

# Practical 7

---



Intro to Computer Graphics

# Project Part 1

# Procedural Terrain

# Demo from Last Year



# Demo from Last Year



# Project Announcement



- 3 parts, ~2 weeks per part
  - Procedural modeling
  - Rendering/texturing
  - Animation/interaction/particle systems
- No more solutions from now on
- TAs are around every Thursday afternoon for support
- Code base: take any of the HWs/practicals used so far

# Timeline (might change slightly)

Week	Practical/Project
7	Project Part I start: procedural modeling
	Easter break
8	Work on project
9	Invited talk: Pascal Mueller, Esri/CityEngine, 45min Project Part 2 start: rendering / texturing
10	Shadow mapping tutorial Intermediate feedback session
11	Project Part 3 start: interaction / animation / particle systems
12	Work on project
13	Work on project
14	Project grading sessions Code submission

# Grading



- Grading done in the last week of the semester
  - Each group will have 10min to demonstrate and explain to us (the TAs) what you did and answer our questions
  - Needs to run only on your laptop (no INF3 compatibility anymore)
  - You'll have to declare who worked on what
- Midterm evaluation session in week 10
  - 5min per group
  - Mandatory, but doesn't count for final grade
  - You'll get feedback and know where you stand

# Grading / Minimal Requirements



- The project is open ended
- We only explain the minimal requirements for 4.0
  - defined on slides 11 and 15 for Project Part 1
- To get a higher grade, extend the project with
  - your own ideas
  - online tutorials
  - some of our suggestions (slide 22)

# Project Part 1 - Terrain

- Procedural Terrain
  1. Generate a height map for a terrain on the GPU
  2. Render the terrain to the screen

