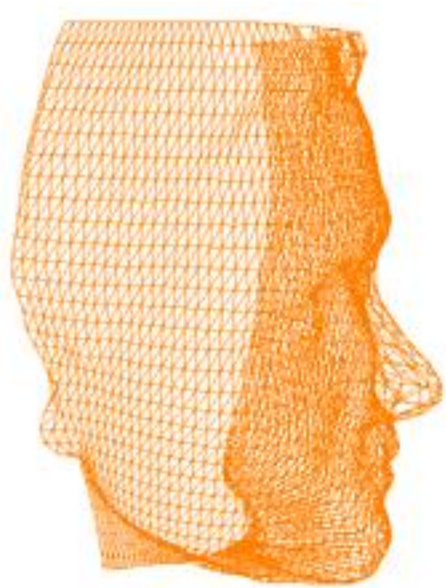
# COSC422 Advanced Computer Graphics



# 2 Terrain Rendering

Semester 2
2021

**R. Mukundan** (mukundan@canterbury.ac.nz)
Department of Computer Science and Software Engineering
University of Canterbury, New Zealand.

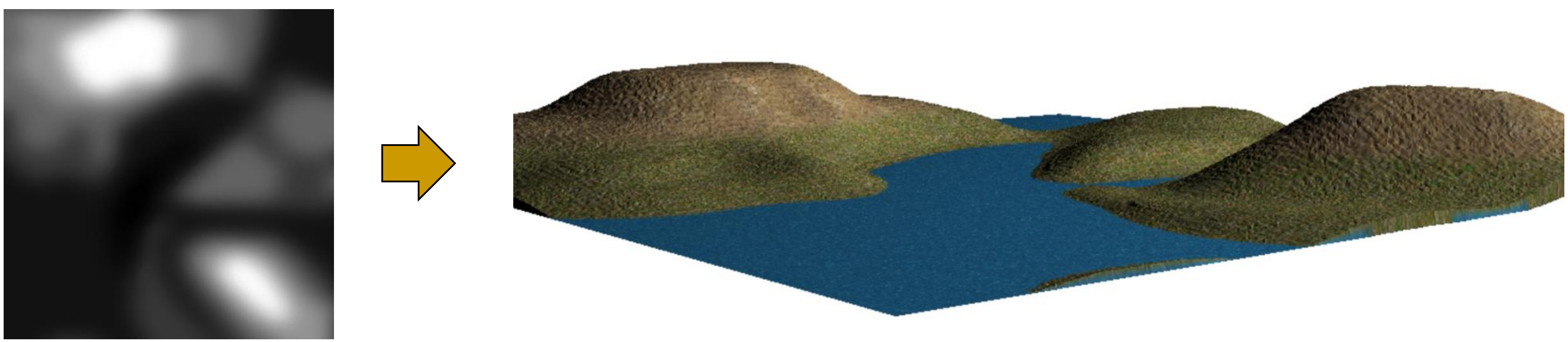
# Lecture Outline

- Height Maps
  - Image sources
  - Formats
  - Procedural generation
- Terrain Modelling
  - Terrain Level of Detail
  - Terrain Texturing
  - Programming Considerations

# Height Maps

- Usually a gray-level image with 256 [0-255] intensity values (8BPP)
  - Intensity 0 corresponds to minimum height (usually, 0)
  - Intensity 255 corresponds to maximum height
  - Use more bits per pixel (e.g. 16 bits TGA) or colour channels for a bigger range of height values

Terrain_hm_02.tga
1024x1024 24BPP

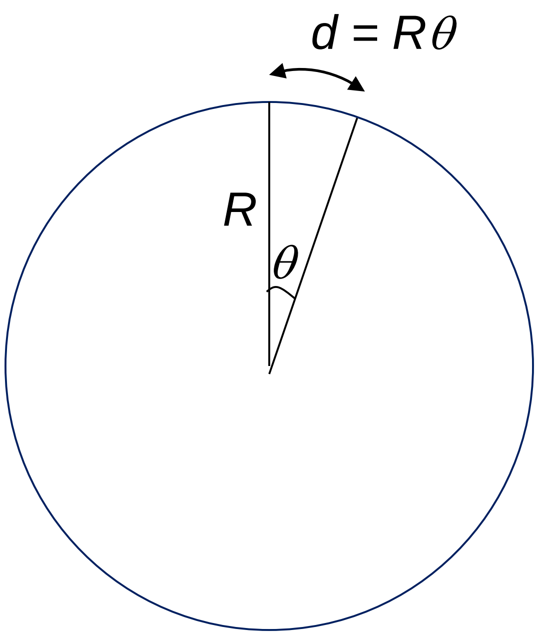


# Tangram Heightmapper

- https://tangrams.github.io/heightmapper/

# Digital Elevation Models (DEM)

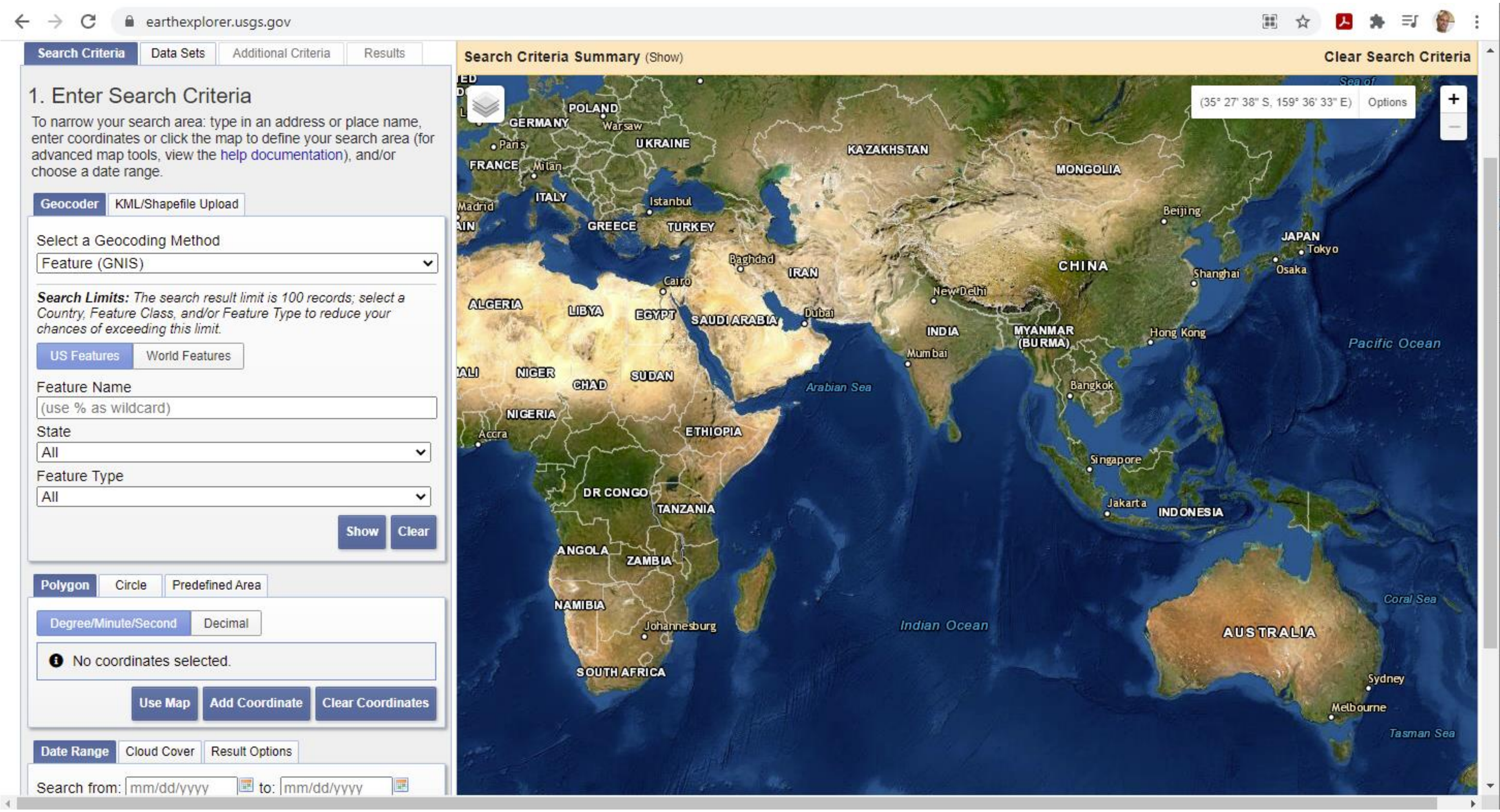
- U.S Geological Survey ([https://www.usgs.gov/](https://www.usgs.gov/))
- Resolution: 1 Arc Sec (1/3600 degree) ≈ 30m
- 500 x 500 pixels correspond to approx. 15km x 15 km




