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

This improved feeling comes at the risk of committing procedural blasphemy[1]. Many would be right in saying Perlin noise is incredibly useful. Its also quite 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. Perlin & similar systems generate these structures easily because they produce a repeating output that is never quite the same. The variations in the output can all be a similar scale, allowing us to use it as a source for hills or islands etc which are created by the same geological processes.

One way to influence the characteristic 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, https://commons.wikimedia.org/w/index.php?curid=4152613

[1] A better procedural blasphemy might be an 'electric sinner'. Use AI, logic and natural language processing to codify the core rule set from a religious text. Build a machine to break those rules (in thought, by visualising them, then perhaps automatically tweeting the images, by communication by inciting others to commit sin and finally in action, would need a robot for this one). Is it possible to carry out any of these activities without committing a sin oneself? I suppose we could consider it educational or somewhat like a morality play. (some might say that video games are already doing this)

[2] I'm just going to call all the value noise systems Perlin noise avoid sentence mangling.

Webmaster.vinarice [CC BY-SA 4.0], via Wikimedia Commons

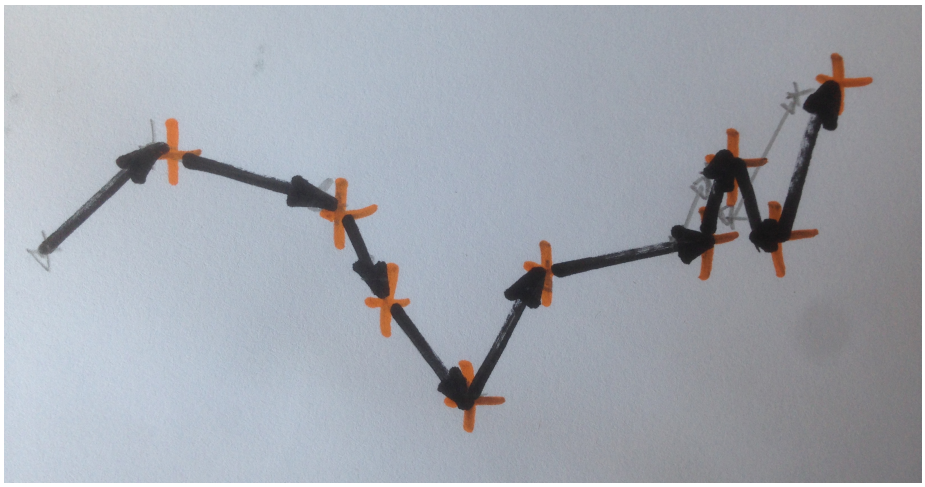
High end procedural generation of sand dunes can be complex. Producing landscapes which although authentic may not appropriate for gameplay. Journey's dunes were hand designed landscapes, because its gameplay relied on them. With extra sand and light rendering to add interest. Conversely in Meteor Storm Escape we included a desert dune racing level, at every step of development we compromised any idea of authenticity for a particular 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 and mathematically replicate the internal states and dynamics of the system found in reality whose aspects we find valuable for our game.

Plate tectonics causes 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)
```

Increasing *constAngle* causes the path to be more wriggly. Decreasing it will push the random walk off in a direction.



This approach is entirely in-authentic in that it bears no relationship with the forces involved in the joining or separating of two tectonic plates. However it *can* easily be tweaked to produce sequences which either 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 vague 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. Play around and see what you get.

We could run an engine's physics system and generate lots of particles. A good intensive solution, and not always an option on web or mobile. We can do the same thing algebraically with one line of code. We can calculate the approx landing point of the lava which is thrown out of the volcano using a simple equation.

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

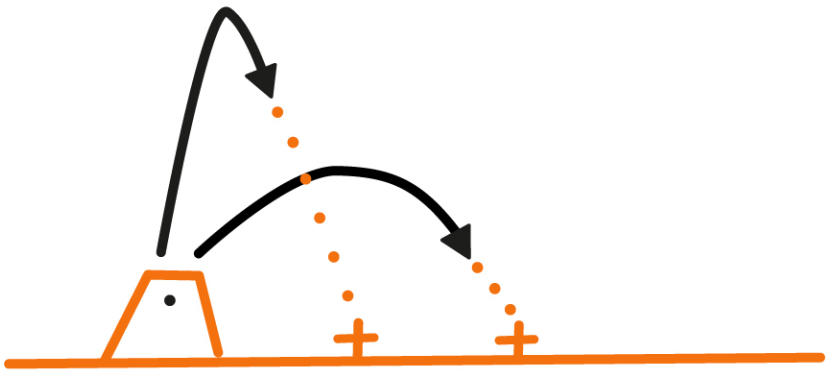
The distance away that something lands is proportional to the square of its velocity times the sine of the angle it is ejected at, divided by the gravity. This assumes the ejection point isn't higher or lower than the landing point. For our purposes these simplifications are fine. Then GCSE trigonometry will translate the distance and direction into a vector offset from the centre of the volcanic island.

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

To which we can then do a little addition on to get a final global position of where the ejecta lands.

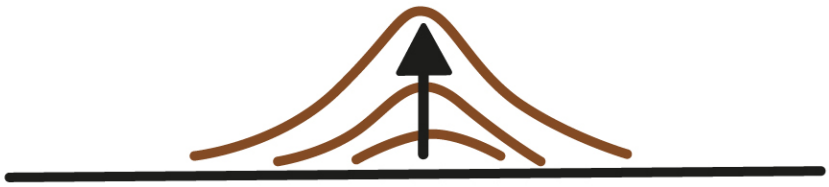
```
var coords = new THREE.Vector3();
coords.x = Math.floor(centre.x + (deltax * (this.size * 0.5)));
coords.z = Math.floor(centre.z + (deltaz * (this.size * 0.5)));