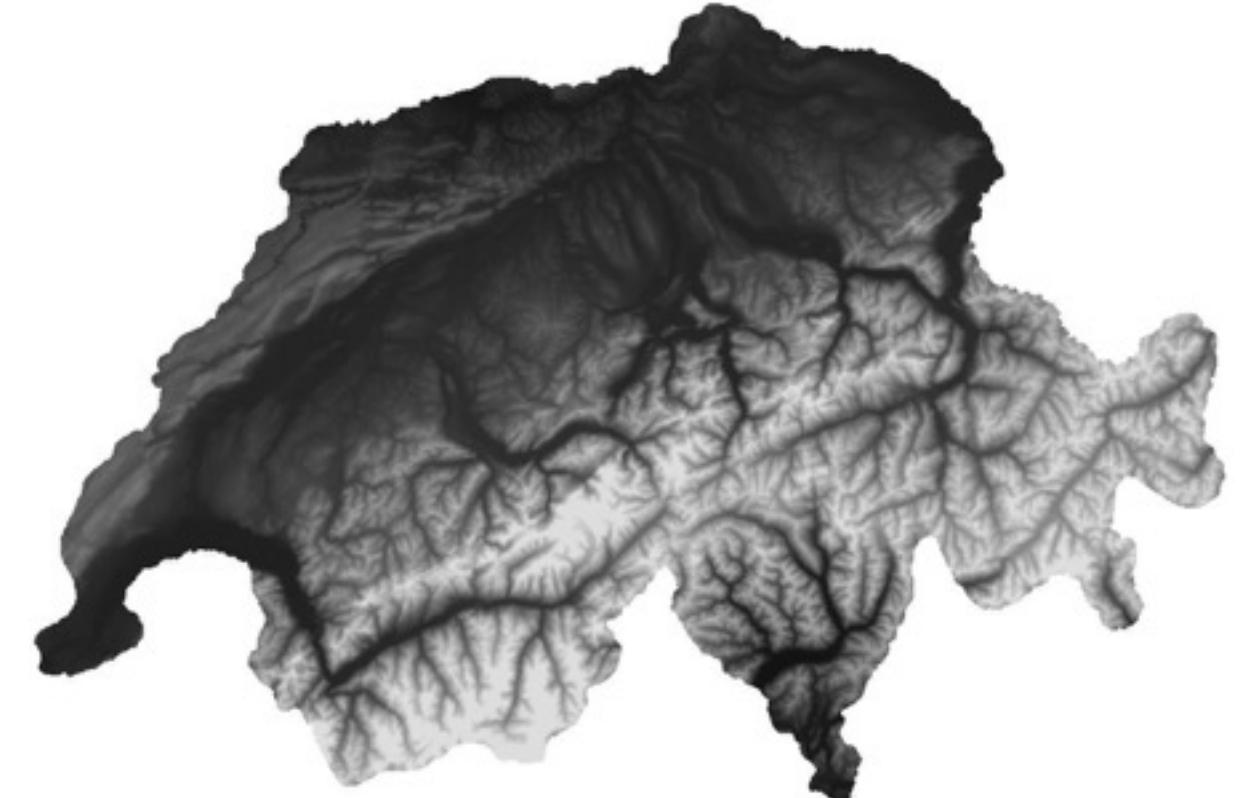


Procedural landscape by Iñigo Quilez

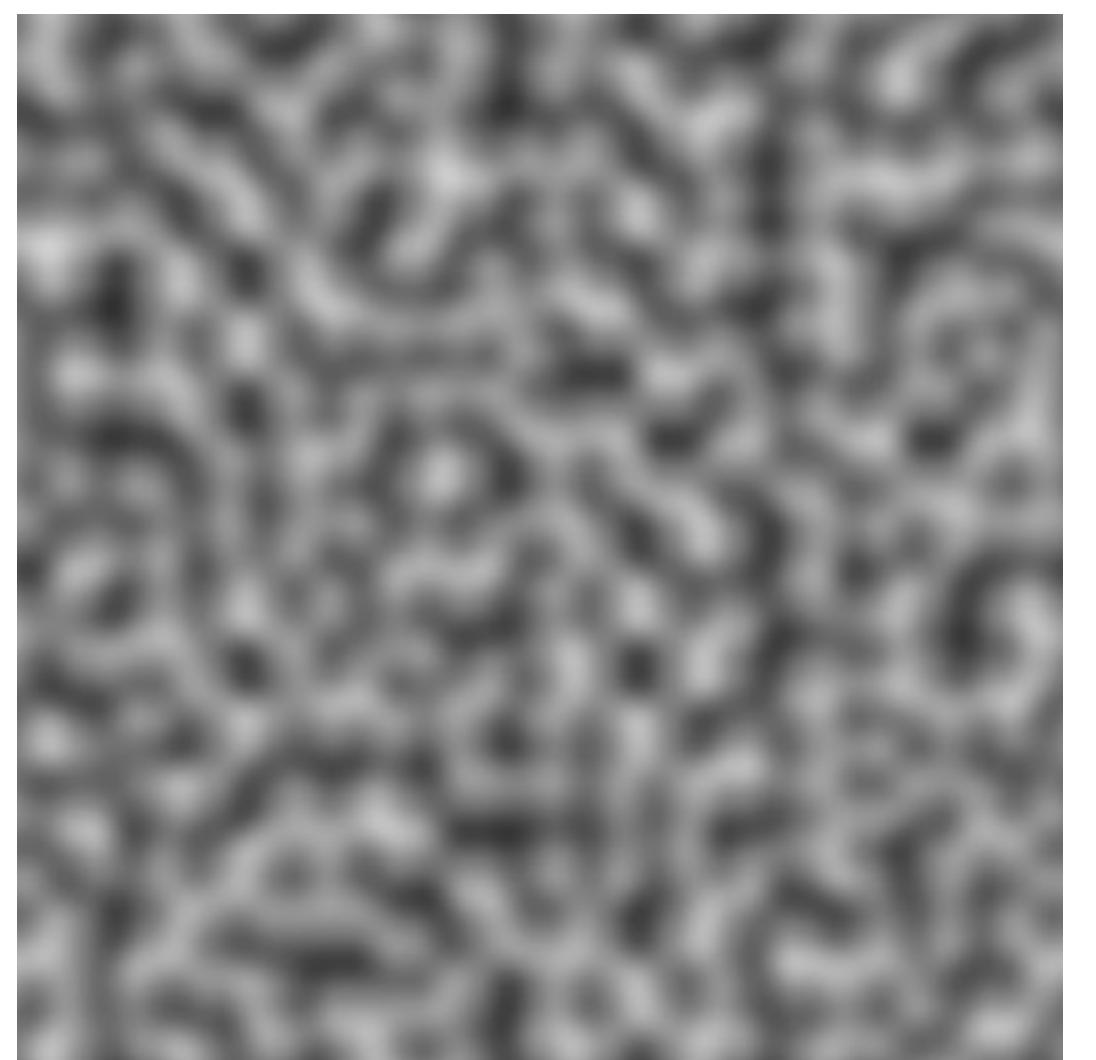
# Part 1.1 Height Map Generation



- Height map
  - Image/texture with only one ‘color’ component
  - Like a topographical map:  $(u, v)$  = position, pixel value = elevation
- Use OpenGL shader to generate height map
  - Draw fullscreen quad to a framebuffer object
  - For every pixel, the fragment shader calculates a height value (using Perlin noise or combinations of it)
- Height map generation as preprocessing step
  - More performant than recalculating the height values in every render iteration
- Reuse and adapt code from HW6



