Throwing Things

I DON'T LIKE PERLIN NOISE There I said it. I feel better already!

This improved feeling does perhaps come at the risk of committing procedural blasphemy[1]. Many procedural developers would say Perlin Noise is incredibly useful and, they'd be right. Its also quite complicated and for some opaque.

As a multilayered, scaled, random-ish waveform Perlin Noise can be put to use in many ways including Clouds, Waves, Islands, Caves etc. Perlin & other similar noise systems generate these kinds of structures easily because they can produce a repeating output that is never quite the same. A valuable aspect of the result is that the variations in the data can all be a similar scale. This similarity allows us to use the output as a source for hills or islands etc which should all be different, but are created by the same processes.

However, in their vanilla form, the landscapes from Perlin noise tend to have a characteristic shape. Luckily we can influence that shape. Most developers put in additional manipulations to create enjoyable geographic structures. One way is to gradually increase scale on the y-axis. With the desirable effect of making the higher points of the landscape more pointy. Giving a profile which matches the mountains and hills all around us.

Public Domain, https://commons.wikimedia.org/w/index.php?curid=4152613

[1] A better kind of 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 actually 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)

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

Academic or scientific approaches to procedural generation of sand dunes often have the goal of generating natural shaped landscapes. When accurate they can be extremely complex, and result in generation of landscapes which although authentic are not appropriate for gameplay. Journey has extensive dunes and because its gameplay relies on them, uses hand designed landscapes with interpolation and mathematically complex rendering of sand and light to add visual and emotional interest.

In Meteor Storm Escape we included a desert dune racing level among others, at every step of development we compromised any idea of authenticity for a particular challenging and exhilarating player experience.

As developers we all have a choice of strategies, from arbitrary algorithms and heuristics which generate forms useful for gameplay or other reasons, 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.

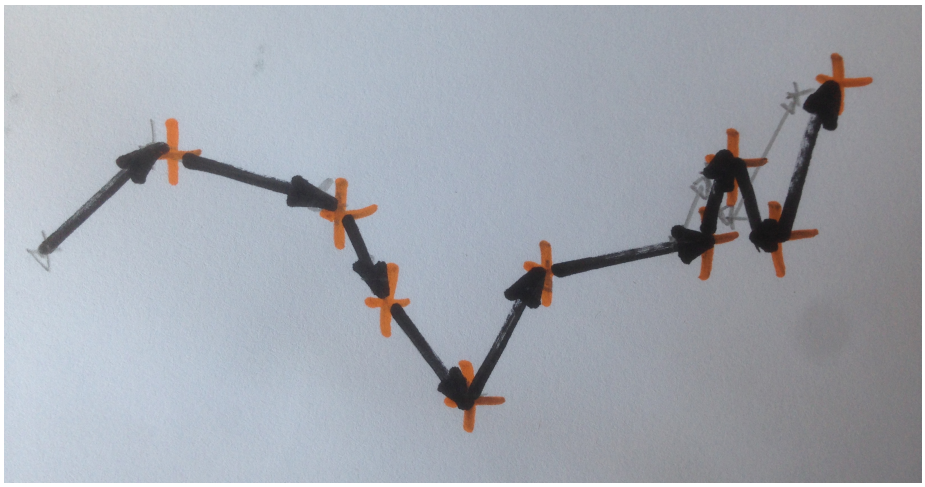
An (In)authentic Algorithm to Generate Oceanic Landscapes With Realistic Contours and Gameplay Fit

Due to plate tectonics the parts of the earths crust where volcanos form can be long wiggly lines. Simulations of plate tectonics exist, though I've yet to hear of a game which uses it to determine gameplay[2]. Its simple enough with a move-turn, loop to manipulate a sequence of vectors to steer a random walk in a particular direction. Parameterising the length of the vectors and size of change in orientation, we can create a more or less linear path to represent the place where two parts of the world's crust are moving together or apart.

```
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* to a full circle will cause the path to be so wriggly it becomes a pure random walk. Decreasing it to a smaller number will push the random walk off in a direction.

[2]If anyone knows a bit plate tectonics please, get in-touch, perhaps we can work together?



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.

Throwing out stuff to build the landscape. In a gross simplification lets assume that the direction and velocity of ejecta is completely random, but the random distribution of the angle of ejection follows some variant of 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. Play around and see what you get.

One could run an engine's physics system and generate lots of particles. A good solution but intensive 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 ) ) / 10
```

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. (The value of 10 is an approximation of gravity which is 9.8 m/s). This assumes the ejection point isn't higher or lower than the landing point. For our purposes these simplifications are fine.