$$\begin{aligned} d &= (6300)(1/3600)(\pi/180) \\ &= 0.0305 \text{ kms} \\ &\approx 30\text{m} \end{aligned}$$

# USGS Earth Explorer

- https://earthexplorer.usgs.gov/



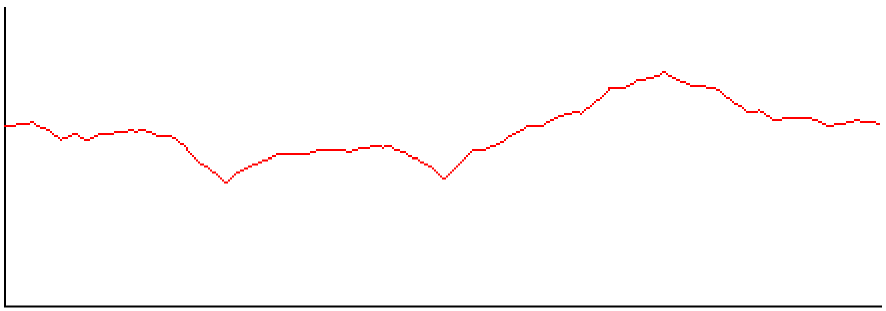
# NASA Earth Data

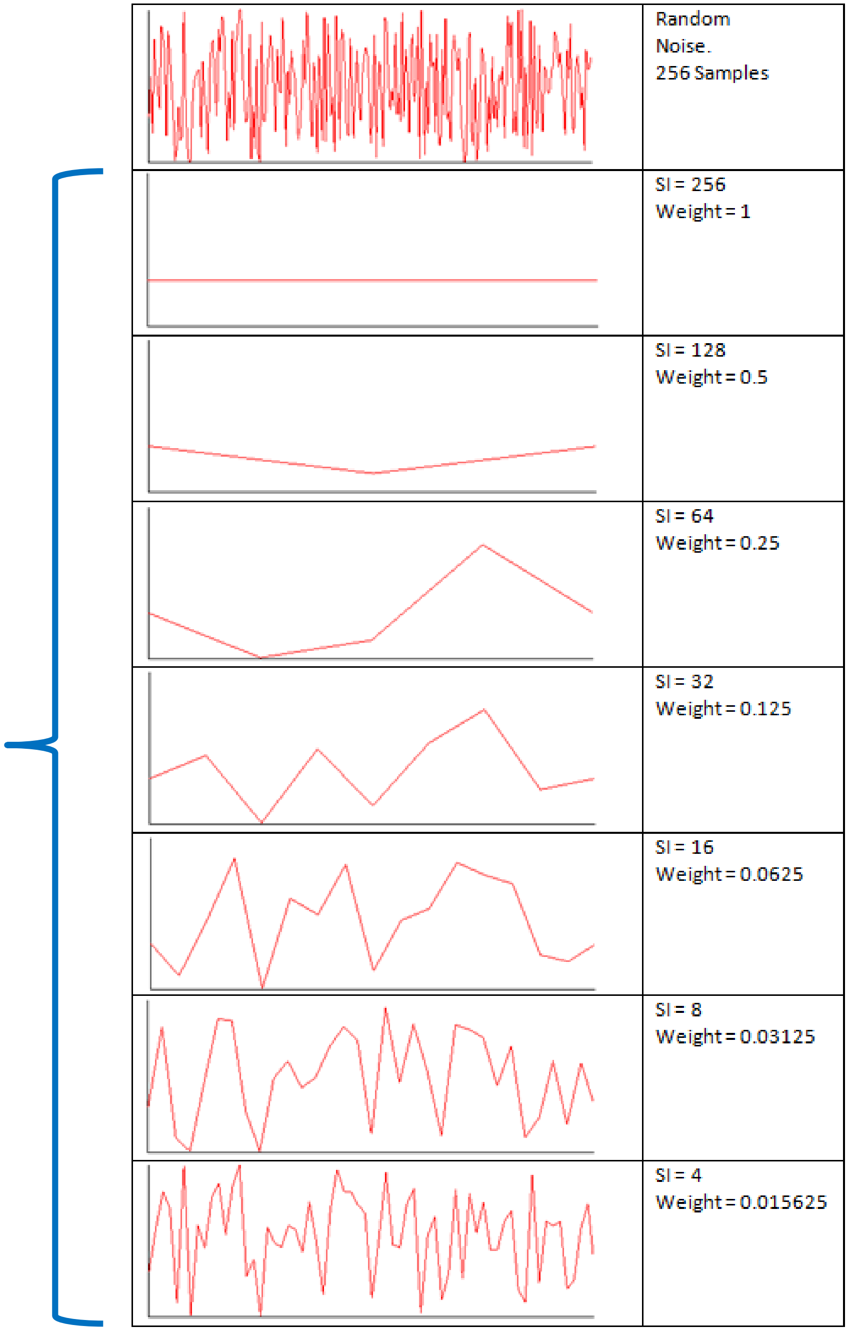
- https://search.earthdata.nasa.gov/

# Procedural Height Maps

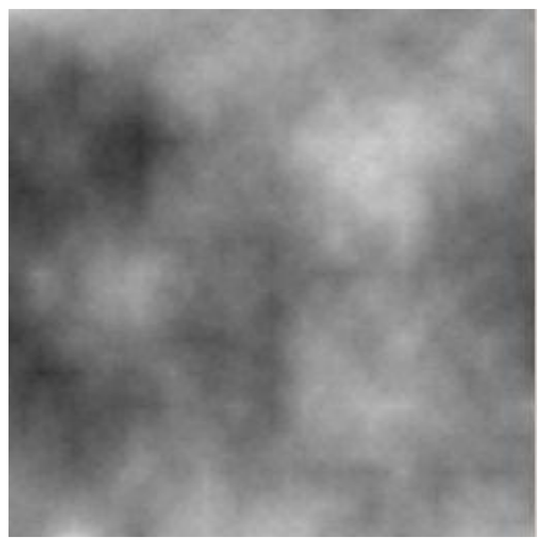
- **Perlin Noise**
  - An important noise used for rendering terrains and textures in the field of computer graphics and game design.
  - Samples a noise signal at various frequencies, and combines the sampled noise signals with a higher weight for low frequency components
  - Creating smooth random perturbations
  - Used for generating heightmaps and cloud images
  - Also known as organic noise as it is useful for generating natural looking things with some randomness.

# Perlin Noise 1D





# Perlin Noise 2D



Terrain

Cloud

Moss

# Terrain Construction

Step 1: Construct a set of patch vertices on the terrain base. (Application)