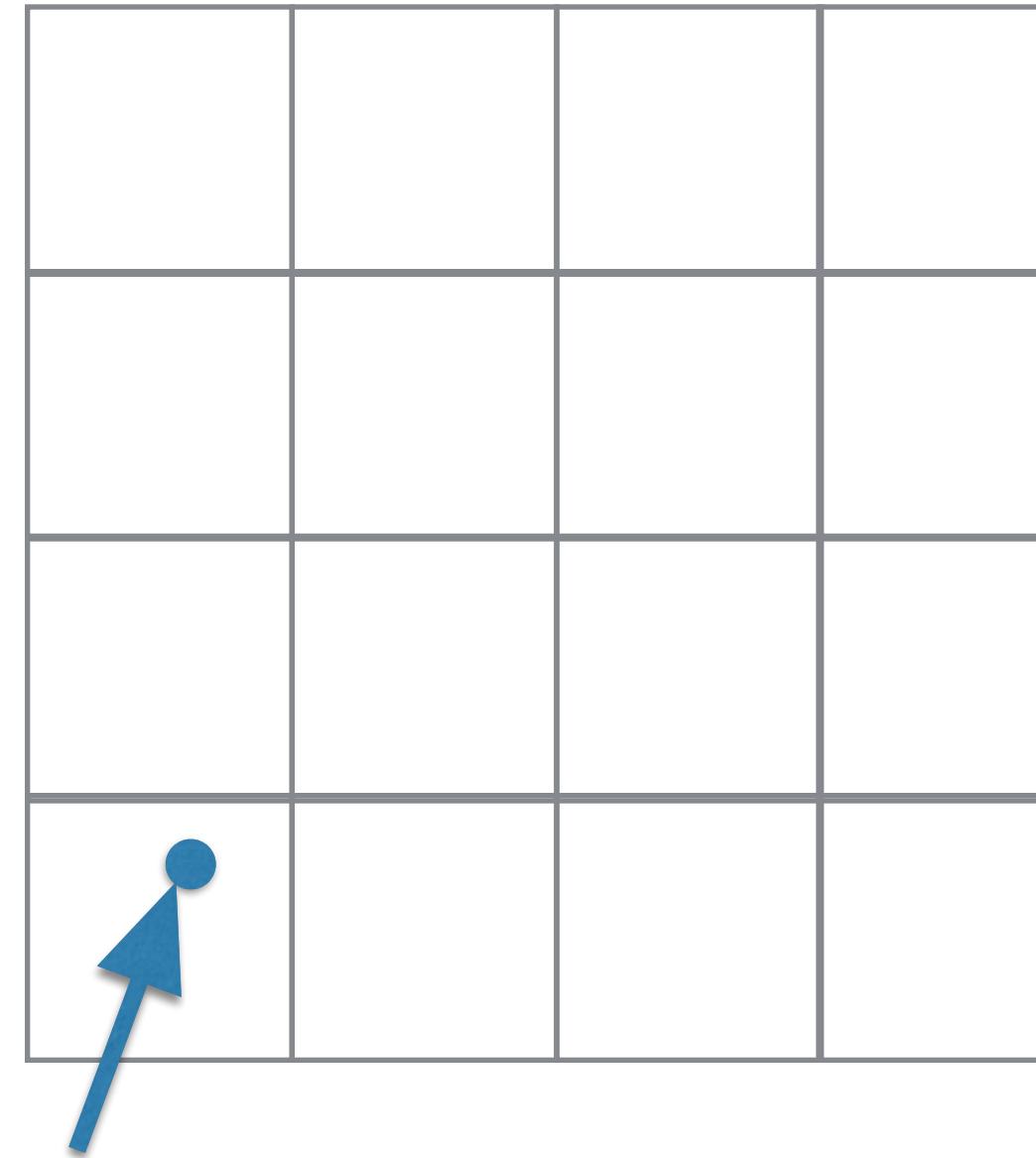
Height map of CH



Perlin noise height map

# Perlin noise in 2D

Subdivide domain into grid  
with unit cells



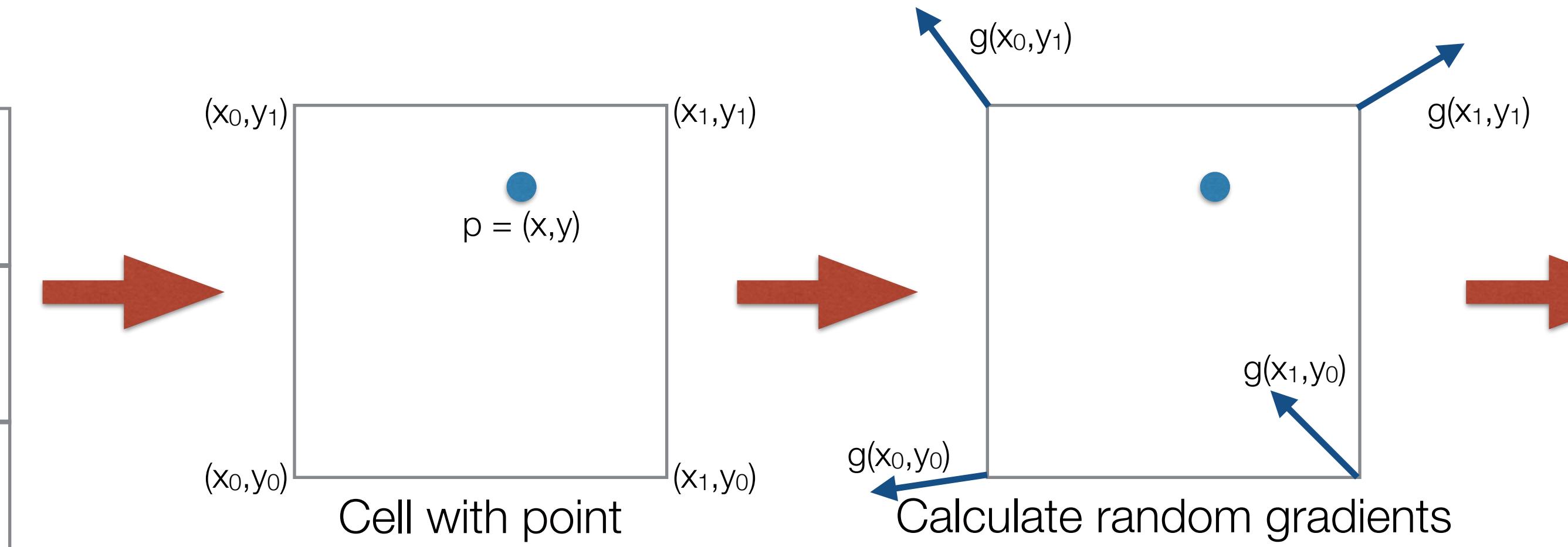
Find cell that  