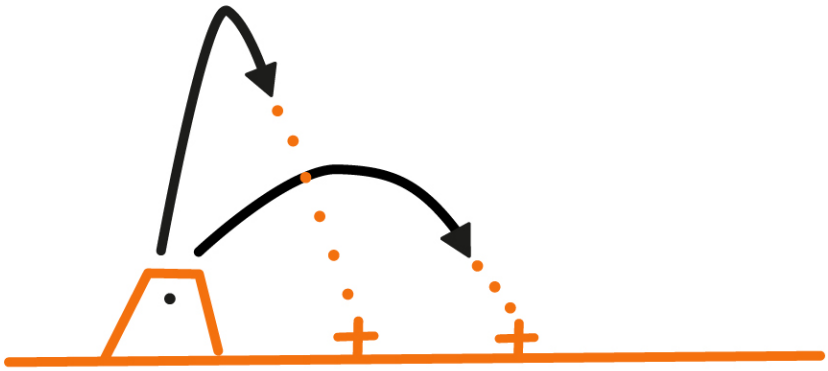
Now, GCSE trigonometry will translate the distance and direction into a vector

offset from the centre of the volcanic island.

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

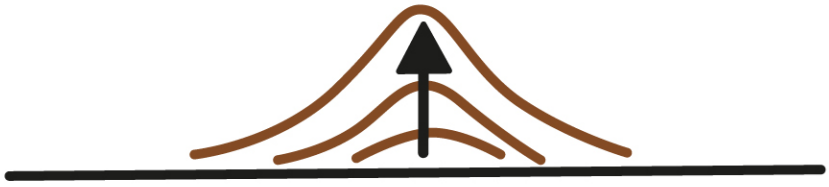
To which we can then do a little addition on to get a final global position of where the ejecta lands.

```
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;
```



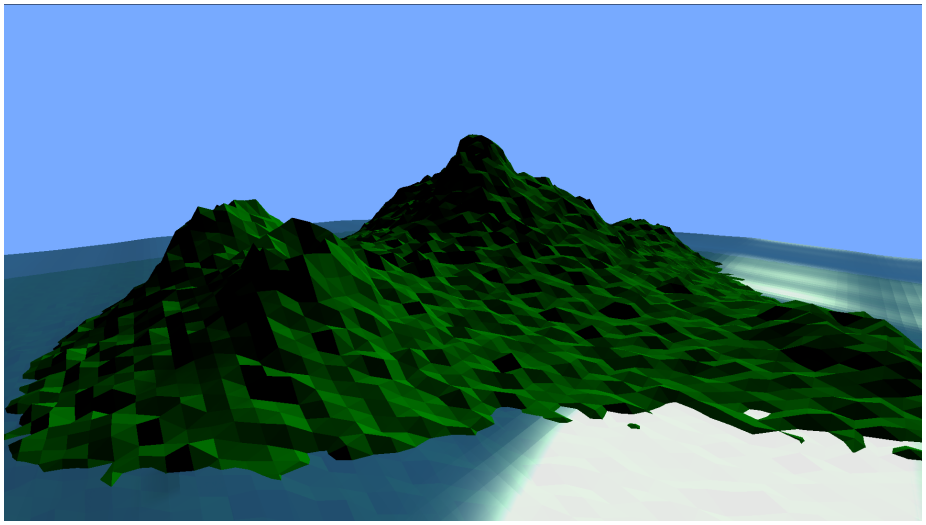
As you can imagine our accumulation is incredibly simple, raise the vertex nearest to where it lands!

