Landscape creation and rendering in REDengine 3

Marcin Gollent
Senior programmer @ CD Projekt RED

GAME DEVELOPERS CONFERENCE® 2014
SAN FRANCISCO, CA
MARCH 17–21, 2014
EXPO DATES: MARCH 19–21

About CD Projekt RED

- Located in Warsaw, Poland
- Established in 2002
- Focused on RPGs
- **The Witcher** (2007, MC:81)
  - PC, heavily modified Aurora Engine
- **The Witcher 2: Assassins of Kings** (2011, MC:88)
  - PC, REDengine 1
- **The Witcher 2: Assassins of Kings Enhanced Edition** (2012, MC:88)
  - PC/X360, REDengine 2

- **The Witcher 3: Wild Hunt**
  - REDengine 3
  - Releasing in February 2015
  - PC/XBO/PS4
- **Cyberpunk 2077** (REDengine 3)
  - Release date TBD

The Witcher 3: Wild Hunt

35x bigger than **The Witcher 2**

- open world
- complex streaming
- a lot of tools refactoring
- different approaches
- DX9 -> DX11

TERRAIN & VEGETATION

Agenda

- How we deal with terrain
  - streaming and LOD
  - texturing
  - shadows
  - memory footprint
- How we distribute vegetation and debris
  - on-the-fly distribution
  - offline vegetation generator
- Stamp tool and workflow summary
- Q&A

Terrain – Our objectives (1)

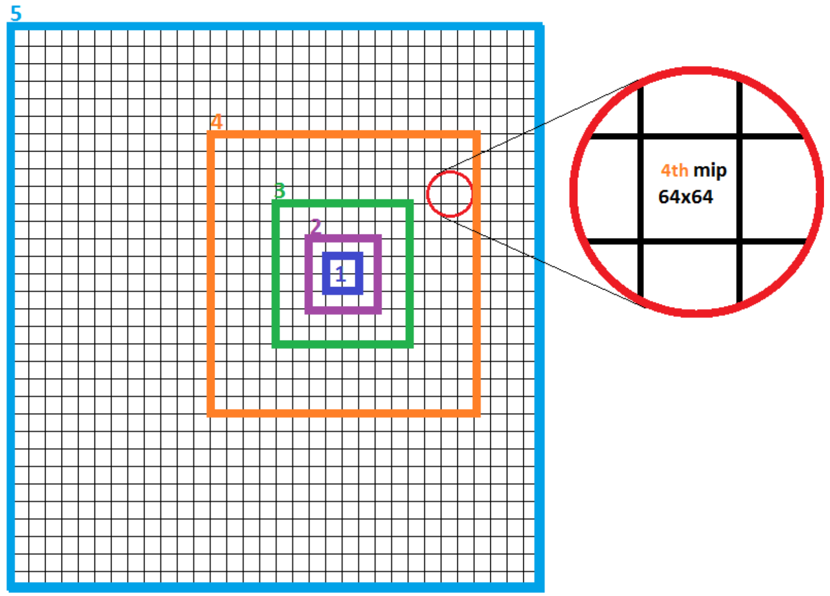
- Support more than 16384^2 resolution maps
- Less than 0.5 meter inter-vertex spacing
- Various landscape characteristics
- Terrain holes for placing cave mesh
- Paint and fill it with a relatively small team
  - Terrain shape generated in World Machine and imported

Terrain – Our objectives (2)

- Big expectations in terms of texturing
  - Nice material blends
  - Slope-based, per-pixel
- Terrain must cast shadows
- Extensive copy-paste functionality over general “landscape”