Step 2: Specify the required levels. (CS)

Step 3: Reposition vertices (*u*, *v*) of tessellated domain using patch coordinates. (CS)

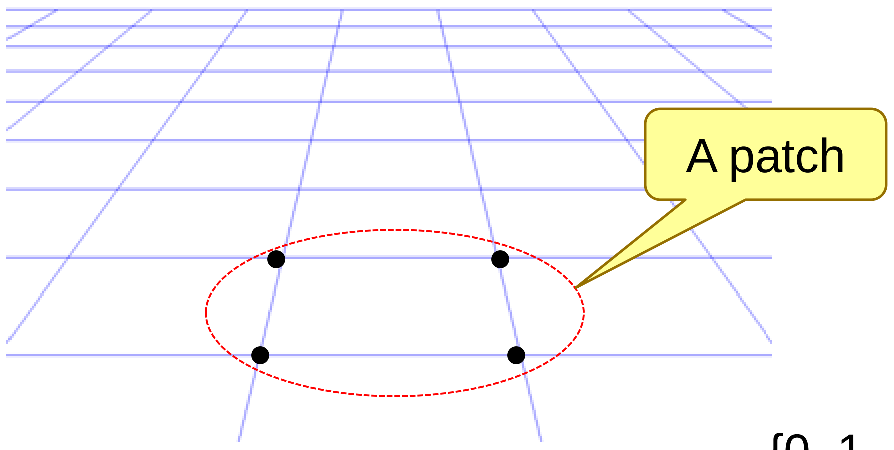
Step 4: Use a height map and a user specified water level to modify vertices of the tessellated mesh. Assign texture coordinates. (ES)

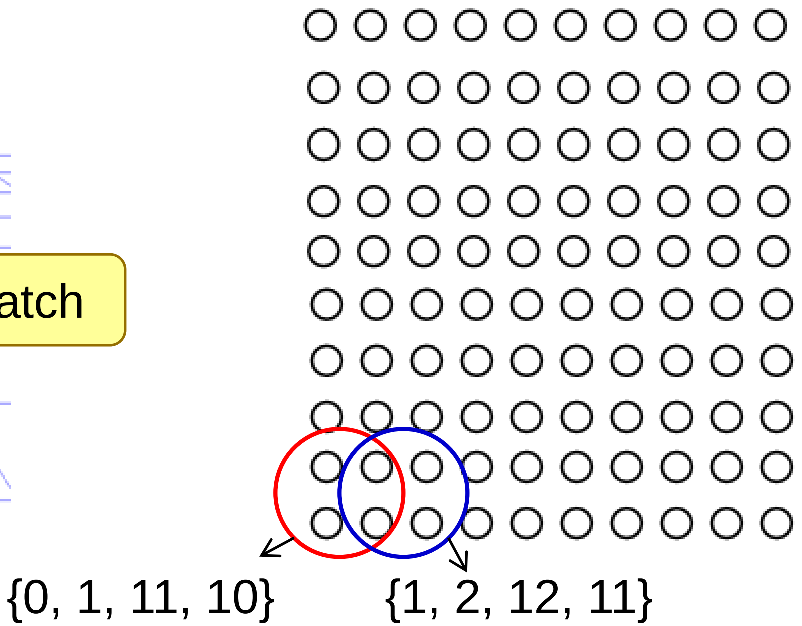
Step 5: Perform lighting computation and compute blending weights for textures. (GS)

Step 6: Perform Texture mapping. (FS)

CS: Control Shader, ES: Evaluation Shader, GS: Geometry Shader, FS: Fragment Shader

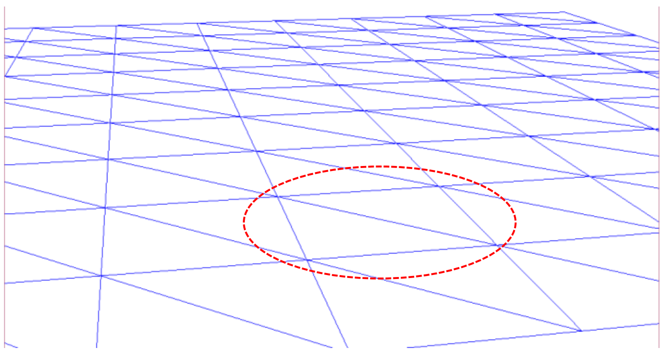
# Step 1: Terrain Base



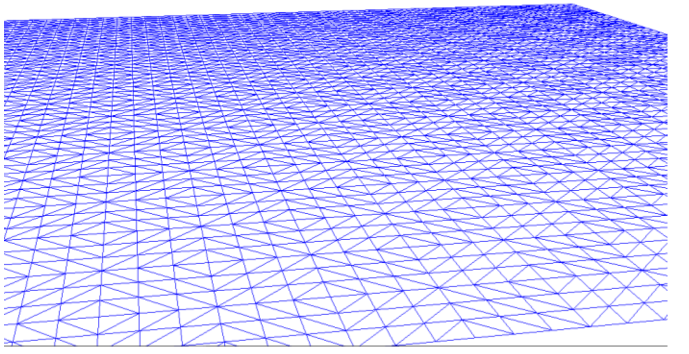




- The terrain base is represented by rectangular array of points on a large quad.
- Each patch has 4 vertices.
- Two VBOs: Vertex coordinates, Element array.

# Step 2: Tessellation Levels (CS)



Tessellation levels: {1, 1, 1, 1; 1, 1}



Tessellation levels: {6, 6, 6, 6; 6, 6}

- A simple pass-thru vertex shader !

