Planetary Terrain

Quadtree Planets for Unity

Contents

1 Getting Started

1. **Create a Noise Module.** Noise Modules store a noise tree and can be created with the node editor. For now, we'll start simple. Create new a *Node Graph* (right click and create in project view), create a Generator and a Saving Node with RMB. Connect the Generator Node's output to the Saving Node. Choose a filename and press *Serialize*.
2. **Generate a heightmap.** You can generate a heightmap from a Noise Module with the built-in Heightmap Generator. Open *Planetary Terrain -> Heightmap Generator.* Press *Generate Preview* if you want, then press *Generate* (This will take a while). Alternatively, you can assign the Noise Module directly to the planet later.
3. **Create a Material for the planet.** Create a new Material and select *Planetary Terrain -> Planet Surface Shader* as Shader. Assign your textures sorted by the height they should be used at, from lowest to highest. Use the *Texture Scale* field for tiling the texture.
4. **Setup the planet.** Create an empty GameObject and add the *Planet* Component. Assign your Material and Heightmap under the *Visual* and *Generation* Tabs respectively. If you didn't generate a heightmap previously, change *Generation Mode* to Noise and assign your Noise Module.
5. **Add Gravity and an Atmosphere (Optional).** If you want you can add the *Atmosphere* and/or *Spherical Gravity* Components. When using *Spherical Gravity*, make sure that your object has a Rigidbody, is tagged "Gravity" and to disable the standard gravity in *Edit -> Project Settings -> Physics -> Gravity*
6. **Play!**

2 Height Providers

A planet can use different data sources for terrain elevation. These are called Height Providers and implement the interface IHeightProvider. A few different Height Providers are included:

- **Heightmap:** Uses a simple equirectangular heightmap of any resolution for terrain generation. Heightmaps are saved with the extension .bytes.
- **Streaming Heightmap:** Like heightmap, but only a fraction of the actual heightmap is kept in memory. This is very useful for large heightmaps to use less memory.
- **Noise:** Uses a serialized Noise Module for terrain generation. Noise Modules can be created with the Node Editor and encode a combination of different generators (Perlin, Simplex, …) and operators (Multiply, Blend, Curve, …) that produce a height value from the 3D position of a vertex. This is by far the most versatile height provider.
- **Hybrid:** Combines a heightmap for the macro scale and a noise module for smaller details.
- **Const:** Returns a const height value, useful for flat planets or oceans.
- **Compute Shader:** Does noise-based terrain generation on the GPU. Compute shaders can also be created in the node editor. They are not as versatile as CPU-bound noise

modules, but allow for faster terrain generation if the target platform has a fast, modern GPU.

2.1 Implementing Custom Height Providers

1. Create a class that implements *IHeightProvider* in HeightProvider.cs. *HeightAtXYZ()* needs to return a deterministic height value for pos, which is a point on the unit sphere. *Init()* is called once in the first frame.
2. Add your custom Height Provider to the *HeightProviderType* enum, to *SerializedInheritedClasses* and to the switch in *Initialize()* (all Planet.cs).
3. Create a GUI for your Height Provider in *PlanetEditor.cs* *switch (planet.serializedInherited.heightProviderType).* Alternatively, you can just edit *SerializedInherited* directly at the bottom of the debug tab of your planet.

3 Node Editor

The node editor is a utility to create Noise Modules by combining different types of noise. The Noise Module can then be assigned to a planet directly or be used to generate a heightmap.

To create a new Node Graph, right click in the Project tab and select *Create -> Node Graph.* Double click on the Graph to open the editor. Nodes can be created with RMB. The noise graph always starts with Generator Nodes, then Operator Nodes can be used to combine multiple Generators to create more complex surfaces. The final node must be a Saving Node, which can serialize the noise graph into a Noise Module or a Compute Shader. The graph used for the earth-like example planets is included as an example.