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

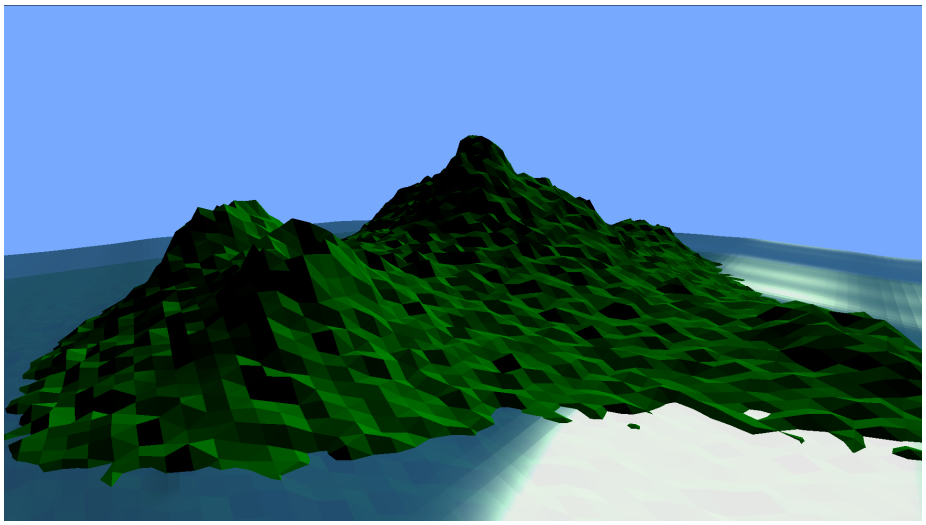


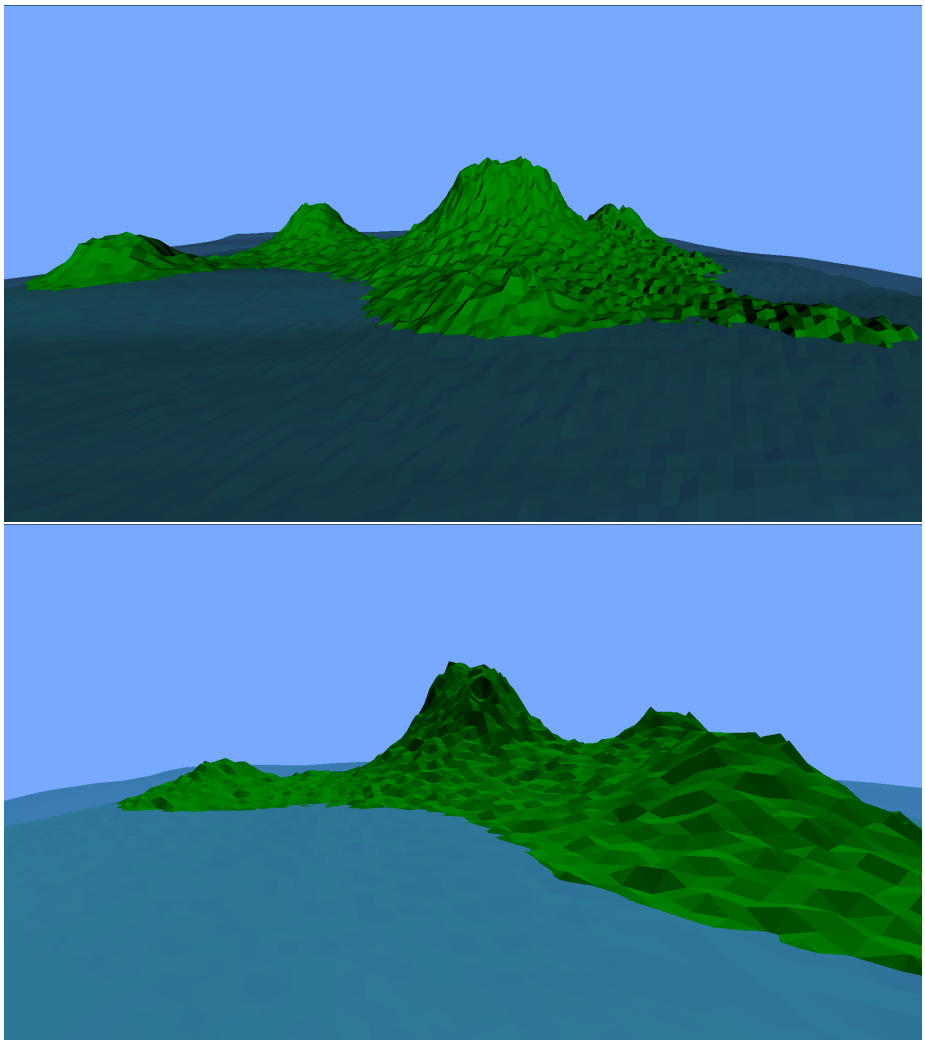
The accumulation is incredibly simple, just raise the vertex nearest to where it lands!