your point is in

$$mix(x, y, \alpha) = (1 - \alpha) \cdot x + \alpha \cdot y$$

$$st = mix(s, t, f(x))$$

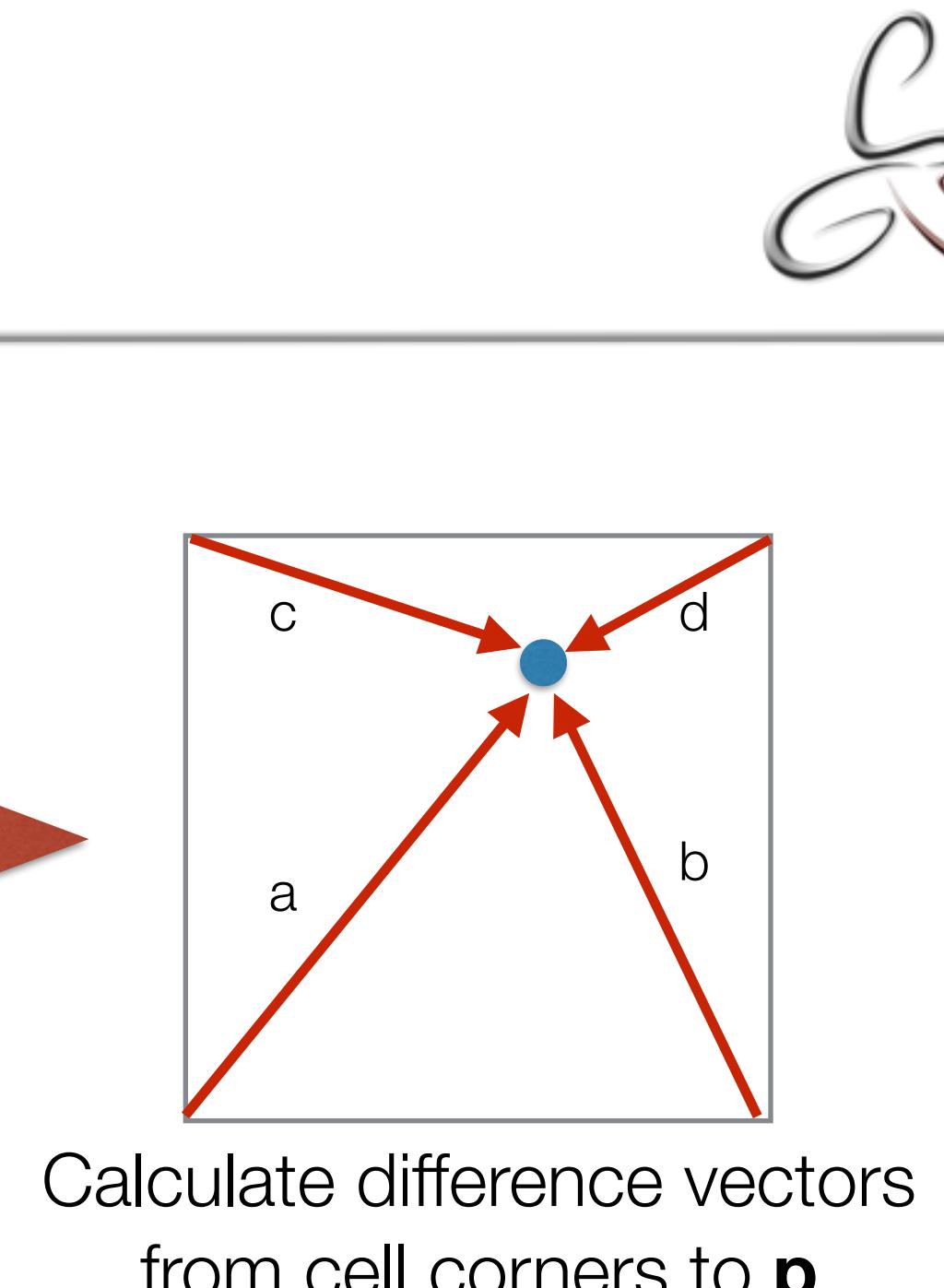
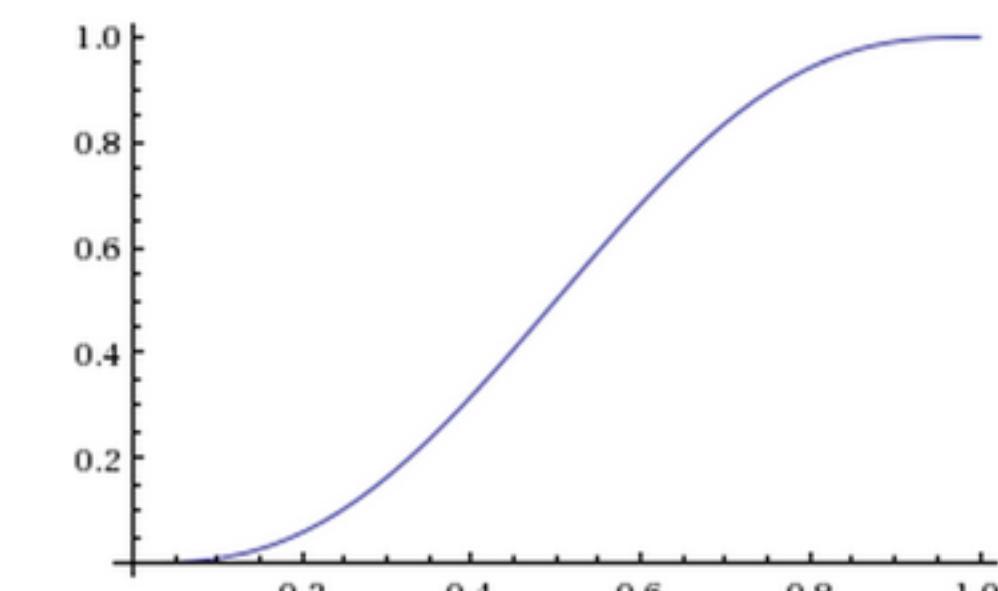
$$uv = mix(u, v, f(x))$$