- **Generator Node:** This Node represents a 3D noise function. It is given the normalized location of a vertex on the sphere and returns a height value.
- **Operator Node:** This Node performs different operations on an input value
  - **Select:** Selects between Input 1 and 2 based on Input 0. If Input 0 is in the interval [min, max], Input 2 is returned. Otherwise, Input 1 is returned. The larger Fall Off, the more the transition is smoothed at the edges of the interval. On the GPU, min is pegged to -1 to avoid conditionals.
  - **Curve:** Applies the value to a curve. For convenience, the height value (which usually ranges from -1 to 1) is scaled to 0 to 1. After the curve has been applied, the height value is scaled back.
  - **Blend:** Blends/lerps between to two input values. If Bias = 0, Input 0 is returned. If Bias = 1, Input 1 is returned.
  - **Remap:** Remaps (scales and offsets) the 3-dimensional input coordinates for all input nodes. This is not supported on the GPU. Do not use this in combination with a Heightmap Node.
  - **Add:** Adds two inputs together. The output can be outside of the standard -1 to 1 range.
  - **Subtract:** Subtracts Input 1 from Input 0. The output can be outside of the standard -1 to 1 range.
  - **Multiply:** Multiplies the inputs.

    - **Min:** Returns the lesser of the two inputs.
    - **Max:** Returns the larger of the two inputs.
    - **Scale:** Scales the input (multiplies it with scale), the output can be outside of the standard -1 to 1 range.
    - **Scale Bias:** First scales, then adds Bias to the input. The output can be outside of the standard -1 to 1 range.
    - **Abs:** Returns the absolute value of the Input.
    - **Invert:** Returns input with flipped sign/multiplied with -1.
    - **Clamp:** Return the input clamped into an interval. Useful if you want to make sure the output value is in the standard -1 to 1 range.
    - **Const:** Returns a constant value.
    - **Terrace:** Applies the terrace effect to the input. Each element in the array is the elevation of a terrace.
- **Heightmap Node:** Allows you to use data from a heightmap in the Node Editor/a Noise Module. Do not use a heightmap node in combination with a remap node. *Name* is only relevant if you want to generate a Compute Shader: Set a name and use the *Load Heightmap Compute Shader* MonoBehaviour to load a heightmap into that name on startup. Using large heightmaps on the GPU is not recommended as every pixel has to be stored as a float. This doubles (16-bit heightmap) or quadruples (8-bit) the required memory.
- **From Saved Node:** Allows you to use an existing Noise Module in your Node Graph.

3.1 Noise Parameters

Seed: Changes the output of the noise function.
Octaves: Amount of detail, or how many times the noise is layered over itself with a higher frequency. This should be higher for larger planets but can be decreased when increasing frequency. More octaves means lower performance.
Frequency: Amount of change per unit distance. Increase for bumpier surface.
Lacunarity: Multiplier that determines how quickly the frequency changes for each successive octave.

3.2 Implementing Custom Operators

1. Implement a custom *Module* in Noise.cs. A Noise Module (that you can assign to a planet) is just a serialized tree of *Modules*. Also make sure to create a new *ModuleType* (top of Noise.cs) and assigning it in your *Module's* constructor.
2. Add your custom *Module* to the switch in OperatorNoise.cs *GetModule()*.
3. Create a GUI for your *Module* in OperatorNodeEditor.cs *OnBodyGUI().* Add your *Module* to the *Operator Type* enum and to the switch. Set *numberOfInputs* to the number of input ports your *Module* needs.
4. (Optional) Add a GPU implementation of your *Module* to ComputeShaderGenerator.cs if you want to use it in a Compute Shader.

4 Heightmap Generator

The Heightmap Generator is a utility to create heightmaps and textures from either a Noise Module, a Compute Shader or an existing heightmap.

The Heightmap Generator generates three files – A raw binary heightmap, and two normal .png textures. The textures are not needed for using Planetary Terrain, but can be used for a copy of the planet in Scaled Space or be edited in an image editor and converted back to a raw binary heightmap with the Texture Heightmap to RAW utility.

The Generator can switch between 8 and 16-bit mode. 16-bit heightmaps have 65536 elevation levels as opposed to for 256 an 8-bit heightmap. They are recommended if size isn't an issue, 16-bit heightmaps are double the size of 8-bit heightmaps.

You can also use a RAW heightmap as the input to (re-)generate the aforementioned PNGs.

5 GPU terrain generation

Planetary Terrain also supports generating terrain on the GPU instead of the CPU. Since GPUs are very optimized for parallel computing, you can easily split dozens of Quads simultaneously on a modern GPU, compared to 2-8 on the CPU.

The workflow when using GPU terrain generation is very similar to using normal noise modules. The Saving Node can serialize your Noise Graph into a Compute Shader, which you can assign to your planet or the Heightmap Generator when selecting Compute Shader mode, just like a normal Noise Module.

Unfortunately, the Remap operator is not supported on the GPU.