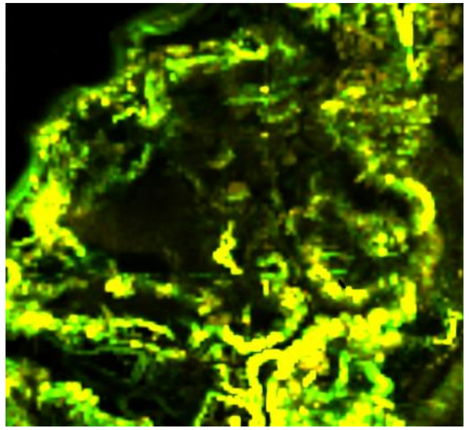
Terrain - Streaming

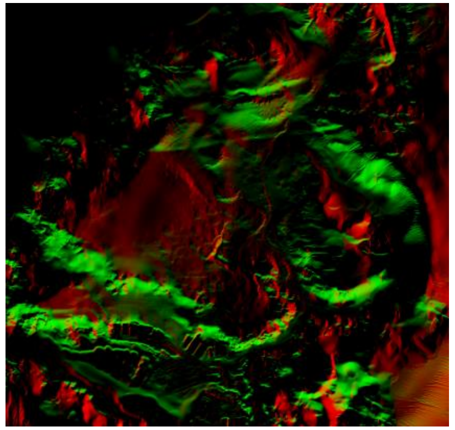
- Clipmap in memory, having regions streamed.
  - texture array
- Working setup for Novigrad:
  - 46x46 tiles, 512x512 each (23552^2)
  - window res = 1024x1024
  - 5 clipmap levels
  - inter-vertex space = ~0.37 cm
  - ~74 km^2



Clipmaps

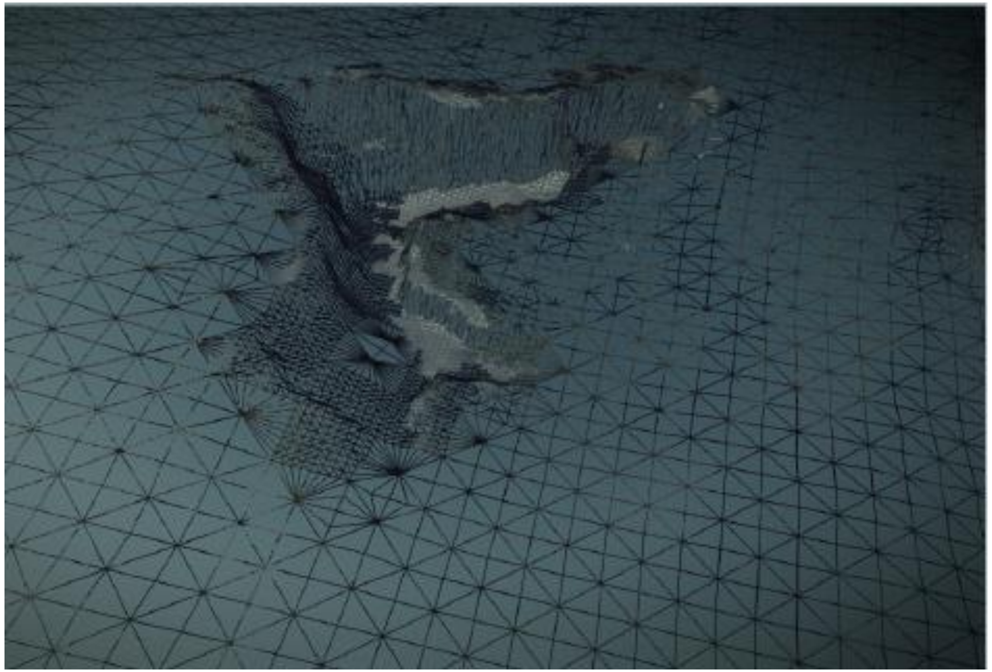
- 3 streamed clipmaps
  - elevation (16 bit unorm)
  - control map (16 bit uint)
  - color (32 bit, reduced resolution - 4096^2 in case of a 16384^2 elevation data set)
- 3 runtime generated clipmaps
  - vertical errors (64x64 common case)
  - normals (optional)
  - terrain shadows





