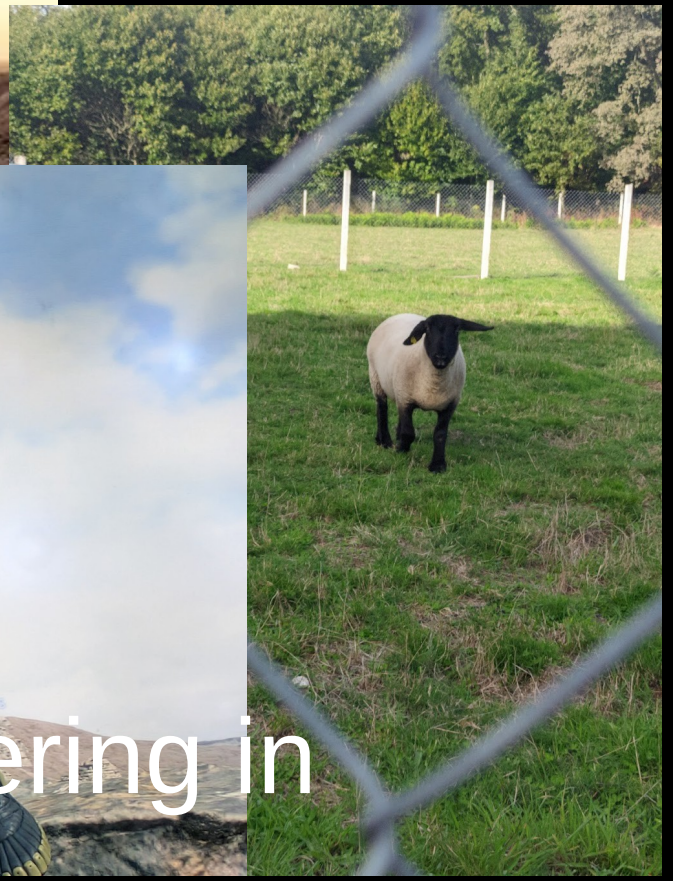


Terrain generation & rendering in Laniakea

AKA touching grass but with less budget

Wot’s a Laniakea?

- Laniakea is a game about a data smuggler and his chud daughter, who travel an interconnected network comprised of various worlds.
- While the game is mostly story-driven, procedural terrain is used in some parts of the game.

Why use procedural terrain generation?

- Your game is endless
- You want to save time
- You’re spanish (have no money)
- Even big games use procedural generation a lot these days, do you really think someone from kojima productions sculpted every single hill in Death Stranding?

The three schools

Terrain generation is broadly divided into three schools of thought:

- Square voxels
- Marching cubes (fancy voxels)
- Heightmap ← We are focusing on this today

The objective of terrain

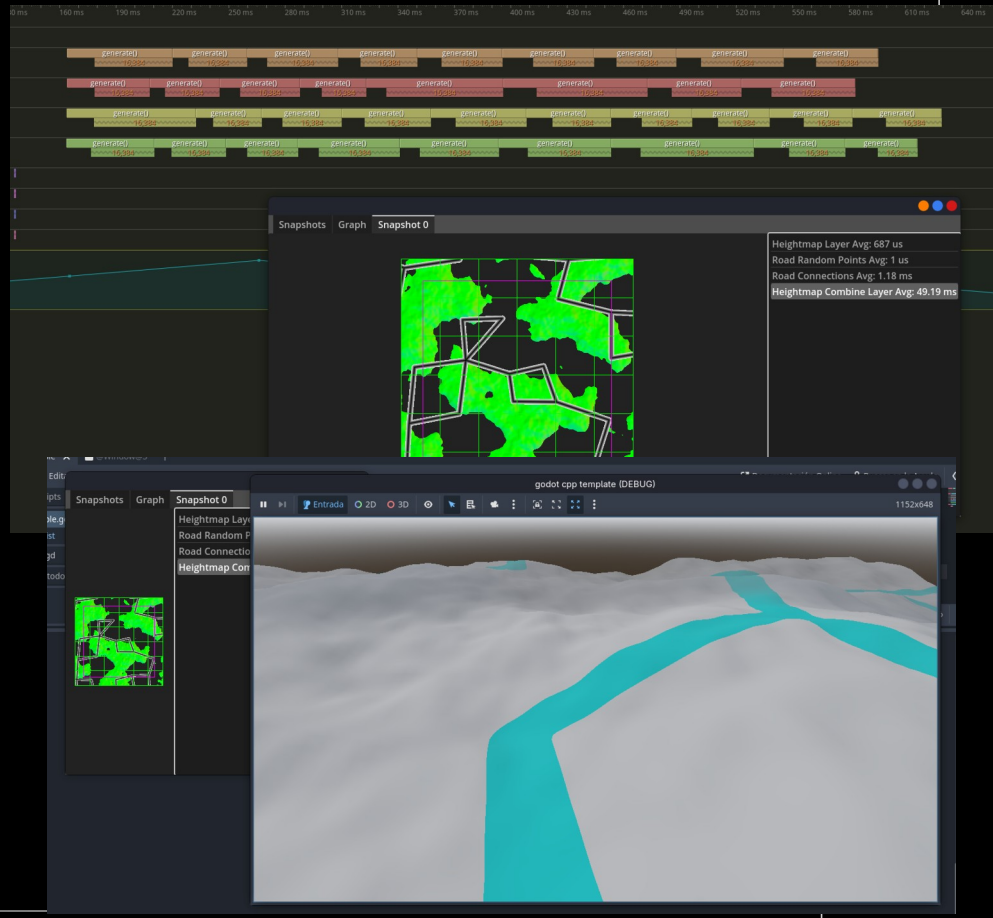
- Conceptually simple, grab data about the terrain and turn it into something you can draw and interact with.
- We usually divide the world into separate chunks that can be generated in parallel.