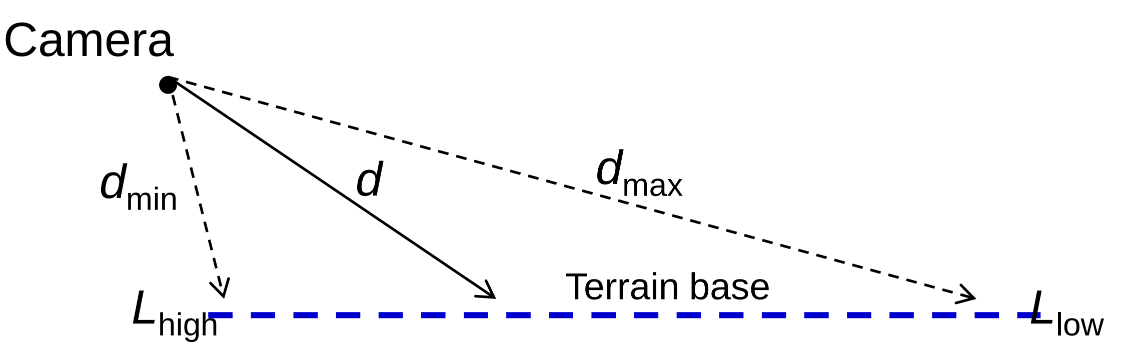
```
layout (location = 0) in vec4 position;
void main()
{
    gl_Position = position;
}
```

- The tessellation control shader receives 4 patch vertices and sets the outer and inner tessellation levels.

# Step 2: Tessellation LOD (CS)

- A design of a terrain model should incorporate a dynamic level of detail algorithm which provides a higher tessellation for patches near the camera and a lower tessellation for patches located farther away.
  - Pass the camera's position to the control shader.
  - In the control shader, compute the distance *d* between the camera and the centre of the patch.
  - Specify minimum and maximum values of patch distance, and the corresponding 'high' and 'low' tessellation levels.

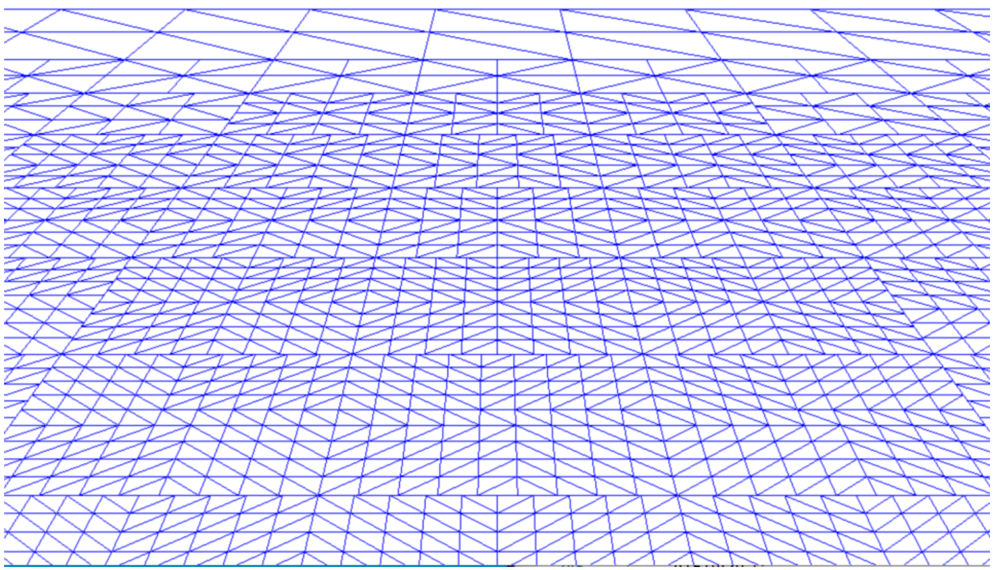
| Distance | | Tess. Level |
|---|---|---|
| $d_{min}$ | $\Rightarrow$ | $L_{high}$ |
| $d_{max}$ | $\Rightarrow$ | $L_{low}$ |



# Step 2: Tessellation LOD (CS)

- Compute the tessellation level for the current patch as follows:

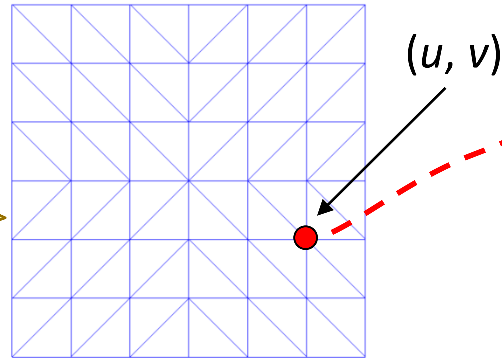
$$L = \left( \frac{d - d_{\min}}{d_{\max} - d_{\min}} \right) \left( L_{low} - L_{High} \right) + L_{High}$$



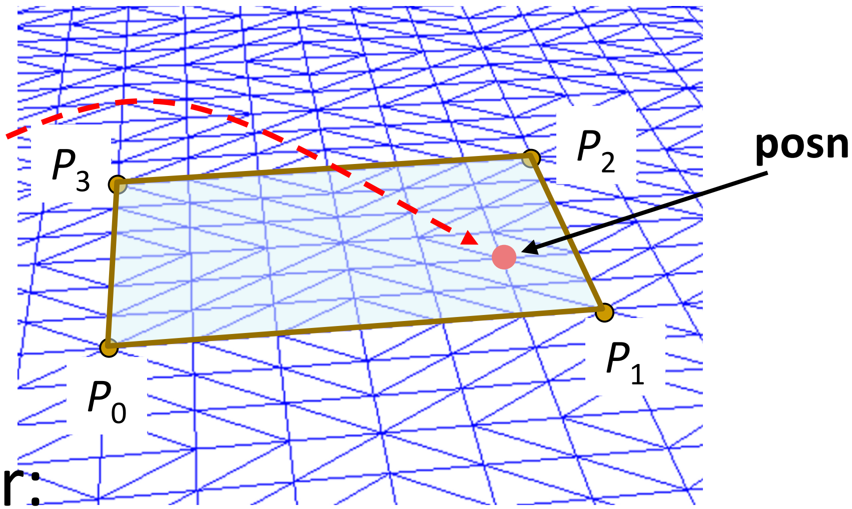
Recommended values: $L_{high} = 100$, $L_{low} = 30$

# Step 3: Tessellating the Base (ES)

Triangle mesh generated by the primitive generator




Quad Domain



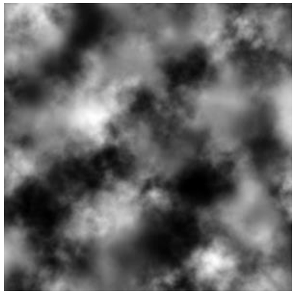

Tessellation Evaluation Shader:

- Receives 4 tessellation coordinates and 4 patch vertices.
- The current mesh vertex $(u, v)$ is repositioned using patch vertices: $u$ = gl_TessCoord.x; $v$ = gl_TessCoord.y;