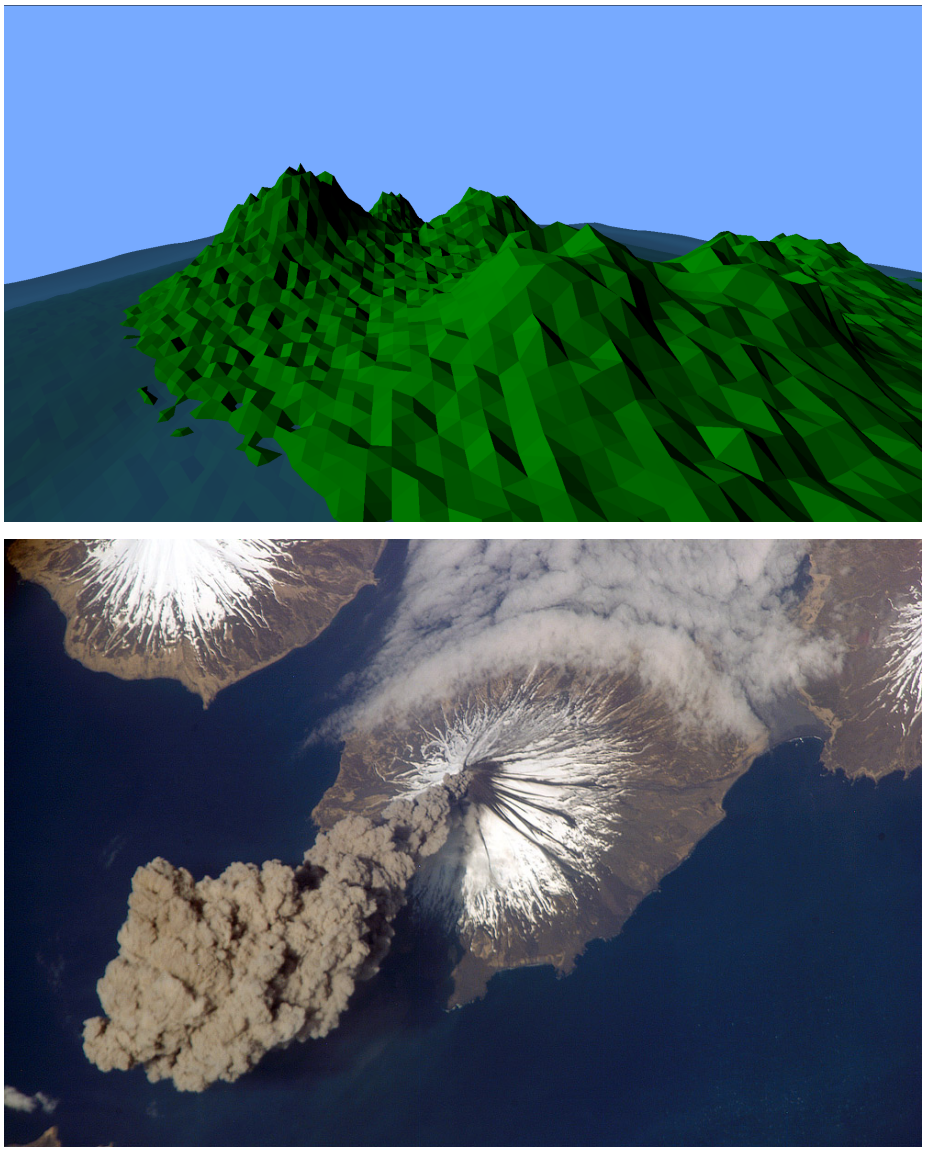
```
meshVertex.y += 0.01;
```



The mountains generated have a various shapes, caldera, ridges, pointy, jagged etc. When in water the islands generated show convex and concave features around their edges. Giving authentic natural landscapes.







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

Algorithms producing realistic landscapes, can be complex and computationally expensive. Perhaps we build approximations which are lightweight customisable, flexible and easy to understand? The maths involved is simple relying on GCSE trig and random numbers. The system is understandable and tweakable. Metaphorical physical simulations (even algebraic ones) close the Gulf of Execution commonly associated with complex procedural systems.