0, 0.0, rotation)
var distance = constDist * ( 0.5 * (Math.random() + Math.random()))
Move (0.0, 0.0, 1.0, distance)
```



This approach bears no relationship with the forces involved in the joining or separating of two tectonic plates, but can easily be tweaked to produce sequences which bear a resemblance to the jagged shape of the Mid-Atlantic Ridge or match specific gameplay requirements such as maximum distance between islands.

Assuming the direction and velocity of ejecta is random, but the angle of ejection follows a bell curve.

```
var angle = (( Math.random() *  Math.PI/2) + ( Math.random() *  Math.PI/2)) /3
var direction = Math.random() * ( 2 * Math.PI)
var velocity = minvel + (Math.random() * 2)
```

Adjusting the range and distribution of these values has an impact on the shape of the islands produced. You might prefer more caldera, more sharp jagged peaks etc. Tweak and see what you get.

We could generate rigid bodies and run the engine's physics system. A simulation is not always an option on web or mobile. Instead algebraically, one line of code calculates the approx landing point of that which is thrown out of the volcano.

```
var distance = ( (velocity * velocity) * Math.sin( 2 * angle ) ) / 9.8
```

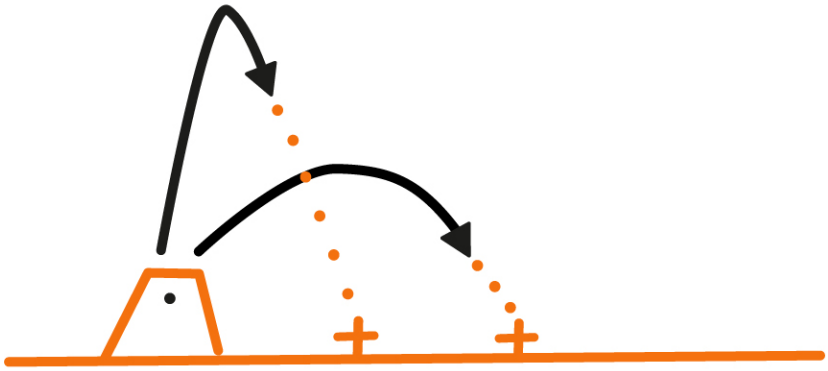
The distance away that something lands is proportional to the square of velocity times the sine of the angle it is ejected at, divided by the gravity. This assumes the ejection point isn't higher or lower than the landing point. Simple trigonometry will translate the distance and direction into a vector offset.

```
var deltax = Math.sin(direction) * distance;
var deltaz = Math.cos(direction) * distance;
```

We can then do a little maths on to get a vertex index where it lands.

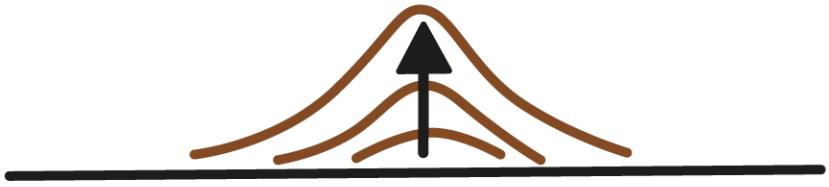
```
var coords = new THREE.Vector3();
coords.x = Math.floor(centre.x + (deltax * (this.size * 0.5)));
coords.z = Math.floor(centre.z + (deltaz * (this.size * 0.5)));

var meshVertexIndex = ( coords.x * (this.size +1) ) + coords.z;
```

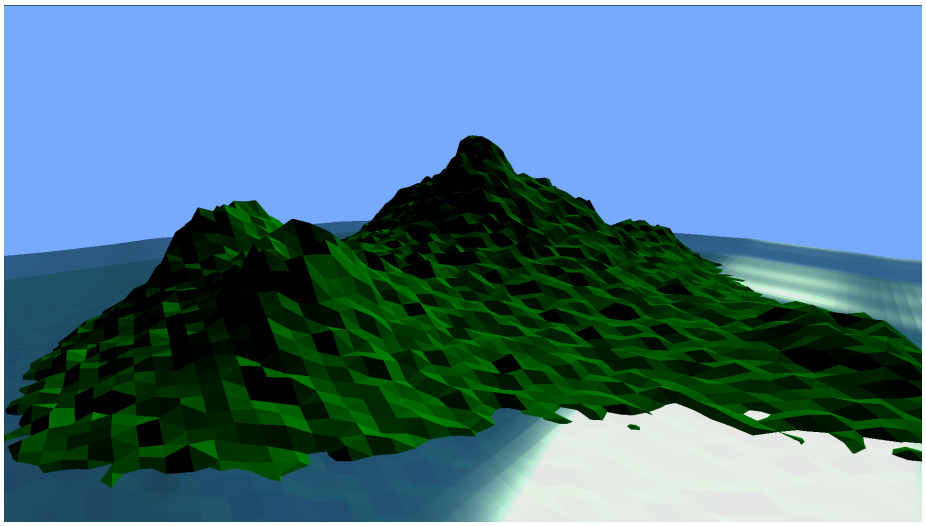


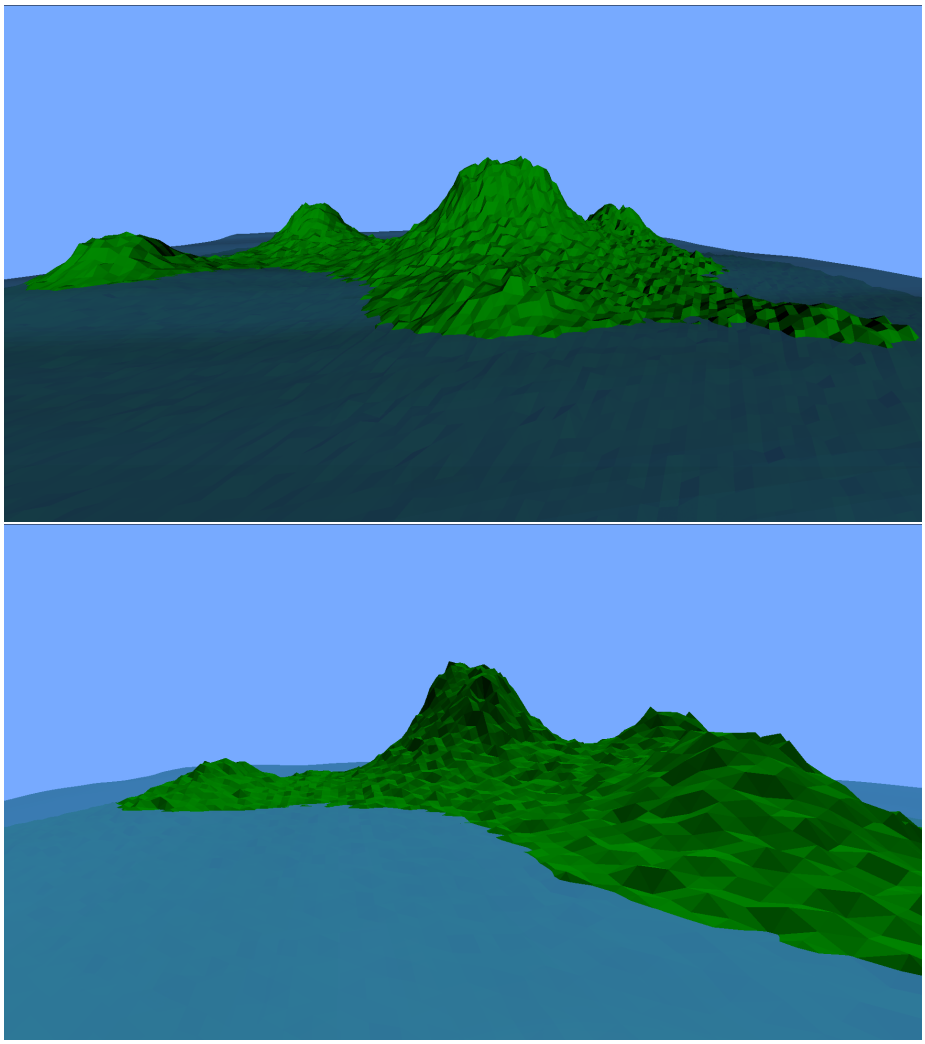
Finally raise the vertex a little!