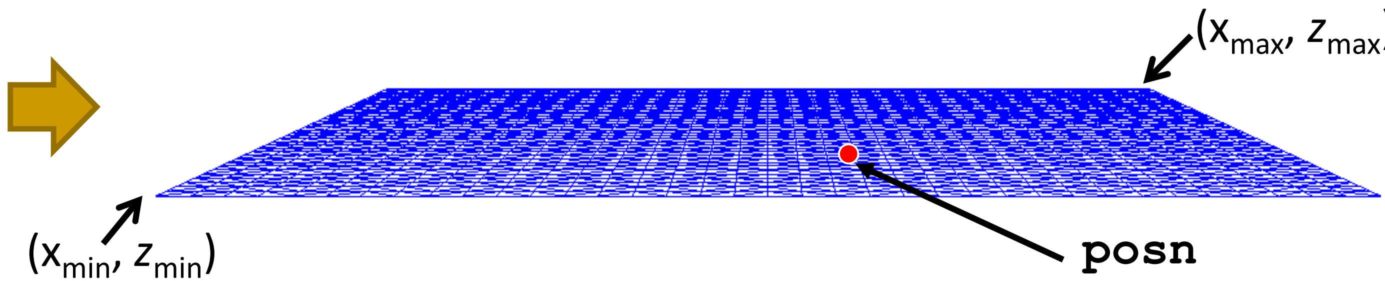
```
posn = (1-u)* (1-v)* gl_in[0].gl_Position
     +   u  * (1-v)* gl_in[1].gl_Position
     +   u  *   v  * gl_in[2].gl_Position
     +(1-u)*   v  * gl_in[3].gl_Position;
```

Patch Vertices

# Step 4: Height Map (ES)



Height Map Texture

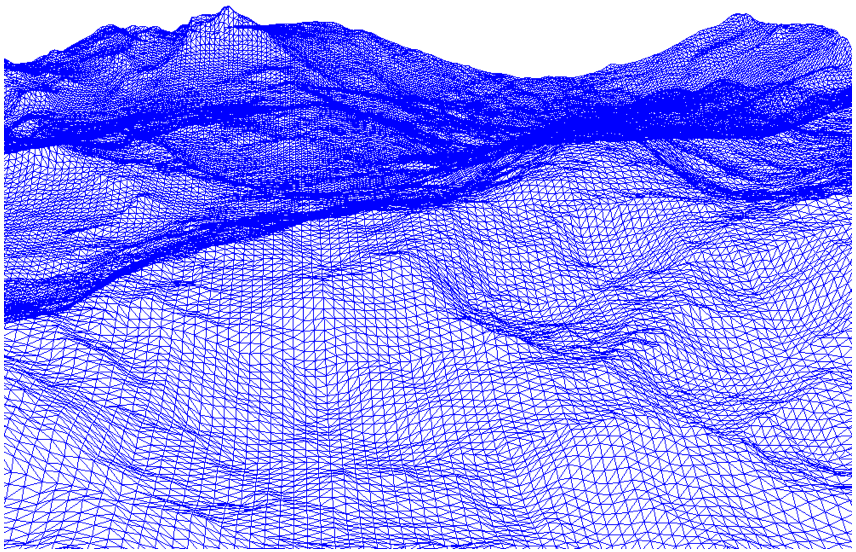



- In the evaluation shader, the height map is mapped to the whole terrain base.
- The $x_{min}$, $x_{max}$, $z_{min}$, $z_{max}$ values of the base are used to compute the texture coordinates of the current vertex at position "`posn`":

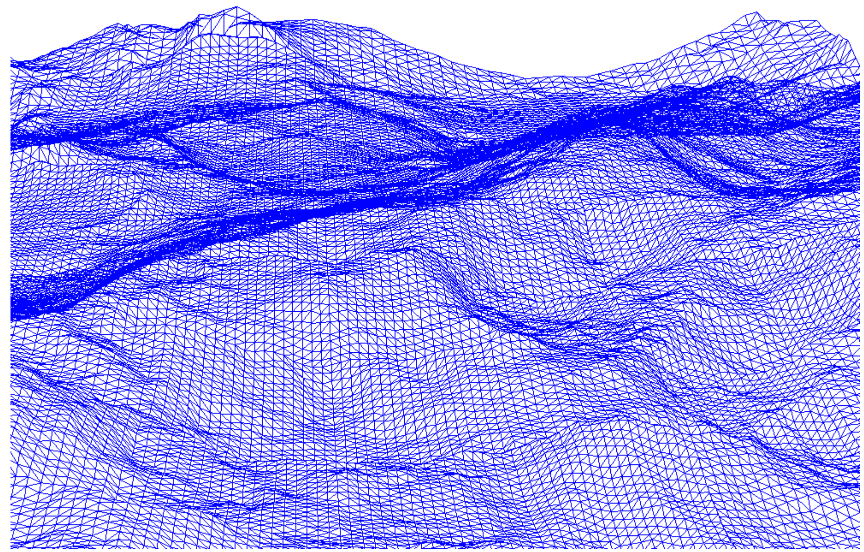
  $s$ = (posn.x – xmin)/(xmax-xmin);

  $t$ = (posn.z – zmin)/(zmax-zmin);
- The height map is sampled using texture coordinates, to get a colour value in the range [0-1] at the current vertex. This value is scaled by a factor $H_{max}$ and assigned as the y-value at the vertex (`posn.y`)

# Step 4: Height Map (ES)



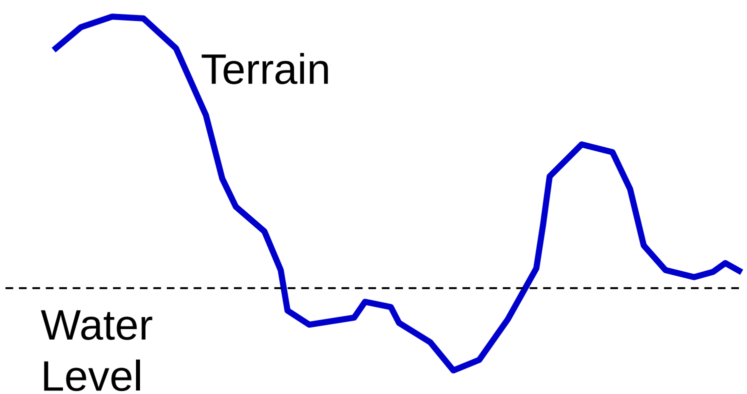
Height-mapped Terrain
Without Terrain LOD