For low-level access you can also edit the *GetNoise*-function in the Compute Shader directly.

6 Planet Parameters

6.1 General

- **Radius:** The planet's radius.
- **Detail Distances:** The distance (to the player/camera) threshold that quads of a certain level need to be under to split.
- **Calculate MSDs:** Calculate the mean squared deviation (bumpiness) of quads. See the section "MSD/Bumpiness Threshold".
- **Generate Colliders:** Set whether collider should be generated at a subdivision level. Usually you only want colliders on the last subdivision level.
- **LOD Mode behind Camera:** How are Quads behind the camera being handled? If *Compute Render* they are handled just like quads before the camera, i.e. have GameObjects. If *Not Computed*, Quads behind the camera are regarded as not visible, therefore they will be combined and don't have GameObjects.
  - **LOD Extra Range:** Extra range for quads that would be behind the camera if *Not Compute*.
- **Recompute Quad Threshold:** Distance moved after which the distance to the camera of all quads is recomputed. If **Update all Quads simultaneously** == false then this process

is done over multiple frames, with **Max Quads to update per frame**. These values can be optimized using the profiler.

- **Floating Origin (if used):** All planets need to have access to the Floating Origin script, if one is used.
- **Hide Quads in Hierarchy:** Hides quad GameObjects in hierarchy, also makes them unselectable.
- **Quad Size:** Side length/number of vertices per side for quads. Check BaseQuadMeshes.cs if you want a side length that's not in 5, 9, 13, 17...

6.2 Generation

- **Generation Mode**
  - **Heightmap**
    - **Heightmap:** The heightmap TextAsset to use. Can be generated with the *Heightmap Generator*, *Texture Heightmap to RAW* or the *Heightmap Importer*.
    - **Use Bicubic Interpolation:** Interpolate using 16 samples instead of 4. Slower but smoother.
  - **Noise**
    - **Noise:** The Noise Module TextAsset to use.
  - **Hybrid**
    - **Heightmap, Use Bicubic Interpolation:** See Heightmap
    - **Noise:** See Noise.
    - **Noise Divisor:** The larger this number the less impact the Noise Module has on the final terrain elevation.
  - **Const**
    - **Constant Height:** The constant elevation to use, from 0 to 1.
  - **Compute Shader**
    - **Compute Shader:** The Compute Shader used for terrain generation. Can be created in the Node Editor.
  - **Streaming Heightmap:**
    - **Base Heightmap:** Low-resolution copy of the actual heightmap used in places where high-res data is not available.
    - **Path:** Path to the high-res heightmap .bytes file. Be careful, this path can be invalid after building! (this cannot be assigned as a TextAsset like other heightmaps because we need a FileStream)
    - **Use Bicubic Interpolation:** See Heightmap.
    - **Loaded Area Size:** Size of the high-res area that is kept in memory. (1, 1) would be the complete high-res heightmap.
    - **Reload Threshold:** Threshold for reloading the area in memory. 1 corresponds to the planet's radius.
- **Height Scale:** Max elevation as fraction of the planet radius, e.g. if radius = 100 and height scale = 0.1 then the max elevation is 110.
- **Quads Splitting Simultaneously:** Number of quads that can split at the same time. Higher number means faster load time but also higher CPU load.

- **Use Scaled Space:** Whether to use Scaled Space, i.e. disable quad generation when far away.
  - **Scaled Space Factor:** Factor by which Scaled Space is smaller than normal space. Needs to be set to the same number as on the Scaled Space script.
  - **Generate Scaled Space Copy:** Generates a copy of this planet in Scaled Space.

6.3 Visual

- **Planet Material:** Material that will be assigned to all quads.
- **UV Type:** Type of UV that will be generated for quads
  - **Cube:** With **UV Scale** = 1, the texture will span over each of the six sides of the spherified cube once.
  - **Quad:** The texture spans once over the smallest quad. Quads one level up tile the texture 2x, two levels up tile the texture 4x …
- **Visibility Sphere Radius Mod:** The visibility sphere is multiplied by this modifier. Usually this should be one, for bumpy planets maybe 1.5. If you want all quads to always have GameObjects, even if they aren't visible, just set this to Infinity.
- **Slope Texture Type:**
  - **Fade:** Fades in the slope texture in range [**Slope Angle** - **Fade-in Angle, Slope Angle**].
  - **Threshold:** Uses slope texture if the slope is larger than **Slope Angle**.
- **Slope Texture:** Index of texture used on slopes (0 to 5).
- **Texture Selection Type:** See Texturing.