$$noise = mix(st, uv, f(y))$$



Smooth interpolation function  
 $C^2$  continuity at the boundaries  

$$f(t) = 6t^6 - 15t^4 + 10t^3$$

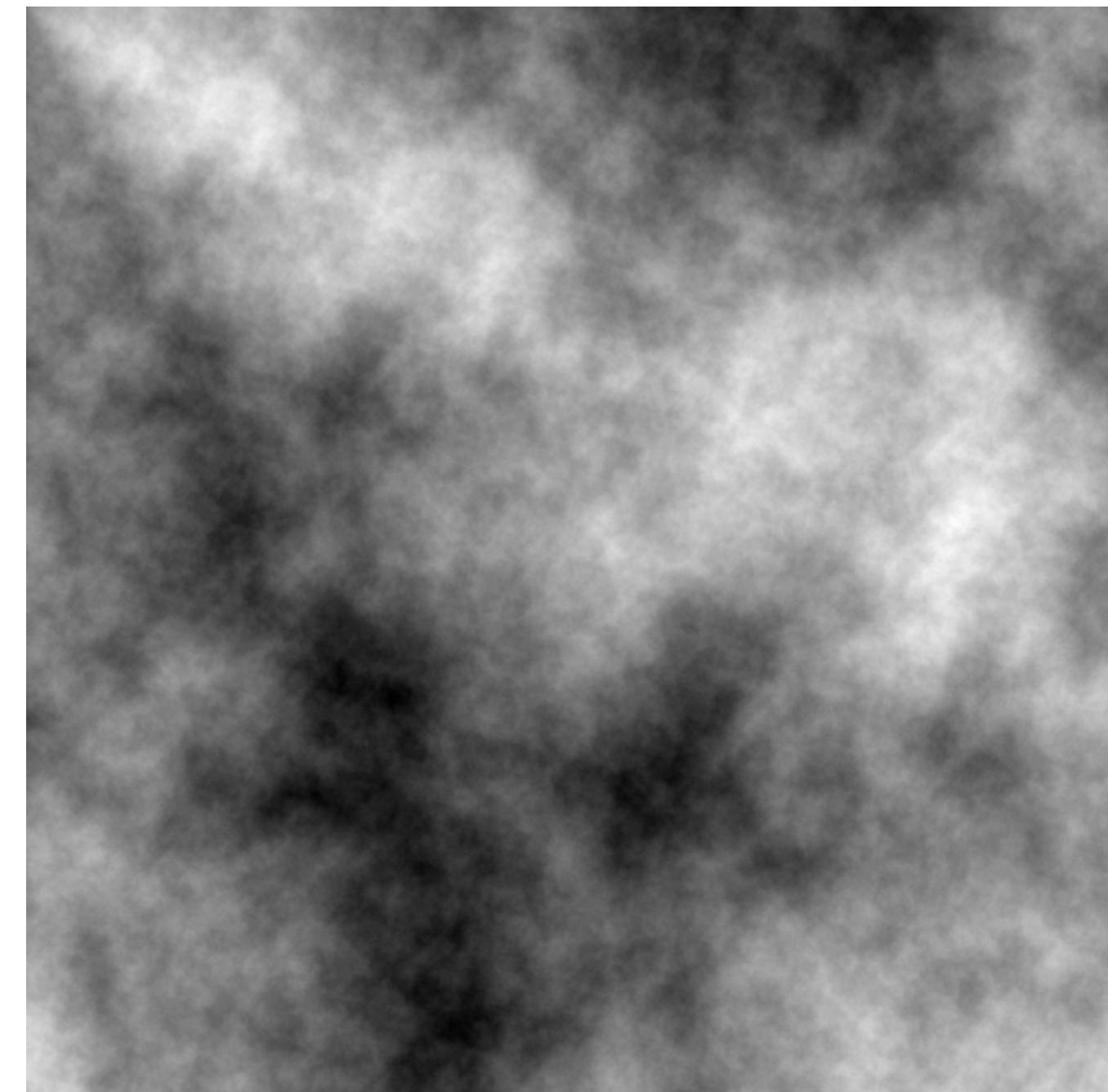


$$\begin{aligned} s &= g(x_0, y_0) \cdot a \\ t &= g(x_1, y_0) \cdot b \\ u &= g(x_0, y_1) \cdot c \\ v &= g(x_1, y_1) \cdot d \end{aligned}$$

dot products to get scalar  
values for the corners

# Part 1.1 TODO

1. Create an FBO with a texture attached to it with 1 component  
The attached texture stores the height map
2. Create and bind a shader program for rendering to that FBO
3. Create a Perlin noise function in the fragment shader  
The fragment shader writes the height values into the texture
4. Render a full screen quad  
The quad reaches from  $[-1, -1, 0]$  to  $[1, 1, 0]$ . Don't forget to resize the viewport to the texture size for this to work correctly (and to reset it after height map generation).
5. Render before `glfwMainLoop()`  
Unbind the FBO after terrain generation, otherwise you keep rendering to it instead of the screen
6. Implement *fractal Brownian motion (fBm)*



fBm height map

# Perlin Noise Resources

- Ken Perlin: **Improved Noise**, SIGGRAPH 2002
  - <http://mrl.nyu.edu/~perlin/noise/>
  - source code available, Mark showed this during the lecture
- Simon Green, **Implementing Improved Perlin Noise**, GPU Gems 2
  - [http://http.developer.nvidia.com/GPUGems2/gpugems2\\_chapter26.html](http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter26.html)
  - source code available, needs to be ported from HLSL to GLSL
- Matt Zucker, **The Perlin noise math FAQ**
  - <http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html>
  - explains maths behind Perlin Noise

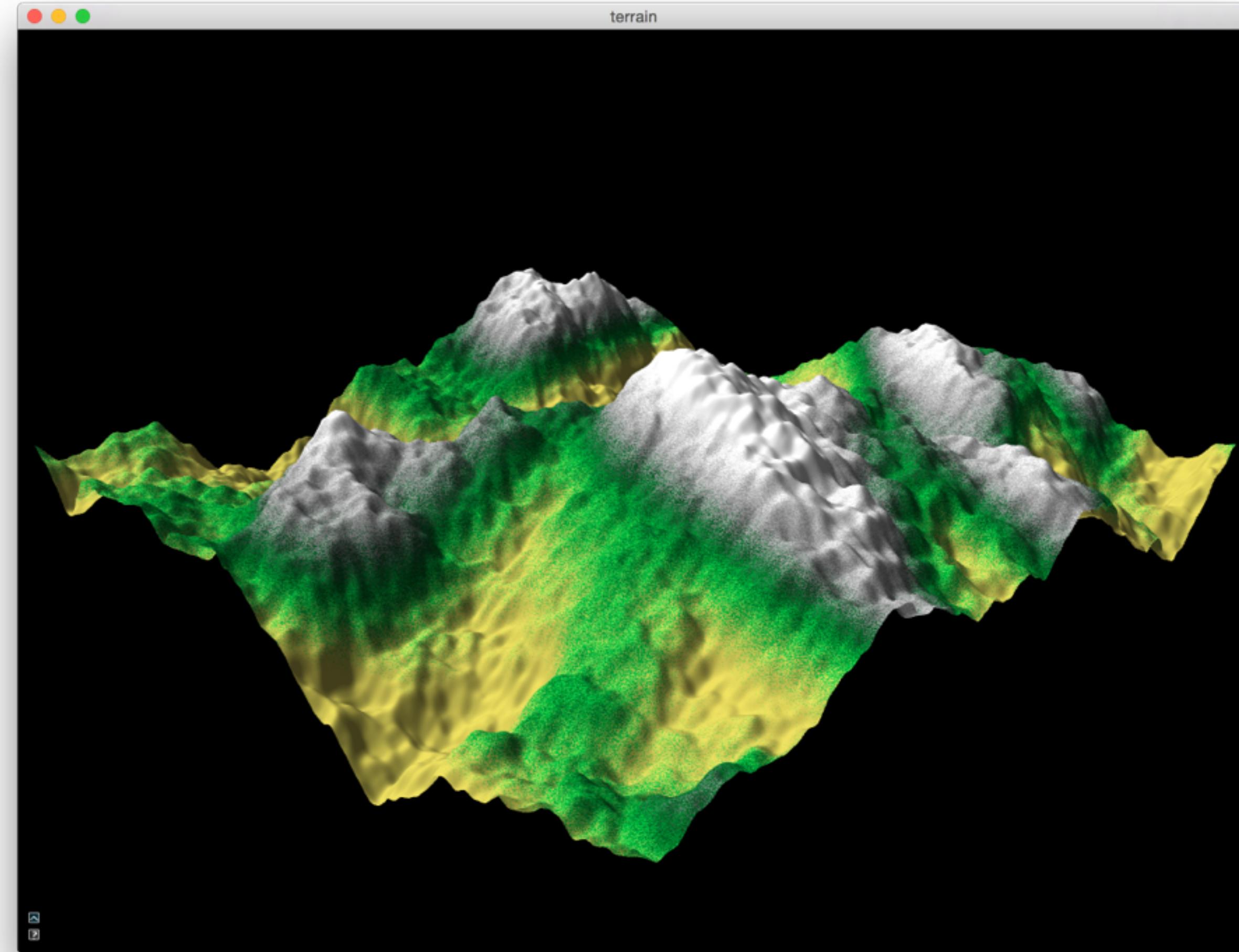
# Part 1.1 Pitfalls



- Pay attention to texture formats
  - Default OpenGL textures are clamped to [0, 1].
  - We recommend to use unbounded floating point textures for the height map, e.g., **GL\_R32F**
  - [https://www.opengl.org/wiki/Image Format](https://www.opengl.org/wiki/Image_Format)  
<https://www.opengl.org/wiki/GLAPI/glTexImage2D>
- If using the GPU Gems 2 tutorial (previous slide)
  - You will use multi-texturing to store gradient and permutation tables
  - Tutorial on texture units and multi-texturing: <https://open.gl/textures>