Storing *xel data

- Usually a scalar field represented as a grid, 2D or 3D
- For heightmap terrain it can be stored as a texture, generally single channel.
- The base geometry (usually a plane) can be transformed based on the texture...

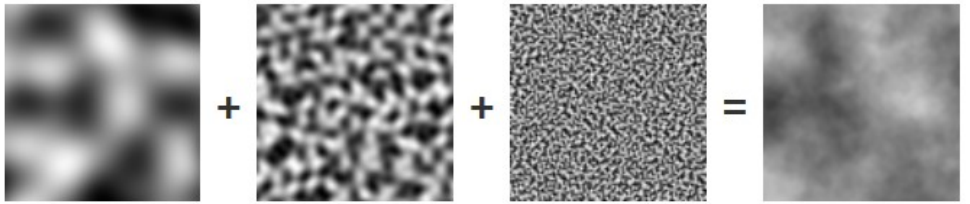
Layered generation

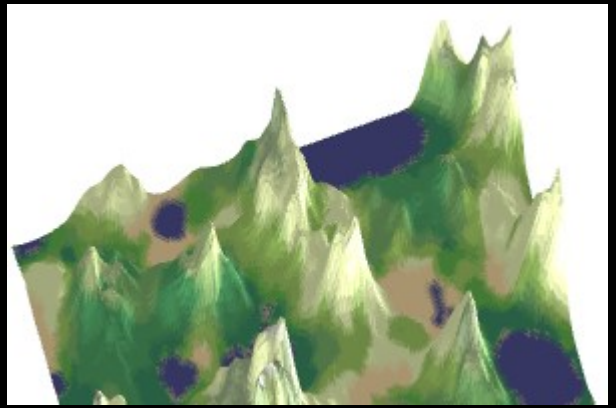
- While the final terrain will have a fixed chunk size in a grid, we can have a layer system that automatically calculates only the information we need, and we can allow every layer to have a different chunk size depending on the workload we want.
- Chunks on each layer can also have a “padding” around them, in case they need information from surrounding chunks.
- We will use our old friend the Directed Acyclic Graph to generate dependencies between tasks and decide which ones we can do in parallel.
- We can travel up the tree from the leaves to figure out which “rects” need to be generated, and generate missing chunks on demand.



How to generate the terrain height

- Use layered noise, with various operations to make it look interesting.
- Some things you can do are:
  - Remap a noise using a designer-defined curve.
  - Grab the noise value (in -1.0-1.0 range) and square it.
  - Use fractal brownian motion to make it more detailed.
  - Warp the generated noise using another noise (domain warping) etc.



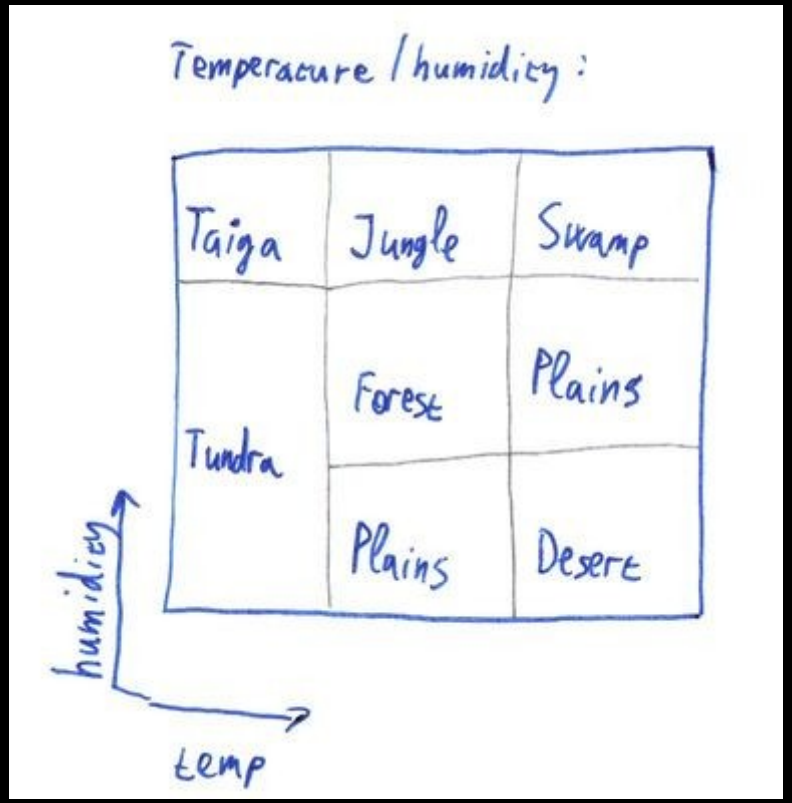


Two ways to do biomes

- There are two big schools of thought for generating biomes:
  - The first is generating the biomes using something like a distorted voronoi diagram BEFORE we generate the heightmap.
  - The second is generating the heightmap and generating the biomes AFTERWARDS.