6.4 Foliage

- **Generate Details:** Enable foliage generation.
- **Foliage Biomes:** Textures/Biomes to generate foliage in.
- **Generate Grass:** Enable grass generation.
  - **Grass per Quad**
  - **Grass Material:** Point Cloud Grass Material. Use the included shader *PlanetaryTerrain/Grass Geometry Shader*.
- **Detail Level:** Subdivision level at which and after which details/foliage will be generated.
- **Detail Distance:** Distance below which details/foliage will be generated.
- **Details Generating Simultaneously:** Number of quads that generate their foliage at the same time.
- **Add Mesh, Add Prefab, Experimental:** See Foliage.

6.5 Events

- **Finished generating Quads:** Called every time the planet has finished generating quads.
- **Player entered Scaled Space:** Called every time the player enters scaled space, also on startup if the player is in scaled space.
- **Player left Scaled Space:** Called every time the player leaves scaled space, also on startup if the player is not in scaled space.
- **All Events are also available as delegate voids if you don't want the overhead of a Unity Event.**

7 Texturing

7.1 Materials

Planetary Terrain includes different Materials that allow the use of different textures on a planet. *Planet Surface Shader (Bump)* is a simple shader that selects/fades between six textures based on the generated data stored in the mesh.

PlanetFadeShader (Bump) additionally fades between the two textures supplied for each biome, based on its distance to the camera, **Fade Start** and **Fade End**.

Both of these Materials allow for the use of up to six different biomes. If you use fewer biomes, I recommend modifying the shaders to exclude those textures for improved performance.

7.2 Texture Providers

Intensity values for each of the six textures (or biomes) of a *PlanetSurfaceShader*-Material are encoded in the mesh's vertex color (texture 0 to 3) and uv4 (texture 4 and 5) channels. When generating the mesh, these values are generated by a Texture Provider, a class the implements *ITextureProvider*.

- **Gradient:** Allows you to define key points for each texture. In between two key points the gradient lerps/fades between the two textures.
- **Range:** Allows you to set a range for each texture. Where ranges overlap, the provider fades between the two or more textures.
- **Splatmap:** Allows you to set splatmaps for texturing. For each vertex, the splatmap is sampled and returns the intensity of each texture at that point. Splatmaps can be Texture2Ds or standard heightmaps. Since there are six texture channels, you need two Texture2Ds or six heightmaps to encode each channel. When using Texture2Ds, rgb of Splatmap A maps to channels 0 to 2, rgb of Splatmap B maps to channels 3 to 5. When using Heightmaps, the texture channel for each heightmap can be selected.

7.3 Implementing Custom Texture Providers

1. Create a class that implements *ITextureProvider* in TextureProvider.cs. *EvaluateTexture()* needs to return a float array of length 6, each float is the intensity of one texture/biome. It is passed the height/elevation (0 to 1) and the point on the unit sphere.
2. Add your custom Texture Provider to the *TextureProviderType* enum, to *SerializedInheritedClasses* and to the switch in *Initialize()* (all in Planet.cs).
3. Create a GUI for your Height Provider in *PlanetEditor.cs.*

8 Atmosphere Shaders

Currently, two atmosphere shaders are included. The first one is based on Scrawk's port of the GPU-GEMS Atmospheric Scattering. It is very fast and simple. To use it, just attach the *Atmosphere* script to your planet.

The second shader is based on Scrawk's port of Brunetons Atmospheric Scattering. This effect is more advanced and thus slower. You can apply it by attaching the *AtmosphereMesh*

script to a Scaled Space Copy and assigning the required values (Compute: *Precomputation*; Material: *RenderSkyMesh*).

9 Foliage

You can generate foliage in the form of meshes, prefabs and grass with the included grass shader on planets:

1. **Go select the Foliage Tab.**
2. **Enable *Generate Details*.** Then select Foliage Biomes, i.e. the biomes in which foliage will be generated.
3. **Generate Grass?** If you want to generate grass, enable it. For now, you can use the included Grass Material under “Planetary Terrain/Materials”. You also need to set the amount of grass to be generated per quad.
4. **Set Detail Level and Detail Distance.** Detail Level sets the level at and after which foliage is generated if the distance is smaller than Detail Distance. Usually, Detail Level is just the size of Detail Distances so that foliage is only generated in the highest-level quads.
5. **Add Detail Meshes and Detail Prefabs.** Now you can add other objects that will be spawned. Detail Meshes have better performance, because they don’t require a GameObject. That also means that you cannot generate colliders for the object, or spawn complex objects with multiple meshes. If you want to do that, use a Detail Prefab.

