

## **Terrafirma**

### **TerraManager**

Every thing starts here so i figure this is where I will start..

-

### **Intialize**

This Function is used when the editor starts, when entering game mode and when component of this type is added to the scene.

First off the system calls CheckForManagerand compares the returned manager to the one calling it to verify it is the only terrManager in the scene.

Next it loads a list of the scene terrains by calling getSceneTerrainListto be used during processing.

### **CheckForMeshUpdates**

Calls a function in the TerraTerrain object of the same name. This checks each terrain against the input camera. If the terrain or camera moves, it'll first check to see if these changes in position were enough to affect the level. If so, it flags the terrain mangager telling it needs to adjust the scene terrains.

There is a flag per terrain, keeping track of which whether or not the level gragh has changed for that instance. I would have to test the filter stage when i re-integarte neighbor support later to see if it doesn't break, though i figure it should, for now though it's just updating every terrain in the scene though single check for ech of the functions below would be enough to fix that.

\* \* \* \*

*These Functions call functions in the scene TerraTerrain object of the same name; Here is brief discription of each.*

-

### **AdjustLevels**

Used to filter the terrain level gragh to ensure patches are compitible with neighbors for stitching.

This is called three times per terrain before batching.\*\* See above note in regards to skipping non-flagged/changed terrains.

### **BatchInstances**

Used to batch the adjusted/filtered terrain level graph in order to reduce draw calls. This is called after the filtering stage. This is called once per terrain\*\* See above note in regards to skipping non-flagged/changed terrains.

### **UpdateMeshReflection**

Used to update parts of the map where needed based on the generated batch list. \*\* See above note in regards to skipping non-flagged/changed terrains.

-

### **CheckForManager**

Checks to see if the scene has terrain manager; If so it returns it.

-

### **getSceneTerrainList**

Grabs all the terrains in the scene and returns them in a list/flat array.

\* \* \* \*

*There are a couple more utility functions for managing the scene list but i figure the are pretty self explanatory.*

### **TerraTerrain**

#### **Intialize**

This Function is used when the editor starts, when entering game mode and when component of this type is added to the scene.

First the component checks to see if it has a terraData object or not. If not it creates a new one.

After the it verifies the terraData structure is present, it then makes sure this terrain is linked to the terraData, by calling LinkTo, on the terraData object. This just gives the terraData object a pointer telling it's attached to the terrain;

Next it verifies there is a manager present in the scene, if not it add ones. It then calls function CheckForTerrainChanges, which simply rebuilds the terrain managers list, if any new terrains have been added or removed. It's called here, in cases where the function is called during component creation.

Next it starts up the materialsets, which are used for rendering the base map, differnt materials at differnt distance, etc.

And finally it finishes the intiatial checks by calling MeshHandlerPreload. This function takes the rendering parameters and the terraData settings and gets everything ready for the terrain to process.

### **InjectTerrainData**

This Function is basically the same as Intialize, the only differnce is it overrides the current terraData with a new one built from the inputted unity terrainData;

-  
  
-

### **MeshHandlerPreload**

As mentioned above; this function takes the rendering parameters and the terraData settings and gets everything ready for the terrain to process and calls the functions;

setUpUtilityTables, levelMapSetup, BuildObjectStructures, BatchSetUp, GetClearBuffPointer, CalculateLODs.

### **setUpUtilityTables**

This function builds a few tables, such as precomputed log-base2 tables, patch level axis vertex counts, etc, to speed up calculations.

### **levelMapSetup**

This function builds all the data holders used for storing terrains patch levels.

-

## **BuildObjectStructures**

This function starts off by building the mesh templates used for projecting the terrain via the shader. As of now the system uses, 8 meshes by level (Max 5 levels), for a max mesh count of 40 pre computed planes. This can be reduced, was done naively as it didnt really affect anything enough to warrent a slight more complex look up; Eventually this will change;

-

### **For the 8 planes per level; the vertex counts come in;**

1×1 quad; 2×2 verts (shared)

2×2 quad; 3×3 verts (shared)

4×4 quad; 5×5 verts (shared)

8×8 quad; 9×9 verts (shared)

16×16 quad; 17×17 verts (shared)

32×32 quad; 33×33 verts (shared)

64×64 quad; 65×65 verts (shared)

128×128 quad; 129×129 verts (shared)

\* \* \* \*

*The reason for each set per level is, we store the skips (number or vertices missing between two vertices) to the vertex of the mesh. This can be circumvented a few ways as mentioned it was done some what naively, If we can can get around this, and have a method of telling each renderer its levels via property blocks per instance, the sets can be reduced down to one, or just 8 planes for all levels.*

\* \* \* \*

Depending on the axis vertex/quad count for a single patch at any given level will determine which of these planes it can use;

*For example, using a quad count of 32×32 at level 0 for a 1×1 patch or the smallest quad tree leaf*

sized unbatched,

**level 0 could utilize :**

32×32 quad; 33 x 33 verts (shared) for a 1×1 group of level 0 patches

64×64 quad; 65 x 65 verts (shared) for a 2×2 group of level 0 patches

128×128 quad; 129×129 verts (shared) for a 4×4 group of level 0 patches

**level 1 would then use :**

16×16 quad; 17 x 17 verts (shared) for a 1×1 group of level 1 patches

32×32 quad; 33 x 33 verts (shared) for a 2×2 group of level 1 patches

64×64 quad; 65 x 65 verts (shared) for a 4×4 group of level 1 patches

128×128 quad; 129×129 verts (shared) for a 8×8 group of level 1 patches

**level 2 would then use :**

8×8 quad; 9 x 9 verts (shared) for a 1×1 group of level 1 patches

16×16 quad; 17 x 17 verts (shared) for a 2×2 group of level 1 patches

32×32 quad; 33 x 33 verts (shared) for a 4×4 group of level 1 patches

64×64 quad; 65 x 65 verts (shared) for a 8×8 group of level 1 patches

128×128 quad; 129×129 verts (shared) for a 16×16 group of level 1 patches

And so on and so fourth;