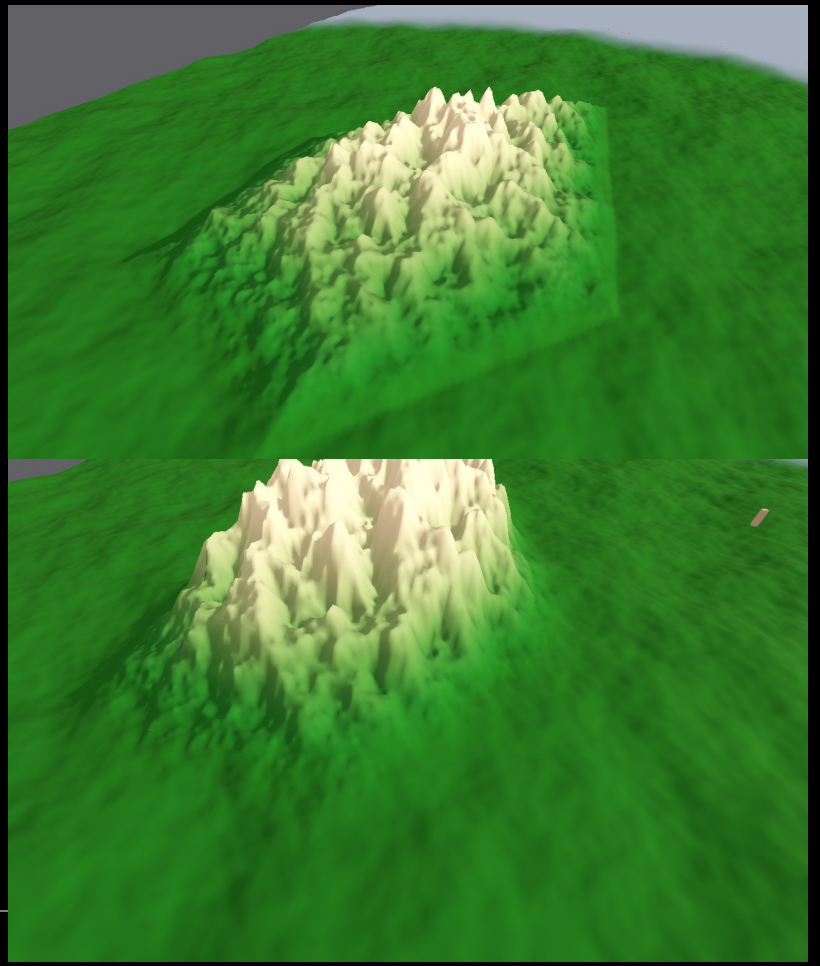
Ensuring biome locality

- It would make no sense for a snowy biome to get generated next to a desert biome.
- To solve this, instead of deciding each biome by throwing a dice, we have the biomes be picked based on certain smooth noise values and we pick using a multi-dimensional map.
- For example, old minecraft had humidity and temperature, new minecraft has continentalness, erosion, peaks & valleys and weirdness maps, which are also partially used for the heightmap.



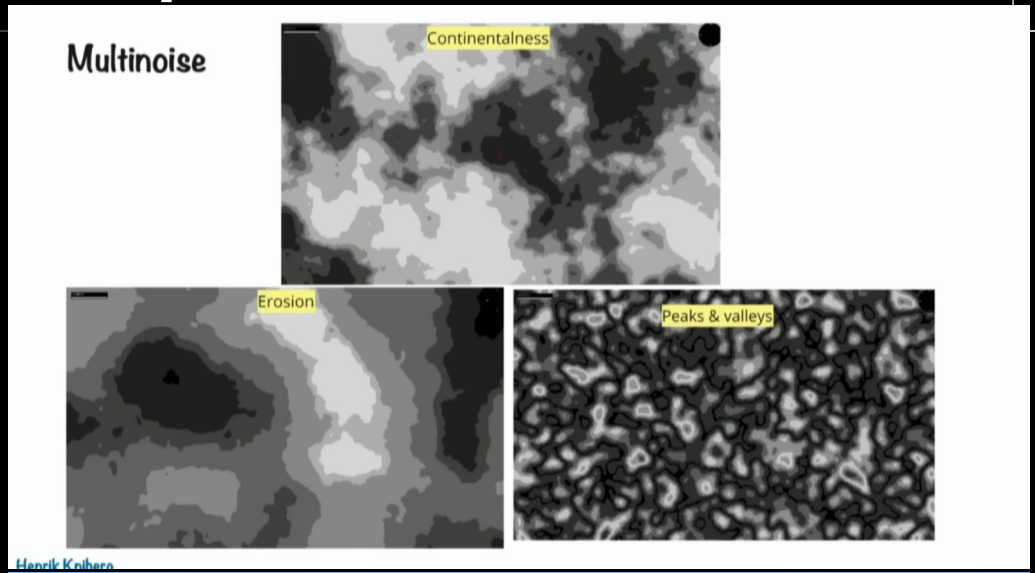
Biomes before heightmap

- The advantage of this method is that we can have wildly different terrain for biomes.
- The disadvantage is that it’s more computationally expensive, we have to find the closest biomes at every pixel (can be sped up using a delaunay triangulation of the biome map). And we have to interpolate between them, I found exponential interpolation to hide it pretty well.
- This is what old minecraft used to do!



Biomes after heightmap

- We decide the biomes of a certain *xel using other noise maps.
- This is faster, but gives less control and is much harder to define, it also generates less interesting terrain.
- This is what modern minecraft uses. This method works well if you have many, many different biomes!



Virtual textures

- In most cases, you’re not allowed to sample different textures altogether inside the same drawcall, it must be dynamically uniform at best.
- However, you’re allowed to sample different parts of the same texture, so we can work around this limitation using array textures!
- Essentially what we have is a 2d “page” texture where each pixel maps 1:1 to a chunk around the camera, we can then grab the index contained inside that pixel and use it on our texture array.

Why sample textures outside our chunk?

- Sometimes it’s benefitial to cross chunk boundaries.
- Some uses cases include shadows and normal calculation.
- Ideally we would like to just grab a world position and automatically figure out what texture in the array we need to sample

Calculating normals

- We need normals for lighting!
- Storing normals in textures would multiply by three our memory usage (if we are using tangential space normals) or by four (if we are using world space).
- We could calculate them analytically, but when you stack different noises on top of eachother it becomes a pain in the ass.
- Just use finite differences.
- I tried to use screen space derivatives but I couldn’t get it to work properly… Too much banding.