9.1 Experimental Foliage backend

The normal foliage system is quad-dependent, foliage is generated per quad. This works very well if the quads are small, but is limiting if this is not the case. Enormous amounts of points need to be generated and rendered if a large quad is densely populated with foliage, even though most of these points are way too far away to be visible.

The new experimental foliage system aims to solve this problem by being quad-independent. Foliage is always generated in a square around the player, no matter how big or small quads are. This allows for a high foliage density even with very large quads. It is less reliable than the normal foliage system though, which is why it should only be used if necessary. You can enable the experimental backend for grass, meshes and prefabs in the *Experimental* dropdown. You also need to attach the *Foliage Experimental* script to your player or camera. *Foliage Experimental* has two generation modes, Raycast (every point is generated with a ray) and Mesh-Based (uses the mesh of quads to generate points). Both need colliders on the quads on which foliage should be generated.

10 Large Planets

10.1 Floating Origin

The asset also features a floating origin system, it allows for very large planets. The origin is always kept close to the player, where the most precision is needed. You can use it by adding the *Floating Origin* Component to your Planet and assigning your Main Camera and

Transforms that should be shifted when the threshold is exceeded (e.g. buildings on the planet).

10.2 Cameras

When using small planets, Planetary Terrain works perfectly well with just a single Main Camera. But as planets get bigger, some issues start to arise. Larger planet radii call for an ever-increasing render distance, i.e. distance between the near and far planes of the Main Camera.

At some point however, you will run into Z-buffer issues. Unity's 32-bit depth precision is too low for the extreme distances. This issue can be resolved by splitting the depth into two cameras, a near and a far camera. This is implemented in the Earth-sized Planet, GPUTerrainGeneration and System scenes.

The far camera is the near camera's child, so the cameras are always in the same place. The difference between the two cameras are in the clipping planes. In Earth-sized Planet for example, the near camera renders from 0.3 to 5000 units, the far camera from 5000 to 3,000,000 units.

Another difference is depth: The far camera's depth needs to be lower than the near camera's, such that it renders first. The near camera's Clear Flags need to be set to Depth Only, when using Scaled Space this also applies to the far camera. Clear Flags allows the picture to be overwritten by another camera.

10.3 Scaled Space

Scaled Space is a layer in the scene where everything is scaled up by a factor of 100,000x (by default). It can be used for a copy of the planet that is shown when far away, where computing the actual quadtree is unnecessary, or for moons and far away planets.

It is implemented as an additional layer that isn't rendered by the normal camera(s), but only by the Scaled Space camera. The Scaled Space camera's position is then set to the main camera position divided by a scaling factor (e.g. 100,000). This scales everything by the scaling factor, which allows you to use a relatively small sphere as a replacement for the planet when far away, the *Scaled Space Copy*.

Using Scaled Space:

1. **Create a second Camera.** Also create a new layer called 'Scaled Space' and set the culling mask so that it is the only layer visible to that Camera.
2. **Add the *Scaled Space* Component.** Assign it to the main planet.
3. **Change the Main Camera's Clear Flags to 'Depth only'.** Remove the Scaled Space layer from the Culling Mask. Also make sure that the Main Camera's Depth is higher than that of the Scaled Space Camera. (Do this for both cameras if you are using a near and far camera.)
4. **Generate a Heightmap.** Use the Heightmap Generator to generate a heightmap. The alpha value of the colors will later set the smoothness on the copy in Scaled Space. If you want a reflective ocean, turn it to zero on all colors except the ocean color.

5. **Create a new Material.** Add texture and normal map from the Heightmap Generator. Set the Smoothness Source to Albedo Alpha. A normal map can be generated from the heightmap in Unity's texture import settings. Also increase the max size to 8192 for both textures.
6. **Enable 'Use Scaled Space' and press 'Create Scaled Space Copy' in the *Generation* tab.** Assign your Material in the *Visual* tab.
7. **Adjust 'Scaled Space Distance' to set when the planet switches to Scaled Space.** You can also lower 'Visibility Sphere Radius Mod', then quads in the background will be replaced with your Scaled Space copy.