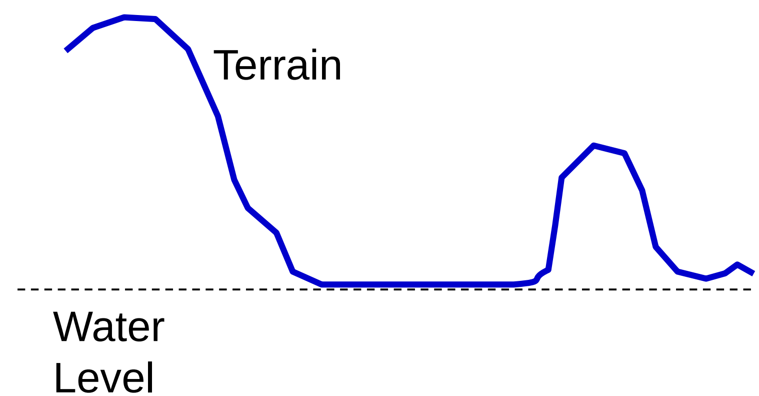


Height-mapped Terrain
With Terrain LOD

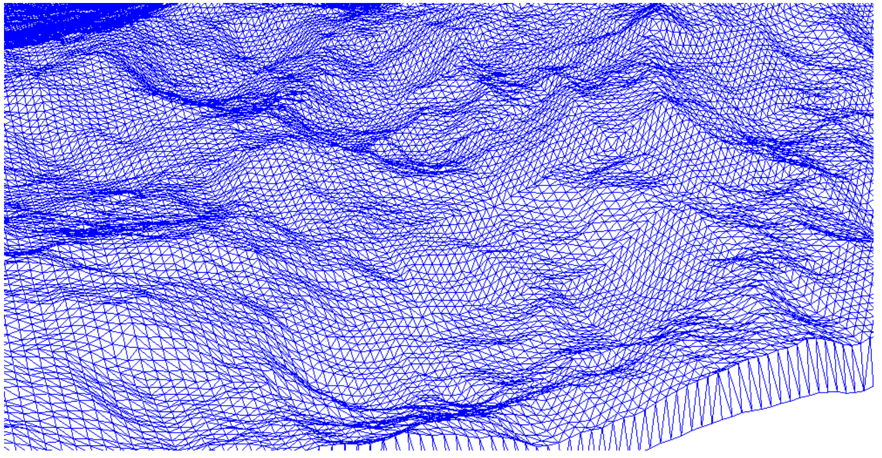
# Step 4: Water Level (ES)

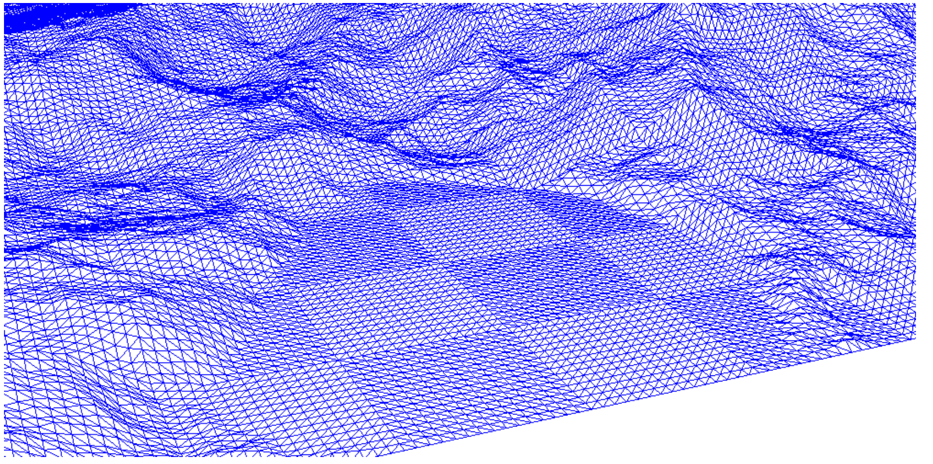
- A parameter specifying the water level in the terrain could be included in the evaluation shader.
- This value is used to model a flat region for the water surface by updating the y-value of the current vertex if it is less than the height given by the water level.

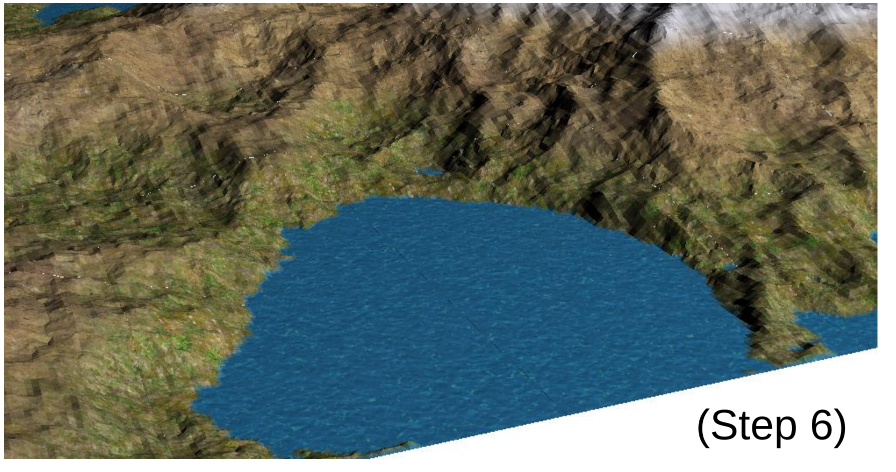


# Step 4: Water Level (ES)



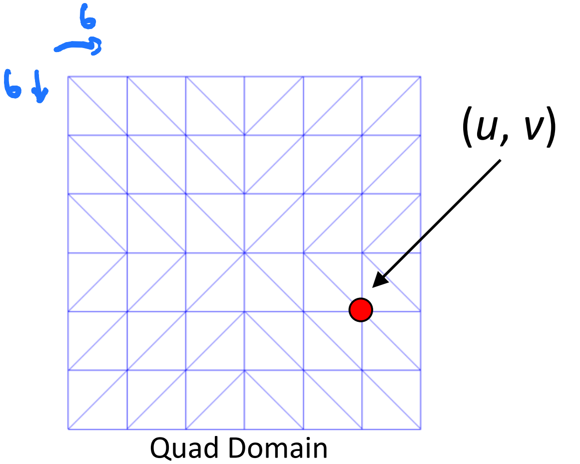




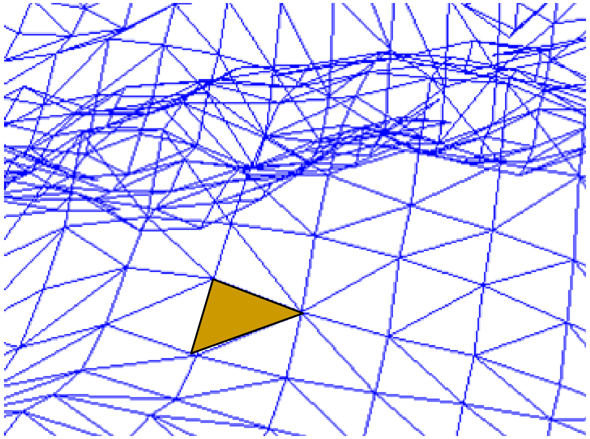
(Step 6)

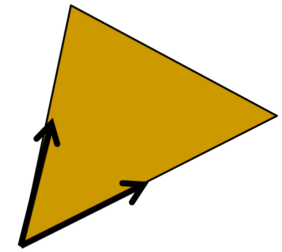
# Step 4: Texture Coordinates (ES)

- Later, in the fragment shader, we will use a set of textures for the terrain's surface and map them to region specified by each patch.
- The texture coordinates of the current vertex are directly given by its tessellation coordinates which have the range [0-1].
- The texture coordinates are passed to the geometry shader.