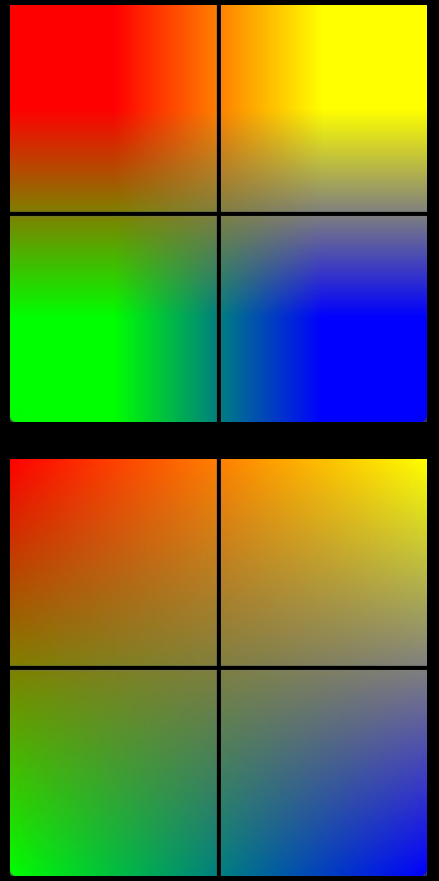
```
const double SAMPLE_NUDGE = 50.0;

const double hL = layer->sample_height(point_to_sample - Vector2(SAMPLE_NUDGE, 0.0));
const double hR = layer->sample_height(point_to_sample + Vector2(SAMPLE_NUDGE, 0.0));
const double hD = layer->sample_height(point_to_sample - Vector2(0.0, SAMPLE_NUDGE));
const double hU = layer->sample_height(point_to_sample + Vector2(0.0, SAMPLE_NUDGE));

// deduce terrain normal
Vector3 normal;
normal.x = hL - hR;
normal.y = SAMPLE_NUDGE*2.0;
normal.z = hD - hU;
normal.normalize();
```

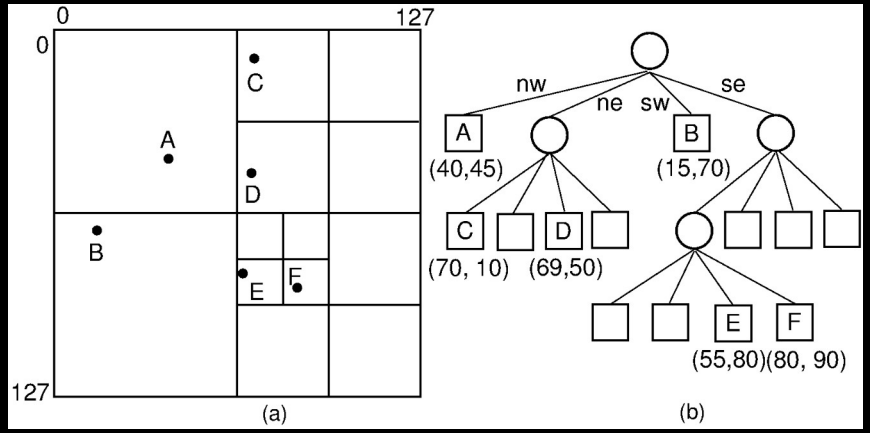
Note about sampling textures

- In OpenGL/Vulkan the origin of each pixel is at the center of pixels, meaning a linearly sampled texture has a half pixel border of the color of the outermost pixel, not good!
- In order to compensate for this, we must remap our sampling coordinate so they start at the center of the top leftmost pixel and end at the center of the bottom rightmost pixel.