10.4 Creating planetary systems

You can create planetary systems by moving and rotating planets, their Scaled Space copies will be moved along with them in Scaled Space. Two simple scripts, *Rotate* and *RotateAroundPlanet,* are included, along with the *System* example scene.

MSD/Terrain bumpiness threshold

In addition to the linear distance threshold that every planet uses you can also use a quad's mean standard deviation (MSD) (basically bumpiness) to set whether it should split or not in a given situation.

Simply enable *Calculate MSDs* in the *General* tab of your Planet. Then you can set an MSD threshold for each one of your detail levels. Usually everything but the last 1 to 3 levels are set to zero. For those levels there is no MSD threshold and even perfectly spherical (flat) quads would be split. The last three levels can for example be set to 0.2, 1 and 10, then only very bumpy quads with an MSD of over ten will reach the last detail level and a certain degree of bumpiness is required for the second to last and third to last levels.

11 API

11.1 Planet

```
/// Called when entering or exiting Scaled Space
public delegate void ScaledSpaceStateChanged(bool inScaledSpace);
public ScaledSpaceStateChanged scaledSpaceStateChanged;

/// Called every time the planet has finished generating quads.
public delegate void FinishedGeneration();
public FinishedGeneration finishedGeneration;

/// Instantiates object on this planet.
/// LatLon: position, latitude (x) from 0 to 180, longitude (y) from 0 to 360
/// pos: position, cartesian coordinates, x, y and z, ranging from -1 to 1, relative to planet
public GameObject InstantiateOnPlanet(GameObject objToInst, Vector2 LatLon, float offsetUp = 0f)
public GameObject InstantiateOnPlanet(GameObject objToInst, Vector3 pos, float offsetUp = 0f)

/// Returns rotation to stand up straight at specified position
public Quaternion RotationAtPosition(Vector3 pos, bool posRelativeToPlanet = false)
public Quaternion RotationAtPosition(Vector2 LatLon)
```

11.2 MathFunctions

```
/// Converts from spherical to cartesian coordinates
/// lat: latitude in degrees 0-180
/// lon: longitude in degrees 0-360
public static Vector3d LatLonToXyz(double lat, double lon, double radius)

/// latlon: spherical coordinates in degrees
public static Vector3 LatLonToXyz(Vector2 latlon, float radius)

/// Converts from cartesian to spherical coordinates, returns in degrees. Y is longitude, in range 0
to 360. X is latitude in range 0 (south pole) to 180 (north pole).
/// pos: Cartesian coordinates relative to sphere center.
/// radius: Radius, i.e. distance to sphere center, of point.
public static Vector2 XyzToLatLon(Vector3 pos, float radius)
public static Vector2 XyzToLatLon(Vector3d pos, double radius)

public static Vector2 XyzToLatLon(Vector3 pos)
public static Vector2 XyzToLatLon(Vector3d pos)
```

12 Importing Heightmaps

There are multiple ways to import heightmaps. The easiest way is to just import them like any other texture and then using the *Planetary Terrain -> Utils -> Texture Heightmap to RAW* utility. This method is limited by Unity's Texture2D however. It only supports textures with a max size of 8192x8192 and 8-bit precision.

To get around these limitations, there is a second utility to import heightmaps, the *Planetary Terrain -> Utils -> Heightmap Importer*. It can convert grayscale RAW images saved with GIMP or Photoshop, or uncompressed TIFFs, to a heightmap. If a 16-bit image is not imported properly, try reversing the byte order.

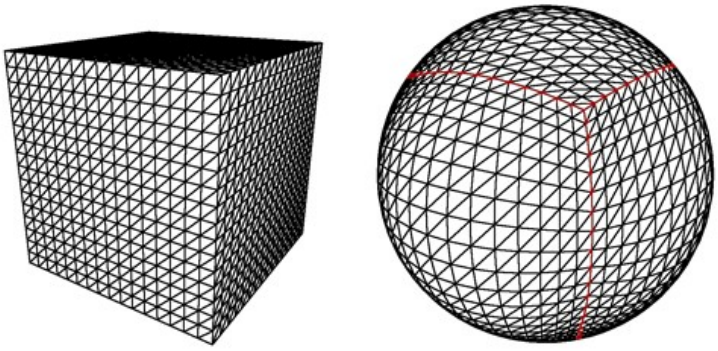
Planetary Terrain heightmaps have a 9-byte header that encodes width (int), height (int) and is16bit (bool). After that the raw pixels follow (little endian if 16-bit).