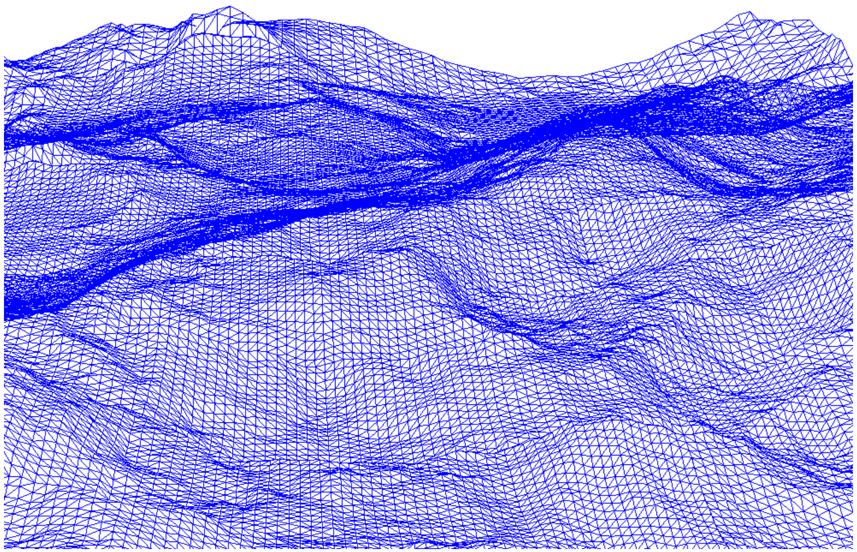
# Step 5: Lighting (GS)

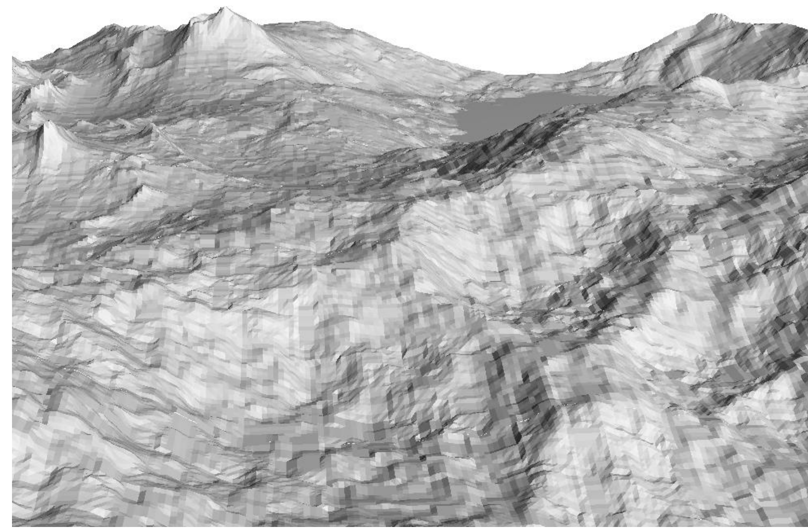




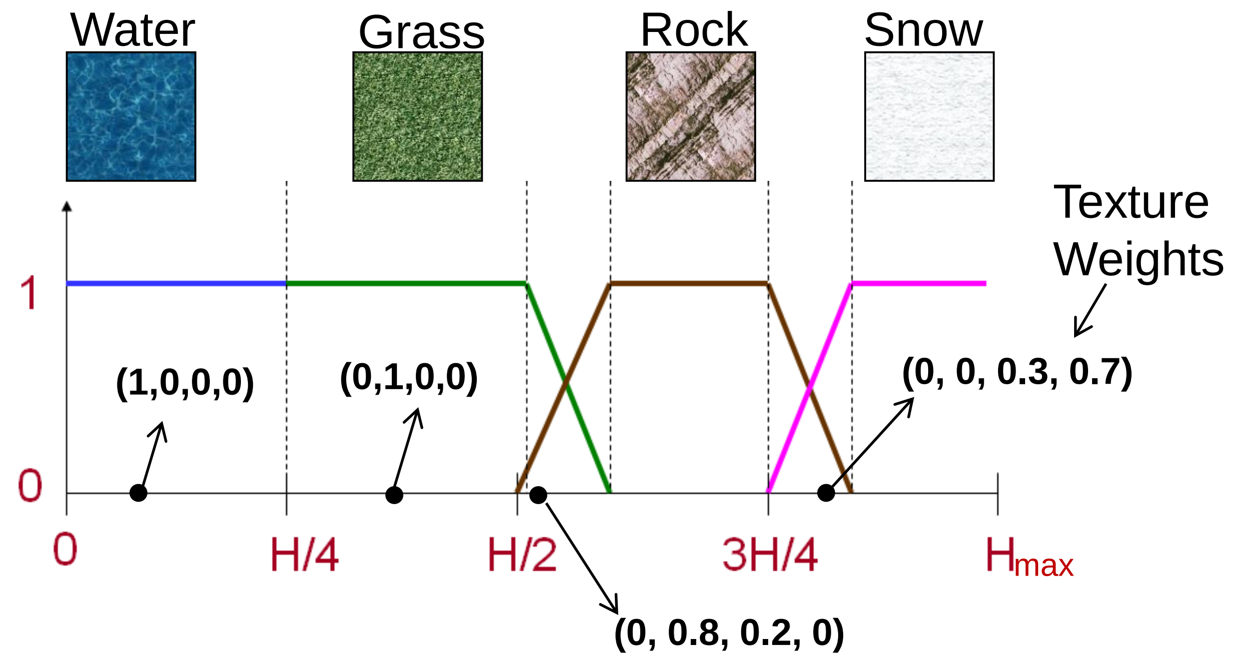
- Lighting calculations cannot be performed inside the evaluation shader as it has access to only one vertex of a triangle created by the primitive generator.
- The geometry shader receives all three vertices of a triangle in the array `gl[in].gl_Position`.
- GS outputs the diffuse term of lighting $\boldsymbol{l}\cdot\boldsymbol{n}$ and the vertices of the current triangle in clip coordinates.