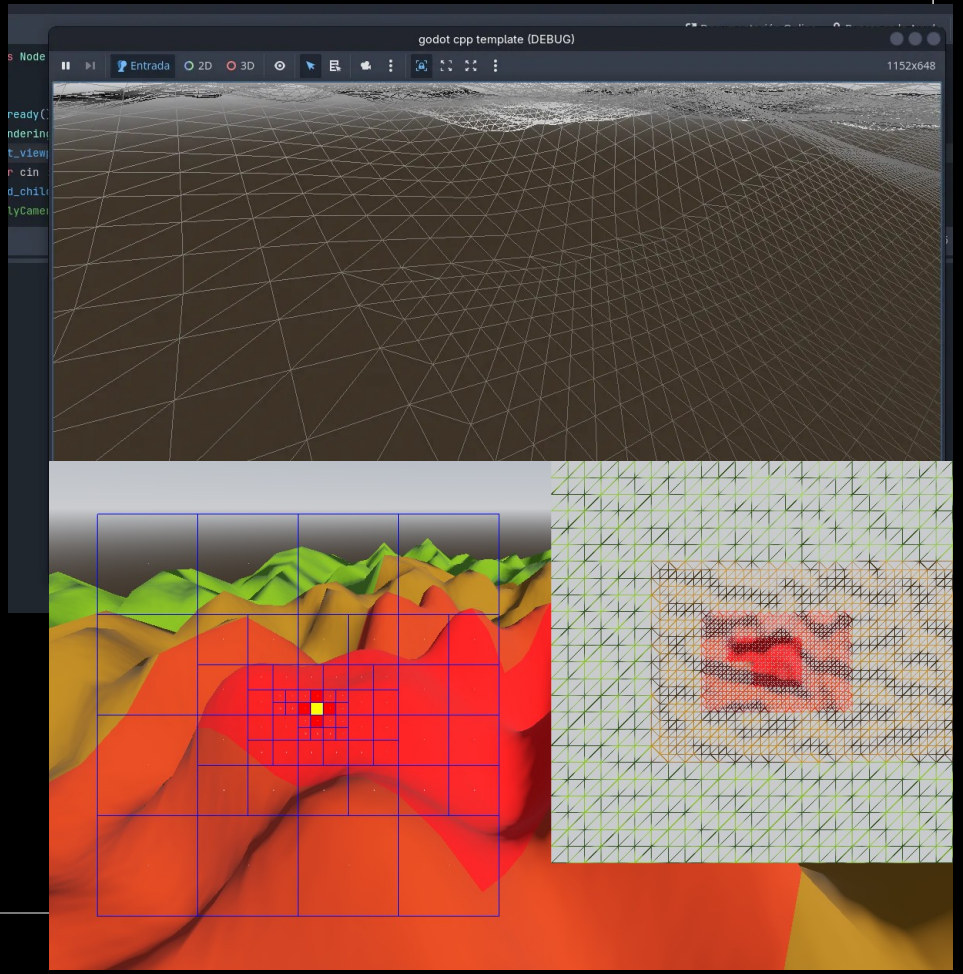
Mesh LOD in Laniakea

- It would be not-very-good if up close chunks had the same amount of geometry as far away chunks, we are wasting performance for no good reason!
- But we would also like to make transitions relatively smooth.
- Enter the humble QuadTree!



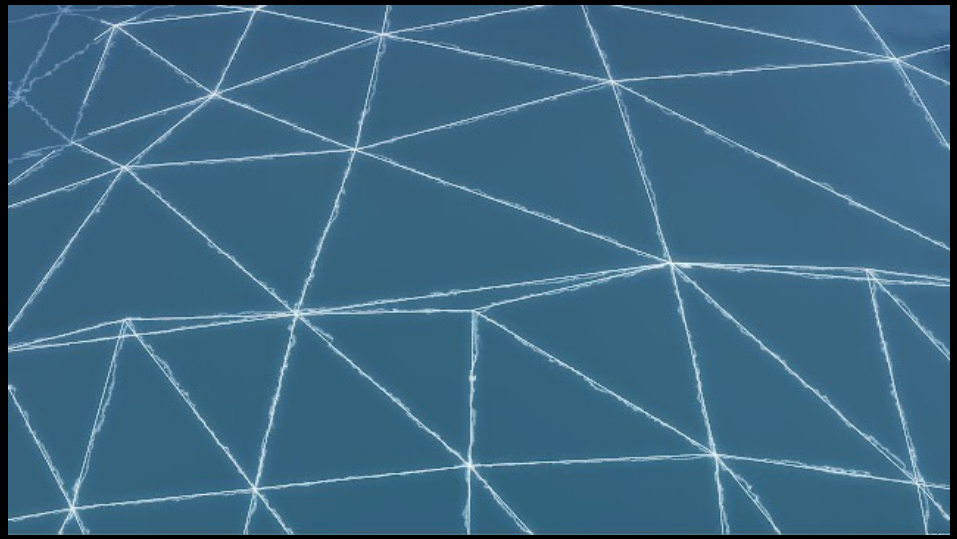
Mesh LOD in Laniakea (II)

- First we generate a chunk (called a superchunk in laniakea).
- Every few frames, we will grab each superchunk and attempt to insert the camera into an empty quad tree. We will use this to decide which meshes (called subchunks) we need and what size they should be.
- Note this is purely for geometry, the textures remain unchanged!



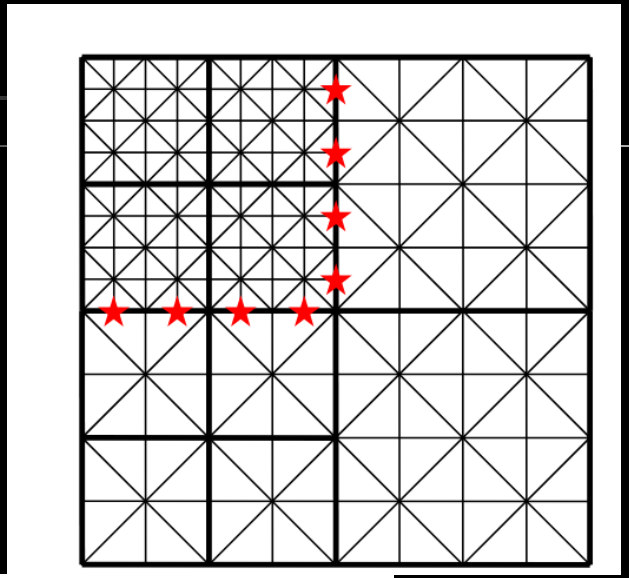
The problem of seams

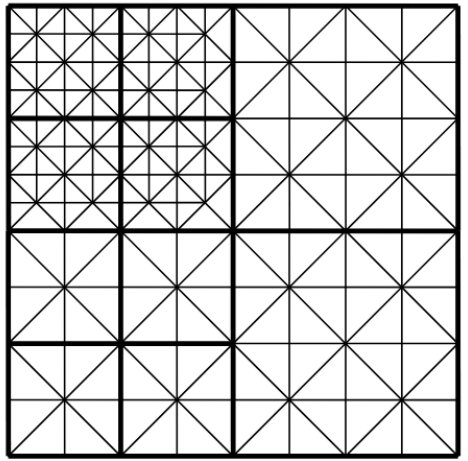
- The problem with this method is that it generates seams, discontinuities at the boundaries between different LOD meshes.
- This looks baaaad.