13 Background

This terrain system is based on the quadtree LOD system. It is quite simple – you start with a grid of 2x2 quads. When a higher level of detail is needed in a certain area, that quad is split into four smaller ones, each which can be subdivided again if needed.

For use with planets, the planar quadtree has to be spherified:

1. A cube is formed from six quads. It always has the same size regardless of the planet radius, as it is scaled up later. This is handled by the *Planet* class, *InstantiateBaseQuads()*.
2. Every vertex is normalized to form a unit sphere, and then scaled by the terrain height at that point. This is handled by the *MeshGenerator* class on another thread. Generation is started with *Quad.StartMeshGeneration()*.
3. If the distance to an existing quad is lower than *detailDistances[quad.level]*, the existing Quad can split into four smaller ones. This is handled in *Quad.Split()*.
4. If the distance to an existing Quad is higher than *detailDistances[quad.level]*, and the Quad has already split (*Quad.hasSplit*), it combines via *Quad.Combine()*. All of the Quads children are then deleted.

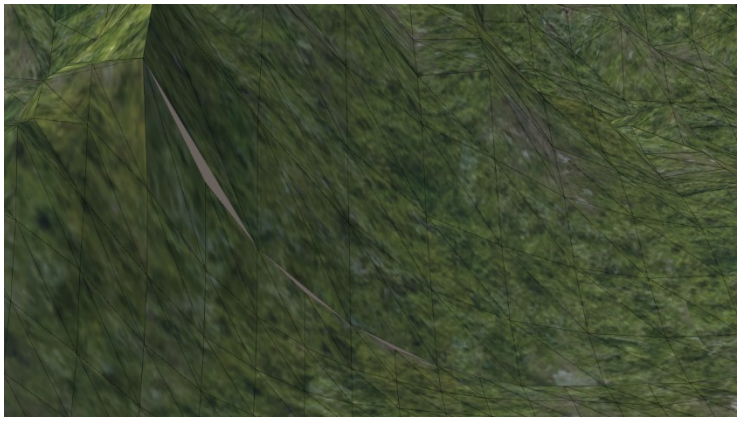


Cube before and after spherification.

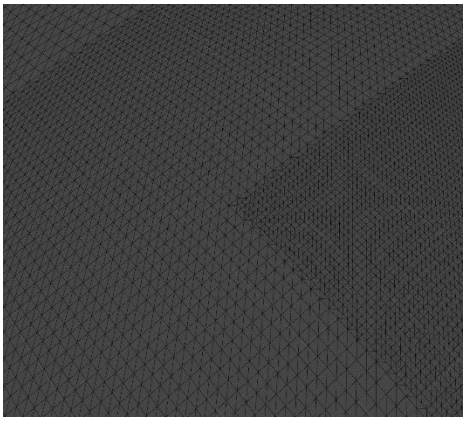
Each quad is given a unique index (*Quad.index*) based on the subdivisions required to generate it. The base quads have base indicies like {0, 1} or {2, 1} (for more info, check QuadNeighbor.cs). When a quad subdivides, the child's index is the index of the parent quad with the number (0 to 3) of the child appended. If base quad {0, 1} splits, its children are quads {0, 1, 0}, {0, 1, 1}, {0, 1, 2} and {0, 1, 3}.

For performance reasons, an index is not stored in an array but in a ulong. Two bits (0 to 3) represent one element of the index. The first 6 bits of the ulong are used to encode how long the following id is, otherwise a zero would be indistinguishable from an unused element. To encode an int[] into a ulong, use *QuadNeighbor.Encode()*. To decode a ulong, use *QuadNeighbor.Decode()*. You can find a quad by its ulong index in the Dictionary *Planet.quadIndicies*.

Following this naive approach, cracks appear at edges where quads of a higher level of detail border ones with lower a lower one. This problem is solved by removing every other vertex on the quad with the higher LOD (quad edge fan). If a neighbor is of a lower LOD, the quad changes its own mesh to one with the right configuration of edge fans by changing its triangles array, the vertices aren't modified.



Cracks



Quadtree Edge Fans

This however requires knowledge of the neighbors of each quad to check if edge fans are needed. This process of finding neighbors is further complicated by the spherification of the quadtree. In this asset it is handled by the *QuadNeighbor* class. An explanation of how it works can be found in the code.