# Part 1.2 Terrain Rendering

- Create a terrain from the pre-calculated height map



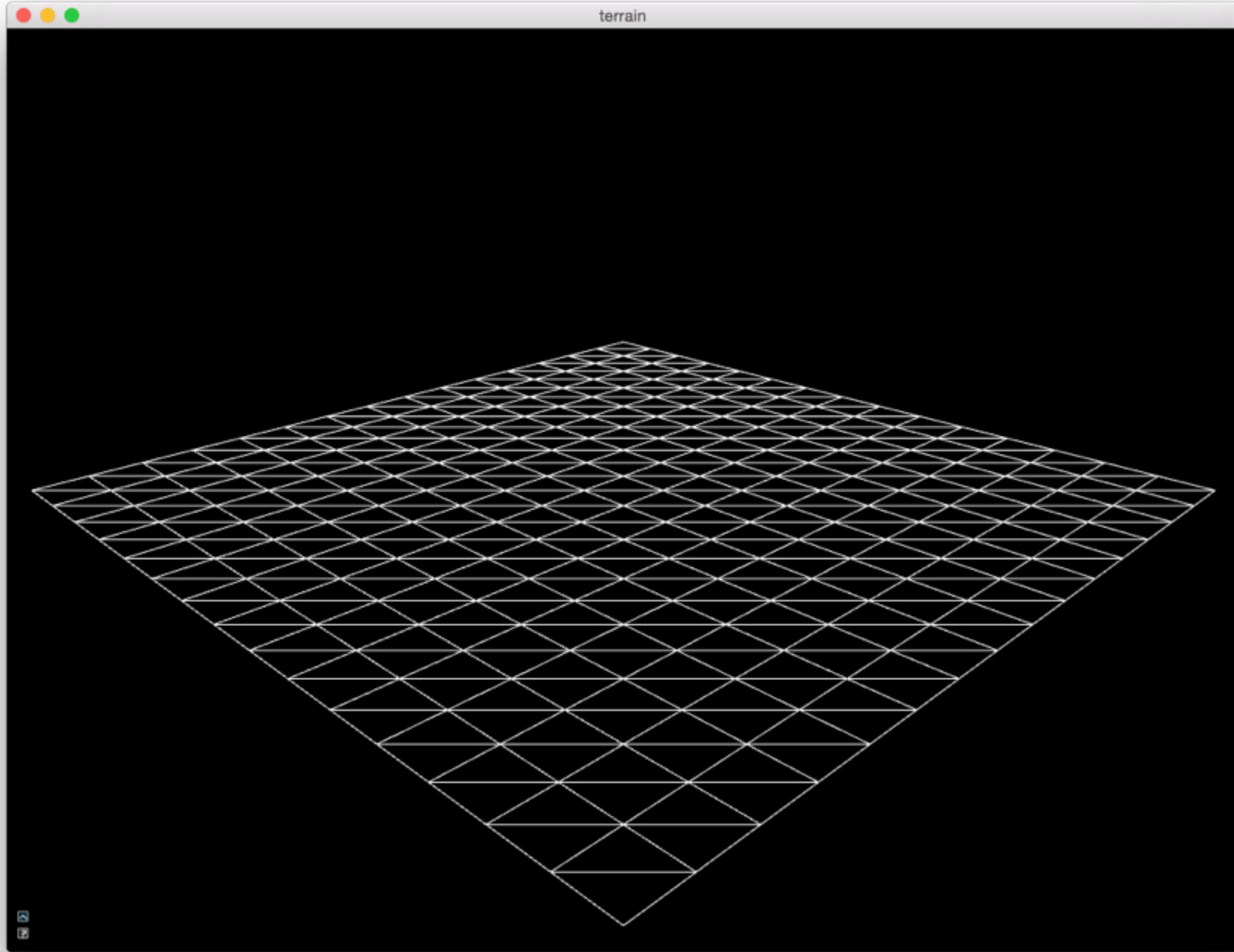
colored fBm terrain

# Part 1.2 TODO

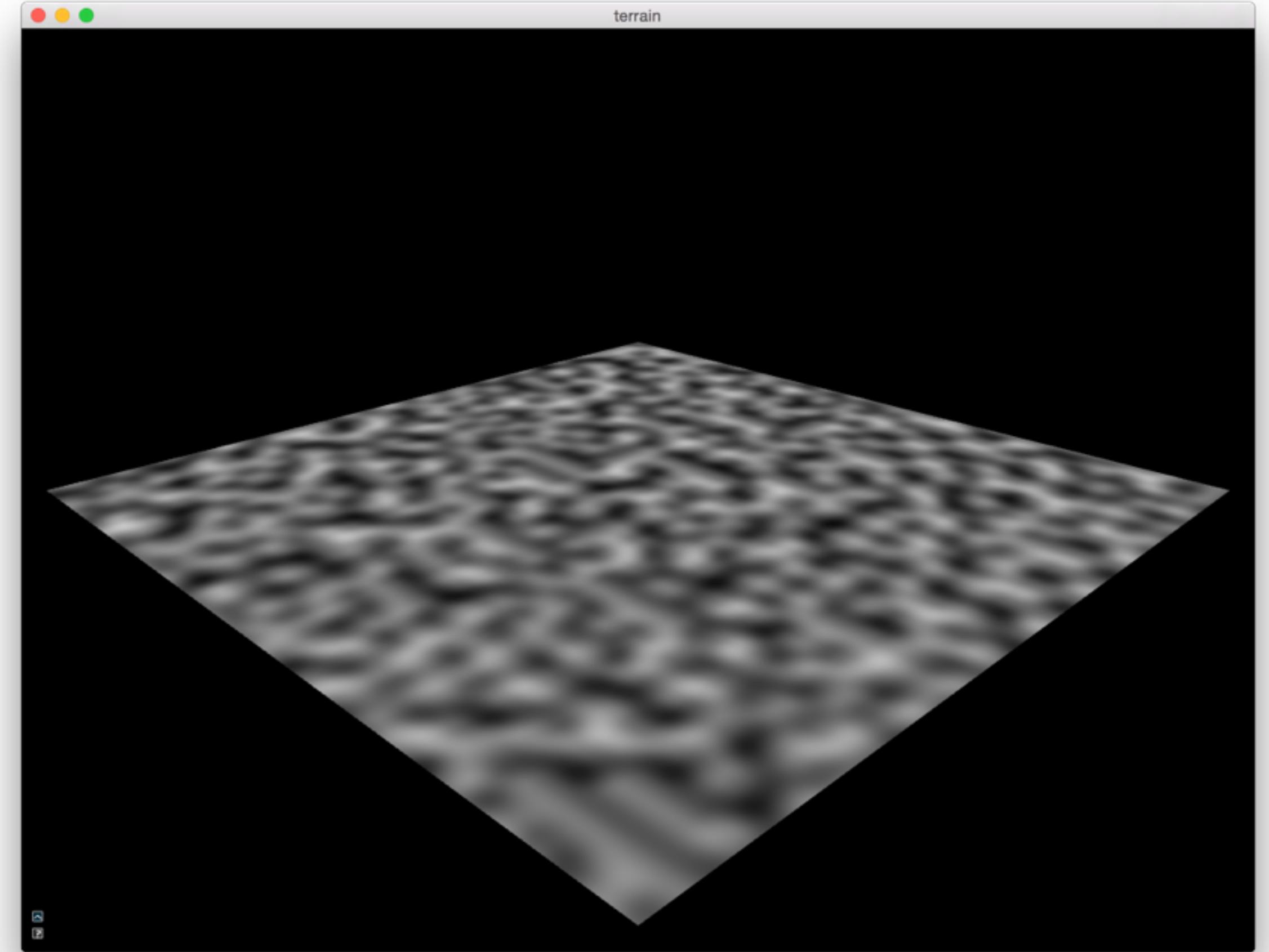


1. Create triangle grid to represent the terrain (at least 512x512).  
(reuse from HW3)
2. Create a shader program for terrain rendering.  
The vertex shader samples the height map texture and displaces the vertices according to the height. (reuse/adapt from HW3)
3. Diffuse shading of the terrain in the fragment shader.  
You have calculated the normal at every position in the fragment shader.  
One way to do this is by using finite differences: you can find the gradient at the current pixel position by comparing the elevation with the elevation of the neighboring pixels in the height map.
4. Use simple schema to color the terrain according to height.

# Part 1.2 Terrain Rendering

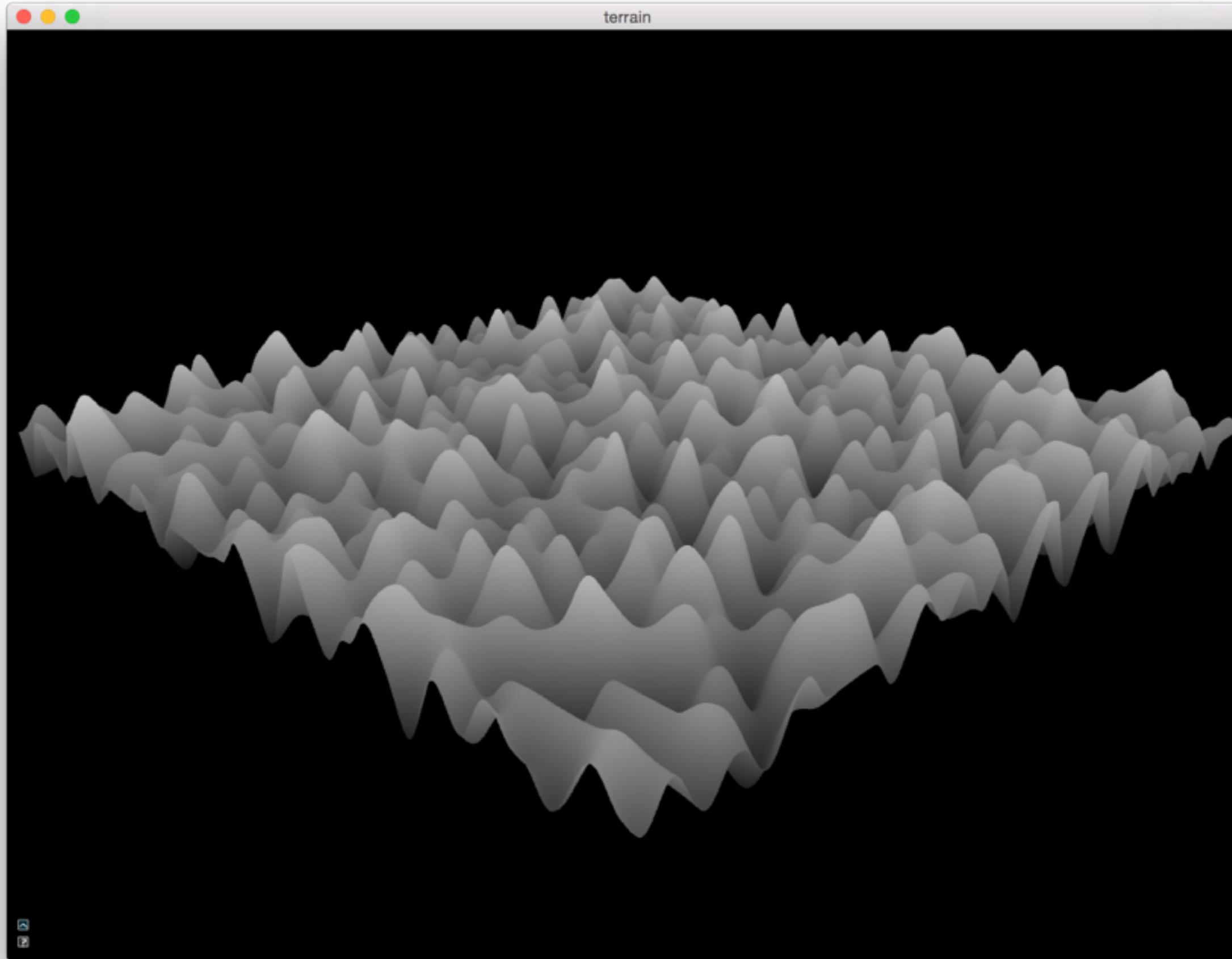


Step 1: triangle grid  
(only 16x16 for visualisation)