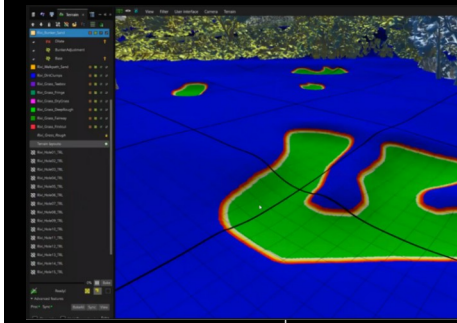


The frostbite solution

- What we can do is constraint our quadtree to ensure the neighbors of all nodes are at most 1 LOD level in difference.
- We can then use different meshes for those tree nodes, we need 9 permutations. (Although in laniakea the geometry is modified on the GPU by creating degenerate triangles).
- I stole this from frosbite.

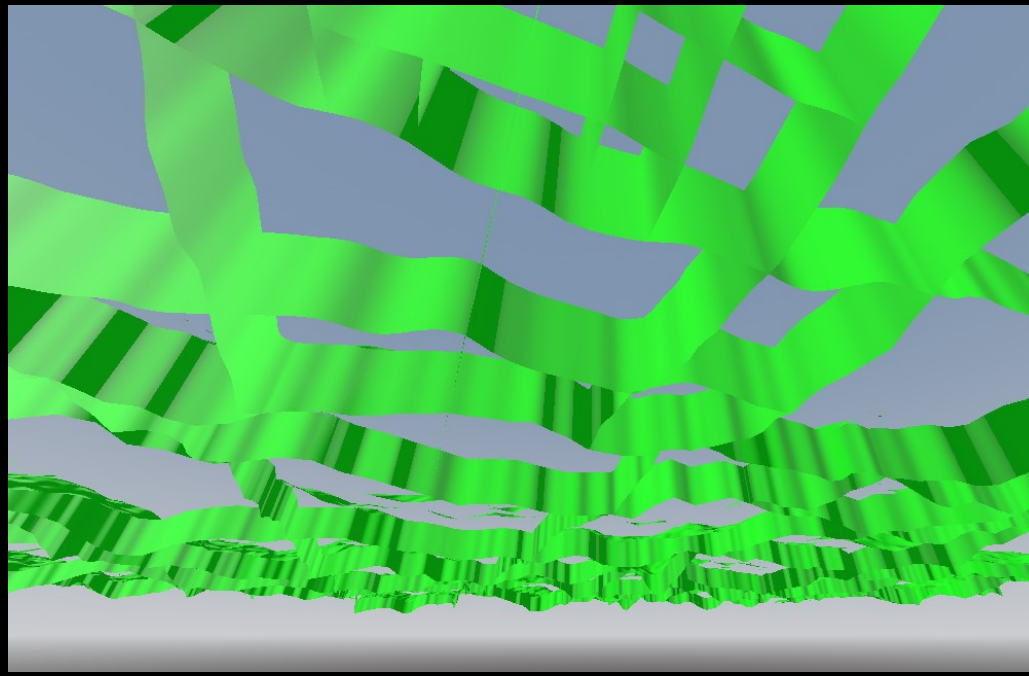






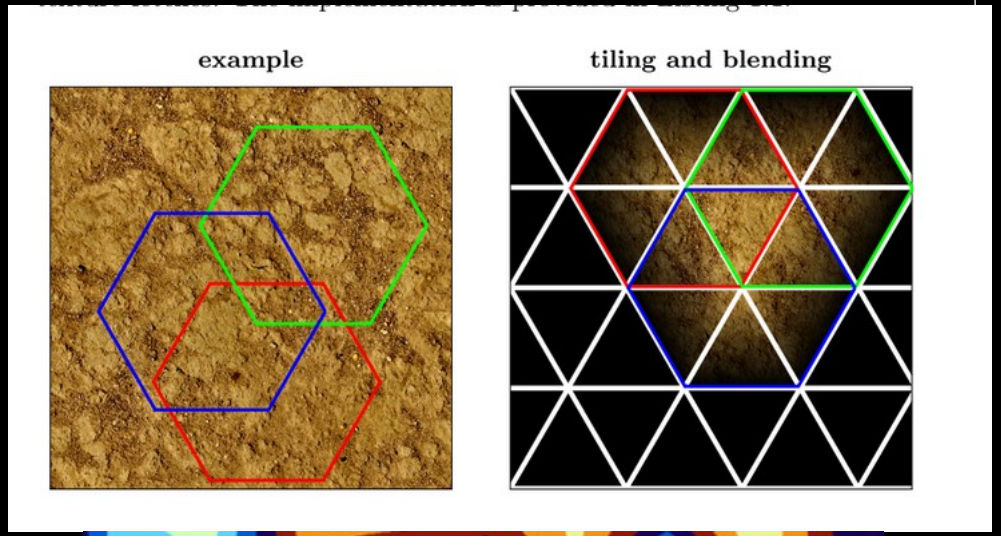
Handling superchunk boundaries

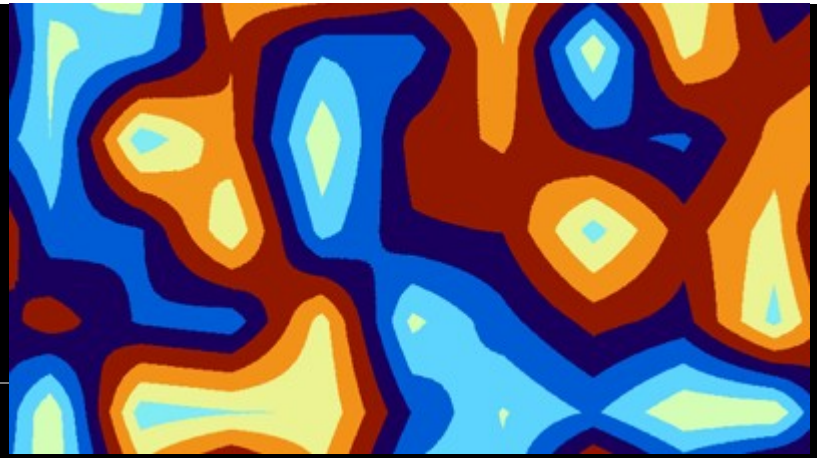
- One problem with this approach is due to the world being broken into parallel superchunks is that we may not have proper transitions between them, which causes seams.
- A solution is to add skirts, which are vertical downwards facing faces that hide the seams at a distance.
- Ideally we should add these only where needed, aka on subchunks at the boundaries of superchunks (bad example pictured).
- This causes a bit of overdraw (drawing to the same pixel more than once) but it’s probably fine...