# Step 5: Lighting (GS)





# Step 6: Height Based Texturing (GS)




Geometry Shader:

- Receives the texture coordinates of all three vertices.
- Each vertex is assigned 4 texture weights ($w_1$, $w_2$, $w_3$, $w_4$) based on the height (y) of that vertex.
- The geometry shader outputs the above weights for each vertex: `out vec4 texWgts;`

# Step 6: Multi-Texturing

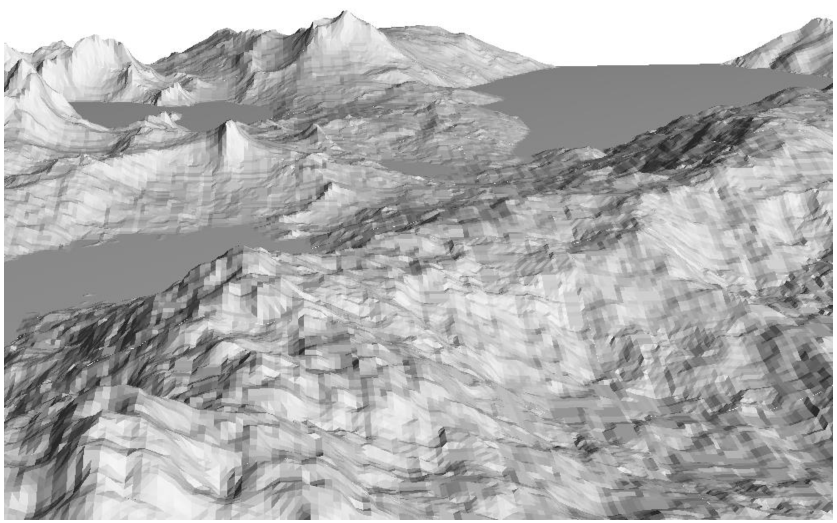
Fragment Shader:

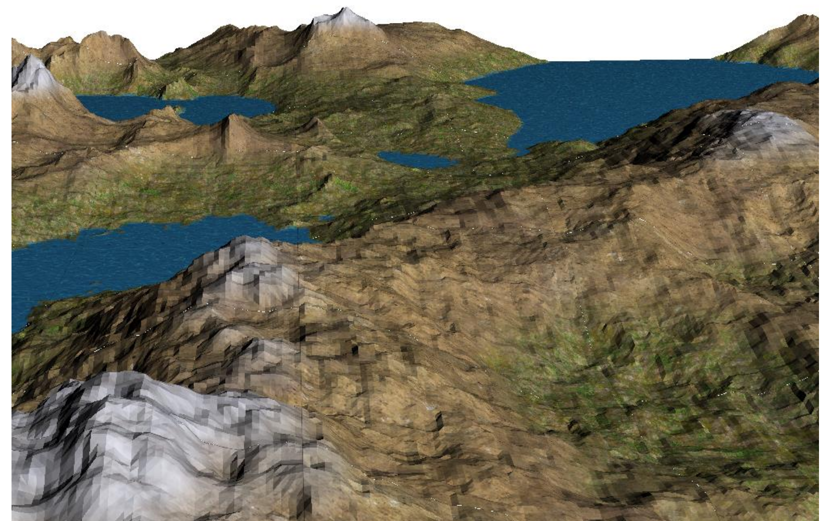
- Receives the interpolated values of
  - Texture coordinates,
  - Texture weights,
  - Diffuse lighting term
- Uses texture coordinates to get colour values from textures
- Multi-texturing: Computes colour as the weighted sum (using texture weights received from GS) of the colours obtained from textures.
- Computes fragment colour as the value of the above colour scaled by the diffuse term.

# Multi-Texturing

•Rough lighting

- lighting is done through finding the face normals



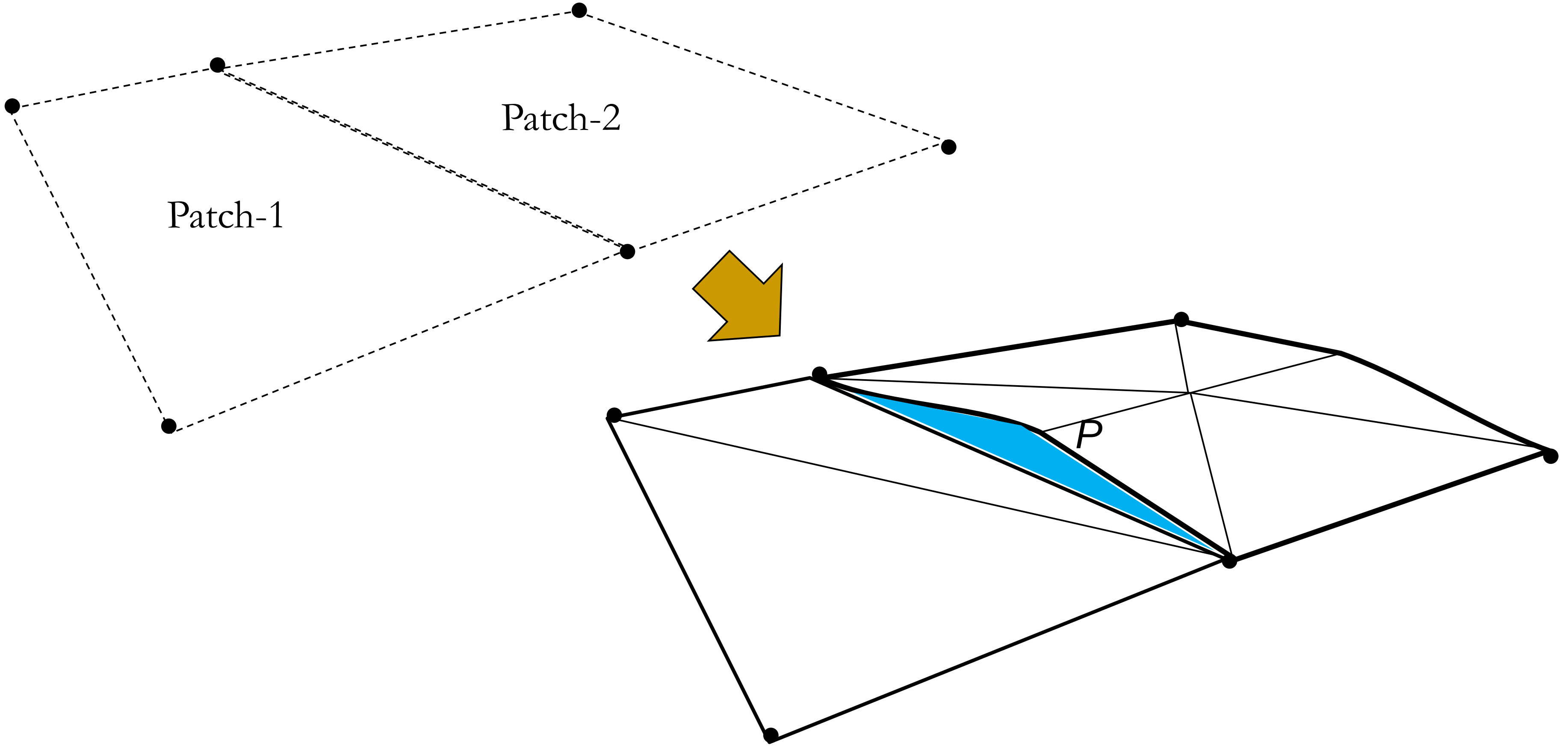


•Smooth lighting

- Find new vertex normals

# Terrain Microcracking

When two adjacent regions are tessellated using different levels, cracks can appear on the surface.



# Terrain Microcracking

- fix this to get an extra points