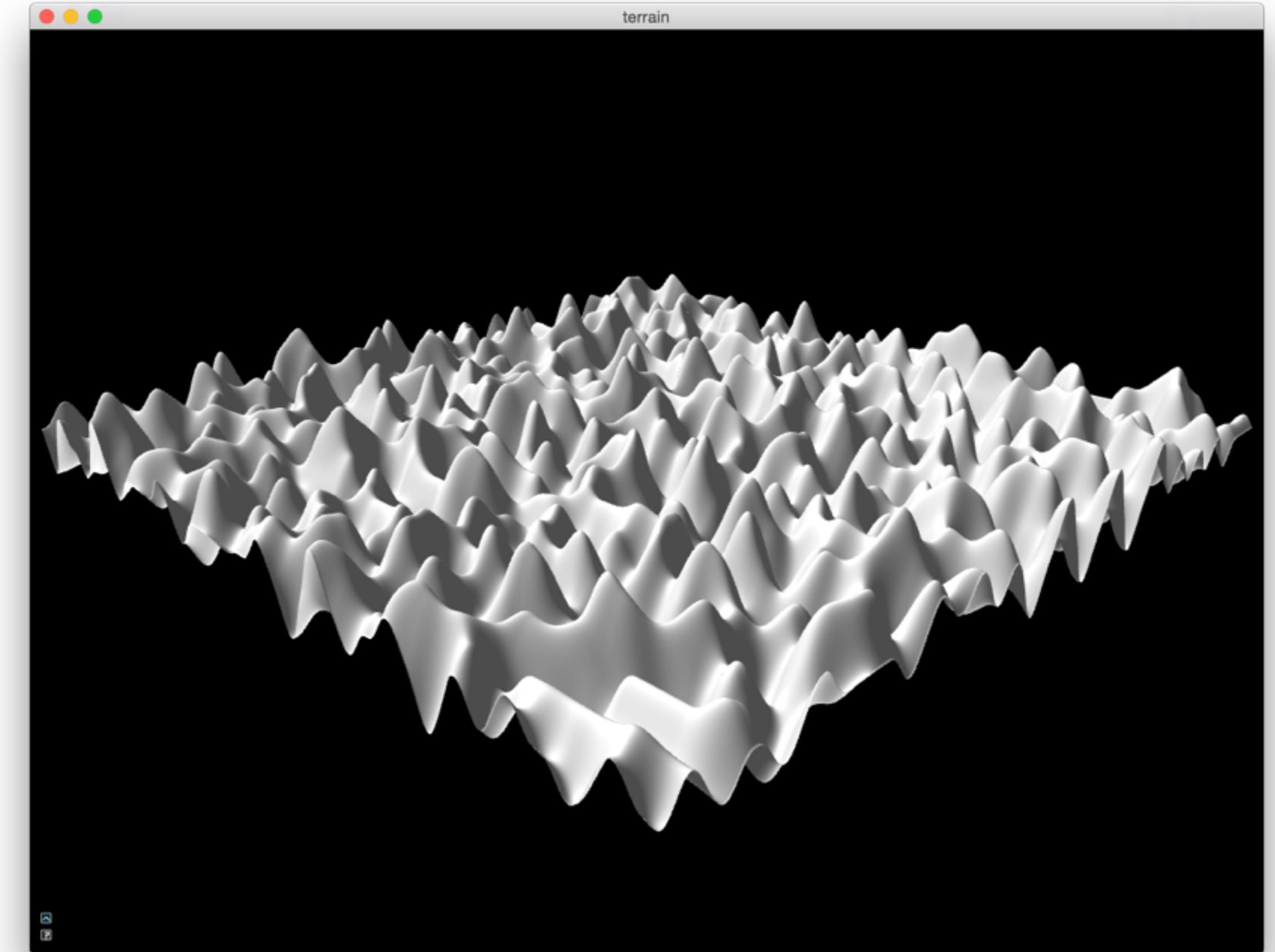


Grid textured with Perlin noise height map

# Part 1.2 Terrain Rendering

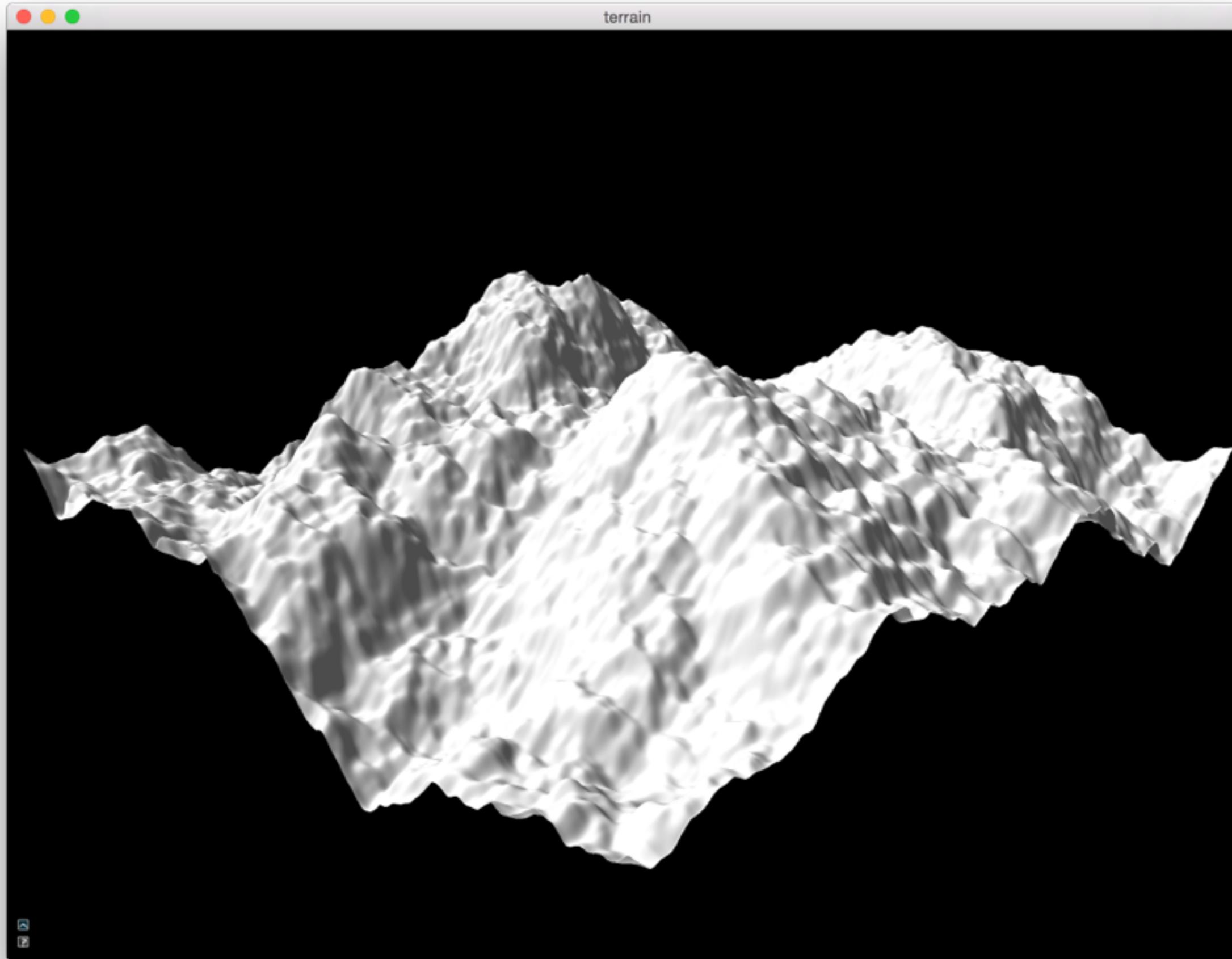


Step 2: Displacing vertices  
(on 512x512 grid)