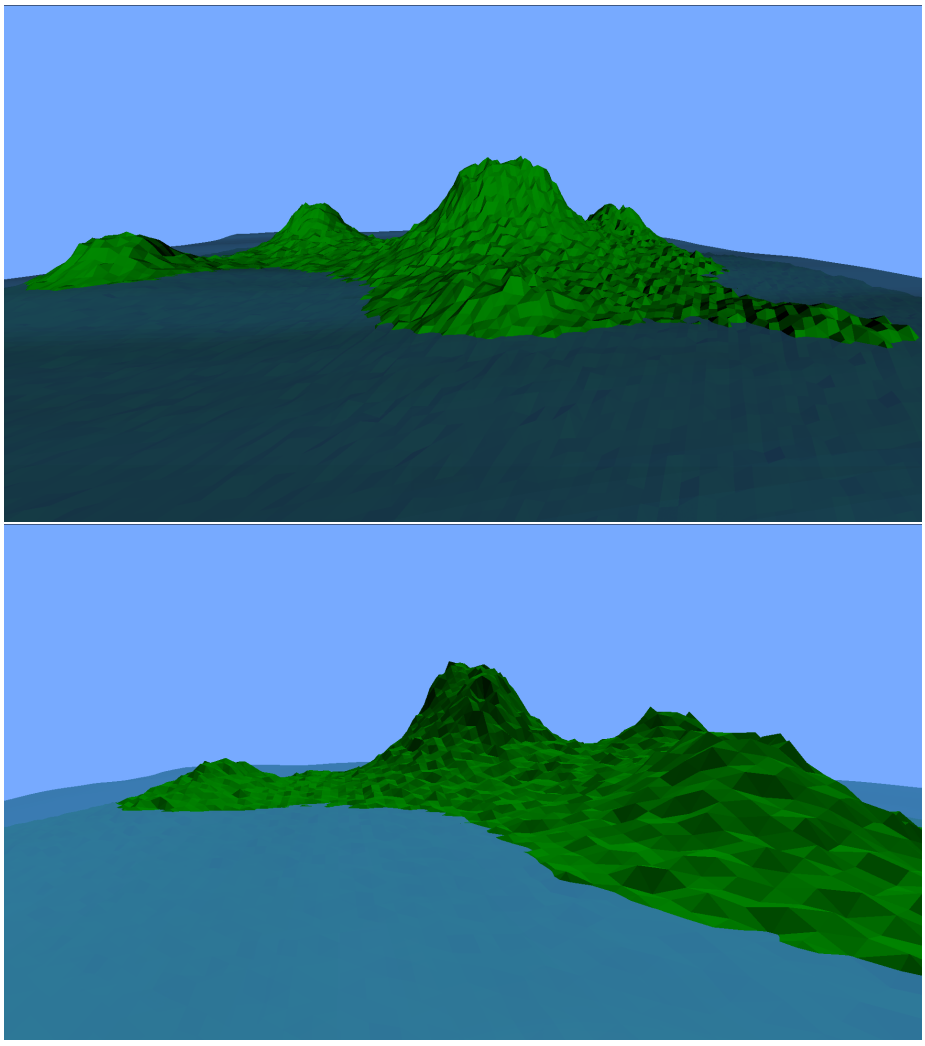
```
var meshVertex = new THREE.Vector3();
meshVertex.fromBufferAttribute( positions, meshVertexIndex );

meshVertex.y += 0.01;
```

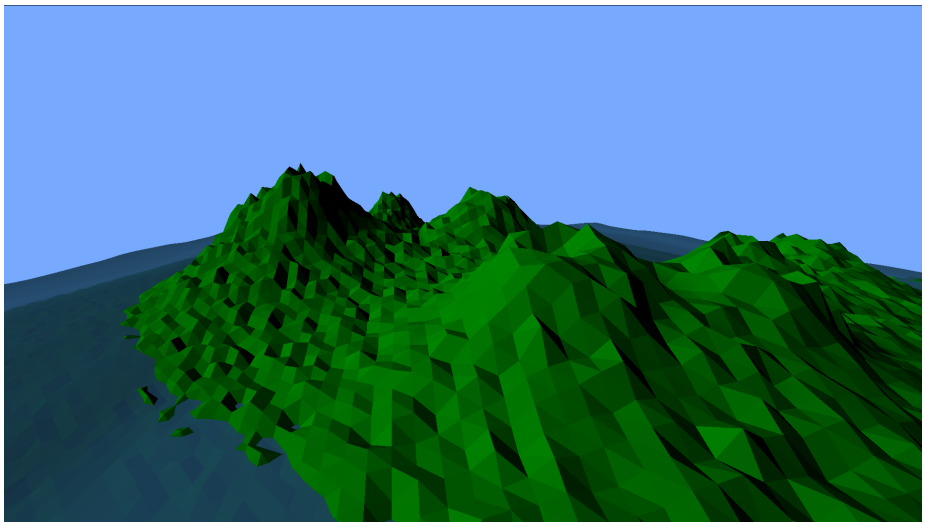


Discussion

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







To focus on the shape generated, we havn't rendered with specialist shaders.

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

The aesthetics of space and visuals in games is different from that in the traditional visual arts. As game developers we must balance the needs of the visual form against the needs of the gameplay form.

In big studio production this is often done by having white/orange boxing and

by having gameplay collision mesh built in parallel with the gameplay design separate process from the (later) art generation.

In procedural generation the management of this dual nature has to be handled differently. Often procedural generation closely ties the visuals and gameplay space.

Researchers have developed many high end algorithms producing authentic natural landscapes, which are complex to implement and computationally intensive.

For an easy approach to landscapes, go for a hybrid approach, build arbitrary authentic algorithms. Passable approximations are achievable which are lightweight customisable, flexible and easy to understand.

The maths involved is in comparison simple relying on GCSE trig and random numbers. The system is tweak-able by a human designer. While each input value is metaphorical, they have a readily understandable effect on the output. Metaphorical physical simulations (even algebraic ones) are an improvement over traditional noise based approaches because they close the Gulf of Execution commonly associated with complex procedural systems.