Texturing terrain

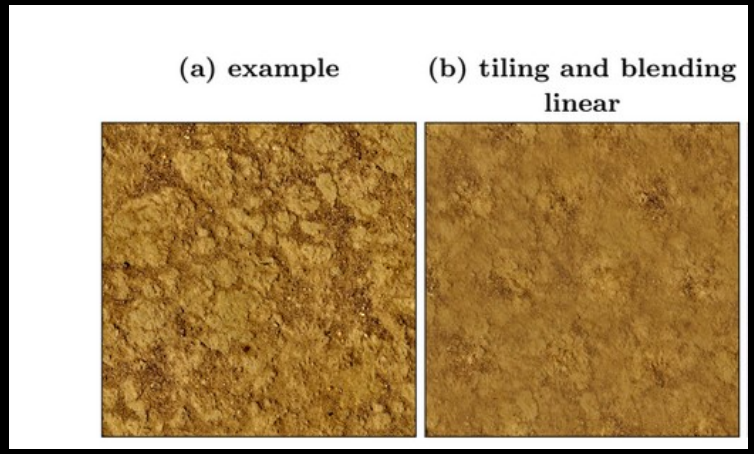
- Texture tiling is ugly :(
- There are many ways to go about this.
- I have tested two options that seem similar, one is to use a triangular grid-like structure, where tiles overlap and each tile has a random orientation/scale of the texture contained that we then interpolate overlapping tiles.
- This first option is called stochastic texturing.
- Another option is to use a noise and quantize it, interpolating everything around.





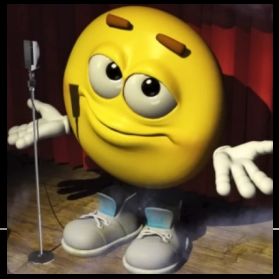
Interpolating terrain textures

- For any of the previous techniques to work we need to interpolate textures, linear interpolation isn’t very good, since it would make us lose a lot of detail.
- The original stochastic texturing paper does some complicated math to ensure histogram preservation.
- However, if we have a texture displacement map, we can just use a height blend operator!
- Notice how the grass appears on the lower parts of the rocks, this is pretty good, isn’t it?



How 2 decide which texture to use

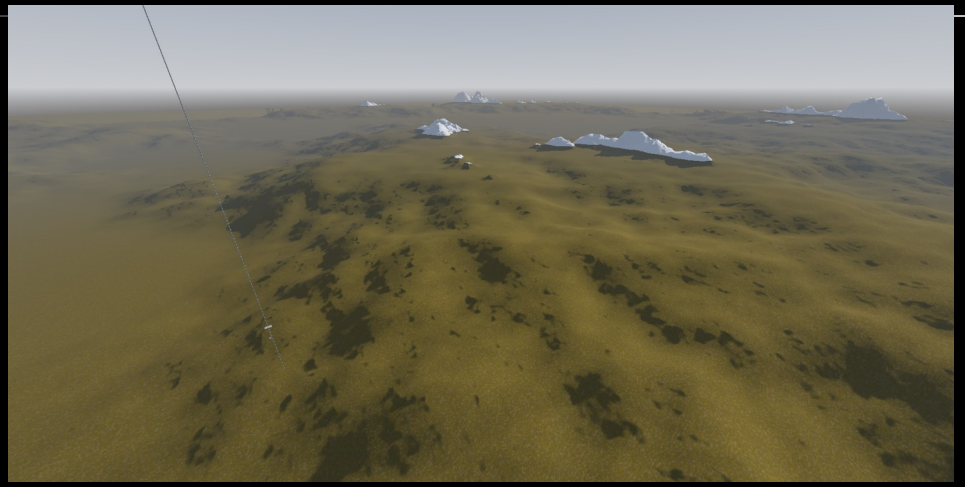
- One solution I’ve seen is packing data into the texture, since we won’t have many terrain textures we could totally use a single 16 or 32 bit integer channel texture and pray that our bitshifting math is good.
- We should actually prefer branching to sampling more textures for things like triplanar mapping if we can avoid it, however in GLSL the shader compiler may not force real branching.