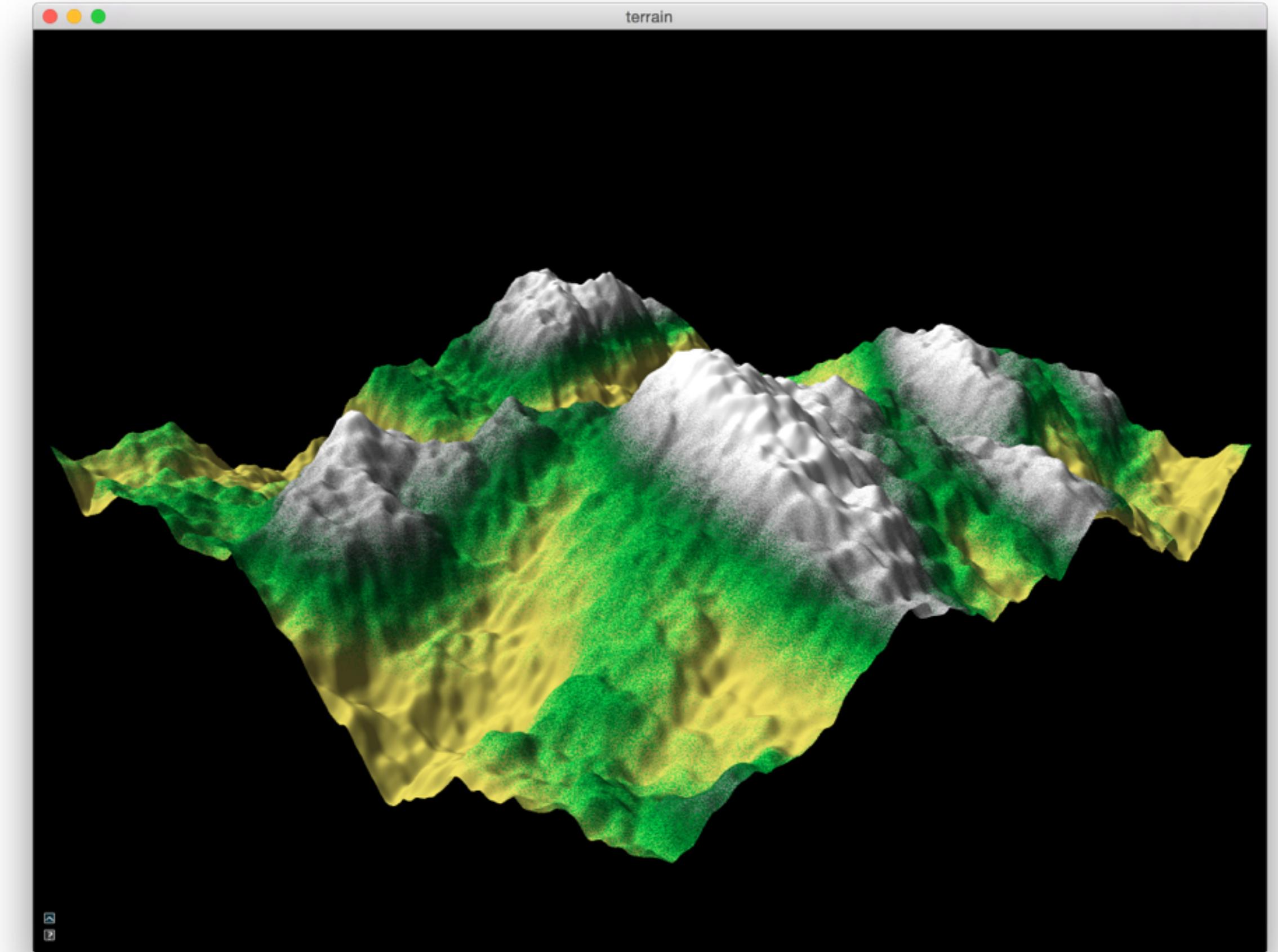


Step 3: Diffuse Shading

# Part 1.2 Terrain Rendering



Fractal Brownian Motion (fBM) Terrain  
with diffuse shading



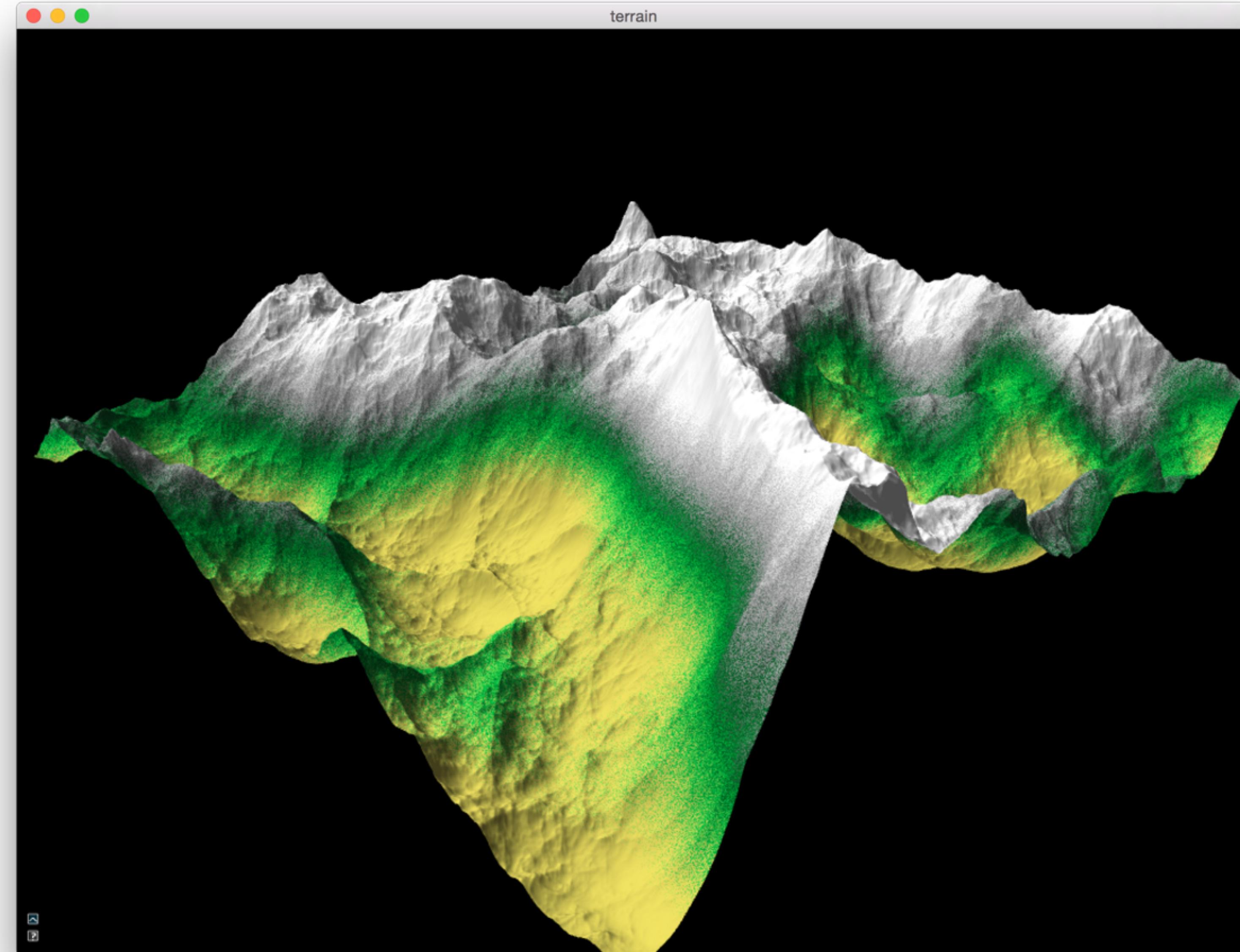
Step 4: Coloring schema depending on height

# Project Part 1 Hints



- Impossible to visualize height map (Part 1.1) without being able to display it (Part 1.2)
  - In practice, it might make more sense to start with the Part 1.2
  - Figures on slides 15-17 show a possible implementation order
- Fractal terrains are extremely parameter sensitive
  - Correct implementation with bad parameters can look wrong
  - Map parameters to keys/AntTweakBar to interactively change them

# Ridged Multifractal / Demo





- Ken Musgrave, **Procedural Fractal Terrains**
  - <http://www.cs.rit.edu/~jmg/courses/cgII/20033/slides/2-1-ProceduralModels.pdf>
  - Good explanation of types of fractal terrains and their parameters
- Textbook: **Texturing & Modeling: A Procedural Approach**
  - more details than Ken's paper, but very chaotic
  - available online at <http://library.epfl.ch>

# Extensions



- Some ideas to improve the project (and your grade)
  - **easy** – other combinations of noise functions for more interesting terrains, e.g., hybrid or ridged multifractals
  - **easy** – use 3D noise to create materials for meshes, e.g., wood or marble
  - **medium** – tessellation shaders for level of detail (LOD) terrain rendering  
Dynamically refine/coarsen the grid depending on the camera position  
This needs a GPU that supports OpenGL 4
  - **medium** – instead of using a flat triangle grid, use 3D noise and map it to a sphere in order to render a procedural planet
  - **hard** – make an infinite terrain that you can navigate through.
  - **very hard** – L-systems to populate your terrain with trees
  - Your own idea.

# OpenGL Help



- If you feel uncomfortable with OpenGL or want more infos than given in previous practicals
- These tutorials cover everything we learned so far (and more)
  - <https://open.gl/>
  - <http://www.learnopengl.com/>
  - <http://www.opengl-tutorial.org/>
- OpenGL documentation
  - <http://docs.gl/> (for us only gl3 or gl4 functions)

# OpenGL Debugging



- Debugging OpenGL code takes long and is painful. Start early!
- Helpful tools
  - gDEBugger (Win and Linux) <http://www.gremedy.com/>
  - OpenGL Profiler (OSX, included in Xcode)
  - `check_error_gl()` in our framework