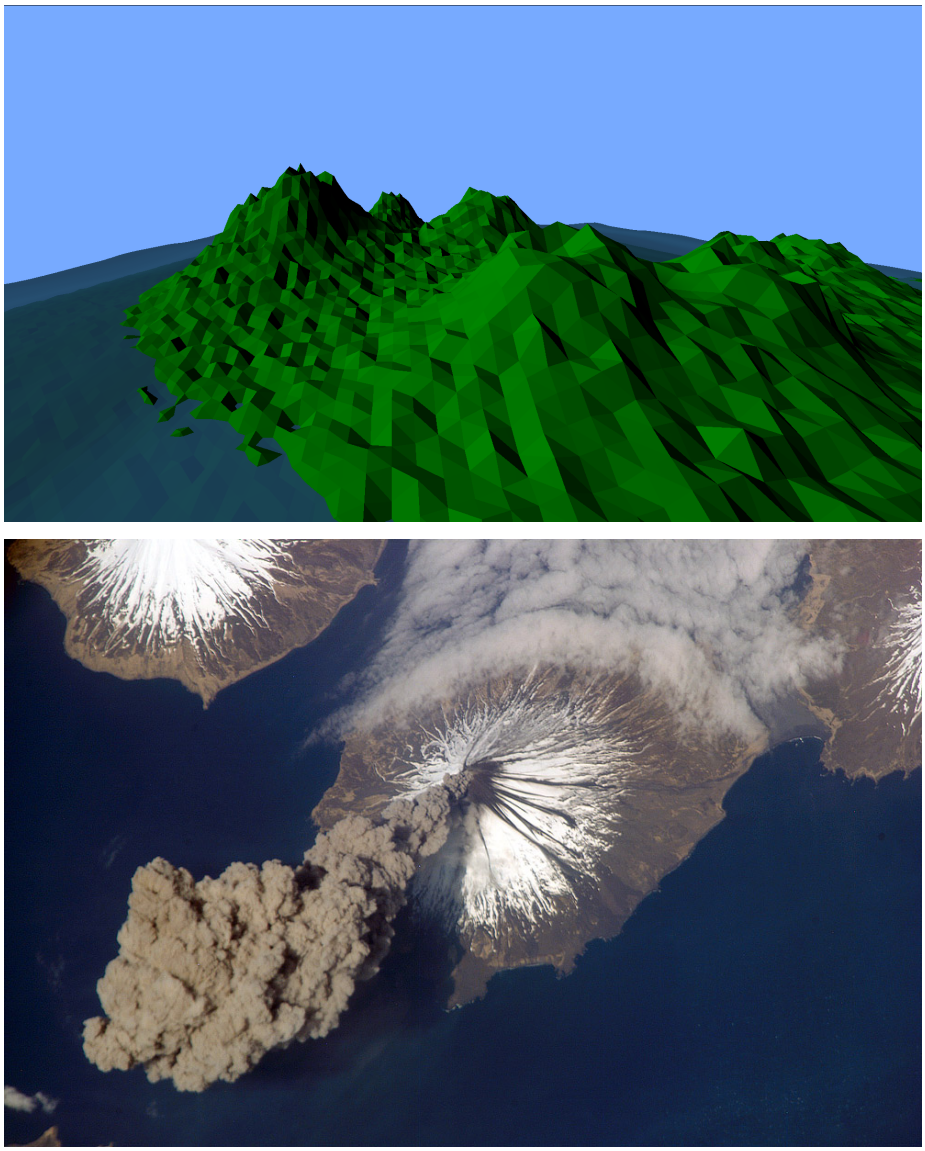
```
meshVertex.y += 0.01;
```



The mountains generated have a various shapes, caldera, ridges, pointy etc. As islands they show convex and concave features around their edges.







ISS Earth Observations experiment and Image Science & Analysis Group, Johnson Space Center. Public domain via Wikimedia Commons

Producing accurate landscapes is complex and computationally expensive, noise systems difficult to tweak. Approximations are lightweight, flexible and easier to understand. The maths is simple trigonometry and random numbers. Simple algebraic simulations close the 'Gulf of Execution' associated with procedural generation.