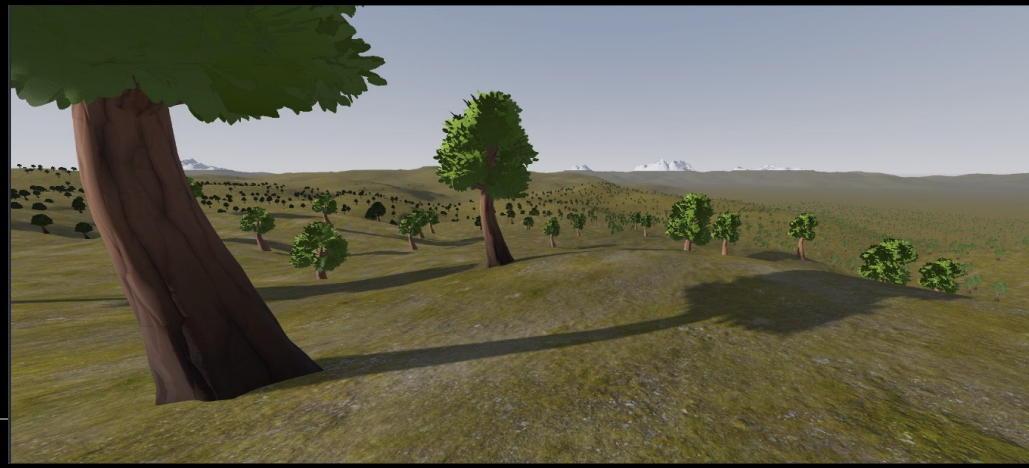


Shadows

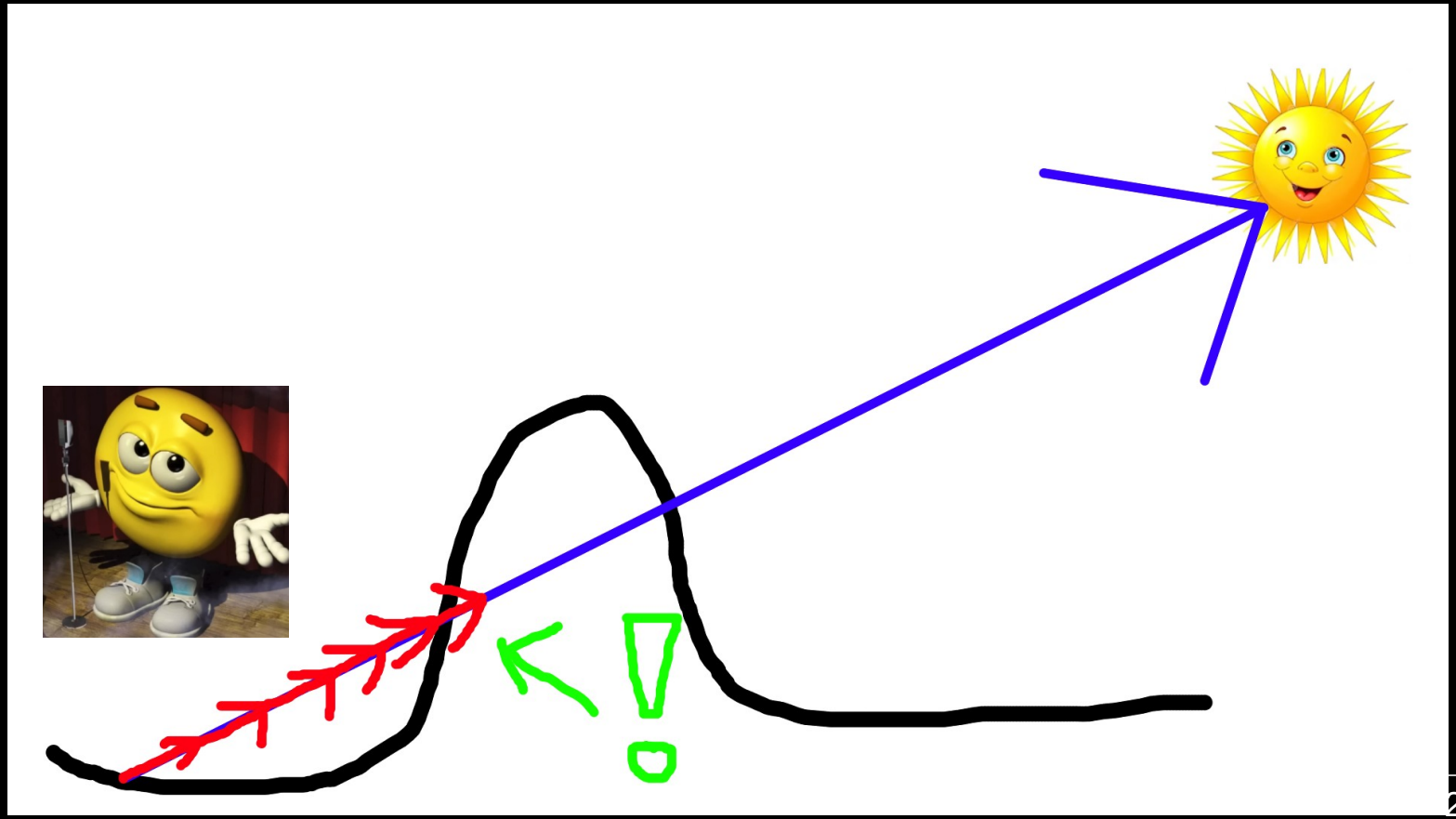
- We could use shadow mapping like a normal person
- But unfortunately our engine (Godot) sucks and it would be hard to do.
- Fortunately, modern GPUs are super fast, so we can just use ray marching*

  * While up-close the normal cascaded shadow maps take over, we may actually want to render to a low resolution shadow map just so we can cover very far away meshes that are not part of the terrain without needing to raymarch for every single one of them.





How 2 raymarch terrain shadows



Potential improvements

- Use bicubic filtering (nicer looking!)
- Move more things to the GPU.
- Use an SIMD noise library for higher throughput.
- Perhaps study a solution that does not require the superchunk skirt hack. Clipmaps are an option, but they don’t allow for frustum culling to the same degree superchunks do (but it may be fine anyways).

References

- Stochastic Texturing by Jason Booth
- Procedural Stochastic Textures by Tiling and Blending by Thomas Deliot and Eric Heitz
- Generating terrain in Cuberite
- Reinventing Minecraft world generation by Henrik Kniberg
- Texture Repetition by Inigo Quilez
- Terrain Rendering in Frostbite Using Procedural Shader Splatting by Johan Andersson