\* \* \* \*

*This may not seem like it wouldn't make much of a difference considering the batch sizes, but it in all my testing cases it reduced draw calls down to 1/4 of what unity would use for the same triangle counts, even with the more accurate LOD scheme i recently added. There was actually no noticeable difference with the new LOD metrics. I also had the option to batch every size aside from binary numbers but again there wasn't any noticeable difference in batch count so i opted to avoid the overhead of having so many extra meshes.*

\* \* \* \*

The rest of this function is rather trivial, it builds a grid of gameobjects with renders/meshfilters to project the terrain onto that reflect the level graph built earlier, and then finish off by call RefreshTerrainRendererMaterial, which updates the terrain material sets.

-

### **BatchSetup**

Like the levelMapSetup, this function builds all the data holders used for storing the terrains batching related data.

-

### **GetClearBuffPointer**

This simply builds a empty graph used for resetting various system tables, such as the level one. The clear buffer is an empty flat array the same length as 80 percent of the arrays used by the sytem. Used to avoid the creation of new data, by copying from when needed.

-

### **CalculateLODs**

Calculates all the patch errors for the entire terrain quad tree, and stores them. Patch errors are the difference in a terrain leaf topology vs the orginal, highest level topology.

We generate one error per leaf of the terrain quad tree, with the exception of level 0; Which always has an error of "0" as it's 'perfect' or not decimated;

### **CheckForMeshUpdates**

This first simply checks to see if a camera was inputed, if not it returns and does nothing. (This is handled as of now through the manager which just passes the main camera to the system for now. More would need to be done for full rendering via scene cameras.)

Next we pass the cam position to a simple check, `CheckTerrainCameraPosition`, which tells us if the terrain has been: reset, moved, or this cameras position differs from the last inputed one (For multi camera, cameras would need their own instances of level tables, among other things like last inputed position).

if `CheckTerrainCameraPosition` returns true; We next move on to resample the quad tree to check for any inconsistencies via `Sample`.

If the sampling finishes and the terrain has become flagged for an update.

If the system requires an update it then copies the level map to a separate buffer for filtering, stitching, etc.

This other buffer `AdjustedLevels` is the true patch level grid.

The reason for having two level maps is to save processing time. The quad tree sampling builds the "raw patch" level map, `Levels`, the system only works on `AdjustedLevels` and the mesh reflection if `Levels` has changed.

The reason for this is the sampling builds an interpretation of a raw quad tree and the filtering stage breaks from that formatting, meaning we would need to apply filtering again to verify the level map has changed.

So I circumvented by checking if the `Levels` has changed during sampling, and only then apply filtering, to `AdjustedLevels`. So we can check to see even after filtering if just the sampling process and just `Levels` array has changed, in order to determine if a or however many patches have changed. Whereas if I just used one map, I would need to push it through the filtering process as well to verify the same thing.

### **CheckTerrainCameraPosition**

This checks to see if the current position of the terrain or camera has changed from the last or if the terrain has been flagged for a level map reset;

-

### **Sample**

This function is used to sample/build the quad tree interpretation of the terrain. Firstly it does a few calculations to get the distance of the patch. Once the distance is determined there are two options on how the tree nodes can go about breaking down :

#### Option 0 :

The terrain can test against the error structure built calling CalculateLODs. Here we simply check to see if the quad tree node were currently at has an error that is below the current set pixel error; if not we break the leaf and test again on the lower leaf nodes, till eventually we get to 0, no error, or level 0 leaf nodes.

#### Option 1 :

The terrain can test pixelerror simply against distance. Here we simply check to see if the quad tree node were currently at's current node's scaled width/(distance to it) is less then the pixel error; Again if not we break the leaf and test again on the lower leaf nodes, till eventually we get to 0, no error, or level 0 leaf nodes.

Once a node is finalized as with in the error metric choosen the system then goes to inject the level. Here we simply sweep across the region the node occupies in the level map and incode the new level.

If a new level differs for even a single patch we from the last update we flag the terrain as updated. At this check is where we also apply culling, the material set flag, essentially anything we want the filter stage (culling), batching (material sets), to work with.

When encoding values, check which materialset the patch should use, using the distance calculated before, check if the batch is culled in and skip it so the filter, batching and rendering stages can as well, test patch height for culling, etc. This is where the filtering vs balancing really shines in favor of filtering.

-

#### AdjustLevels

This function ensures the patch levels transitions properly for stitching. In unity terrain, the quad tree is balanced for this purpose but inorder to support dynamic placement driven featur like patch culling, voxel patch inject, material sets, etc. I opted for this instead. Not only is it faster then balancing the tree ontop of the above mentioned pluss, the more dynamic nature of patch placement in my opion produces nicer results.



What this function does is for each index in the to AdjustedLevels object, (copy of the raw level map ; see CheckForMeshUpdates) it checks to see what the highest neighboring level is around it. (n,s,e,w) and then it raises the level so its of the highest returned level by only one.

This ensures each batch can stitch properly as well as connect with neighboring terrain when that is reintegrated. (Need to rething interface, planets would be interesting).

-

### **BatchInstances**

Used by the terraManager, simply calls; BatchLevels.

-

### **BatchLevels**

-This function groups regions of patches/levels in AdjustedLevels into batches of binary dimensions (1,2,4,8,16, etc) and records the positions, levels as well as the group size to hand to off the updateMeshReflection. See BuildObjectStructures for more more on batching sizes.

-

### **updateMeshReflection**

TerraData