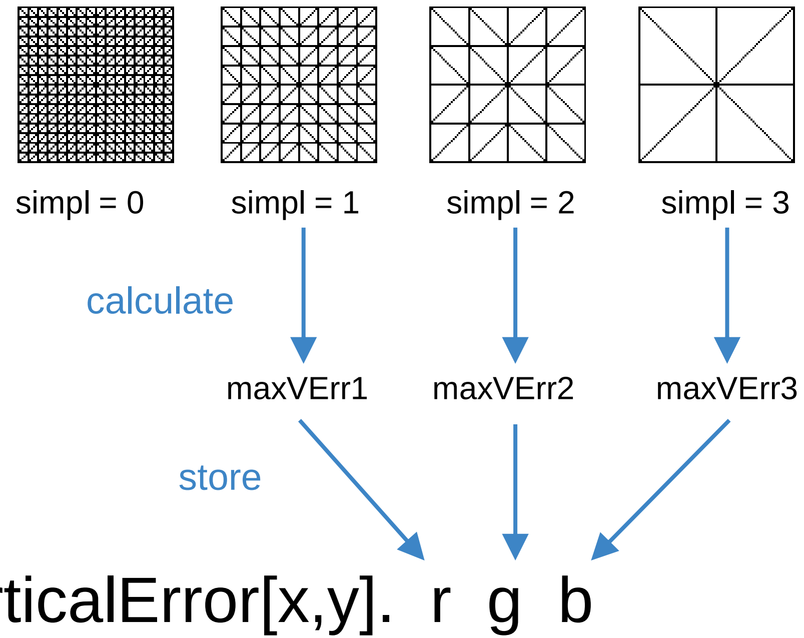
Terrain - Tessellation

- Inspired by a Gpu Pro 3 article, similar technique applied to the clipmap
- Triangle count still very good
- Gives best results with max tess factors of 8 or 16, which is good especially for console GPUs



Vertical error maps generation

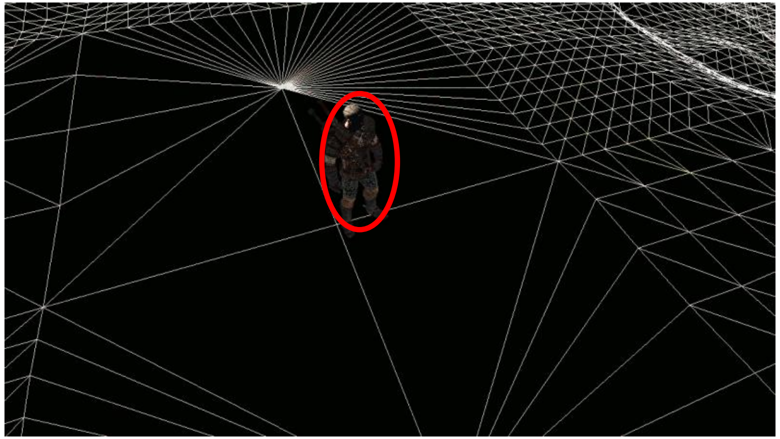
For each tessellation block [x,y]
(1 control point = 1 tess block)

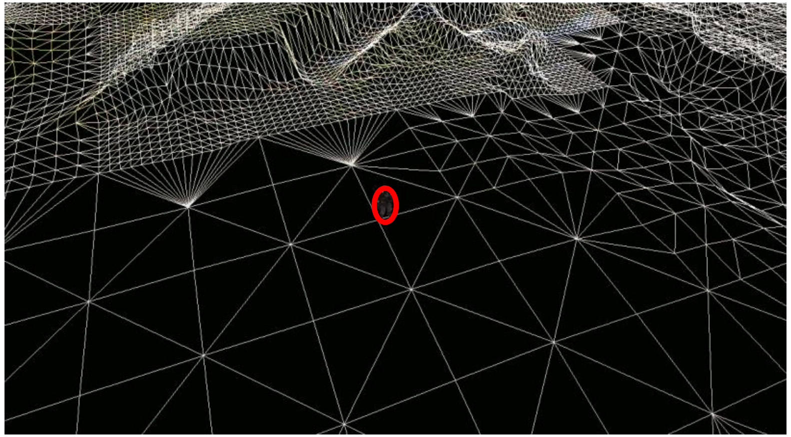


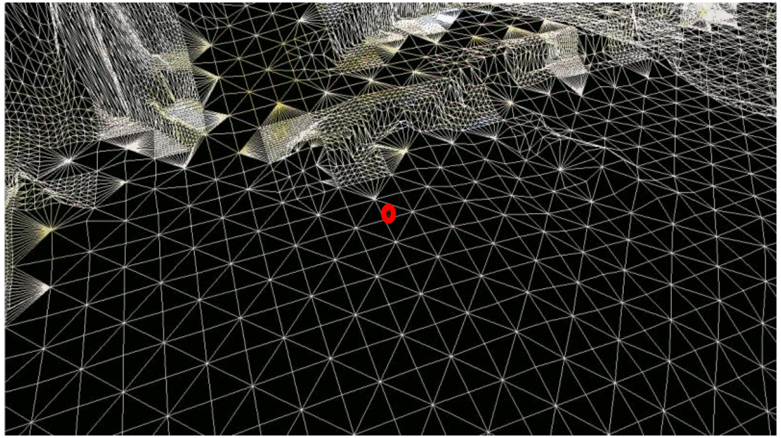
vertical error clipmap window res = elevation clipmap window res / tessellation block res (common case: 64 = 1024 / 16)

Software tessellation

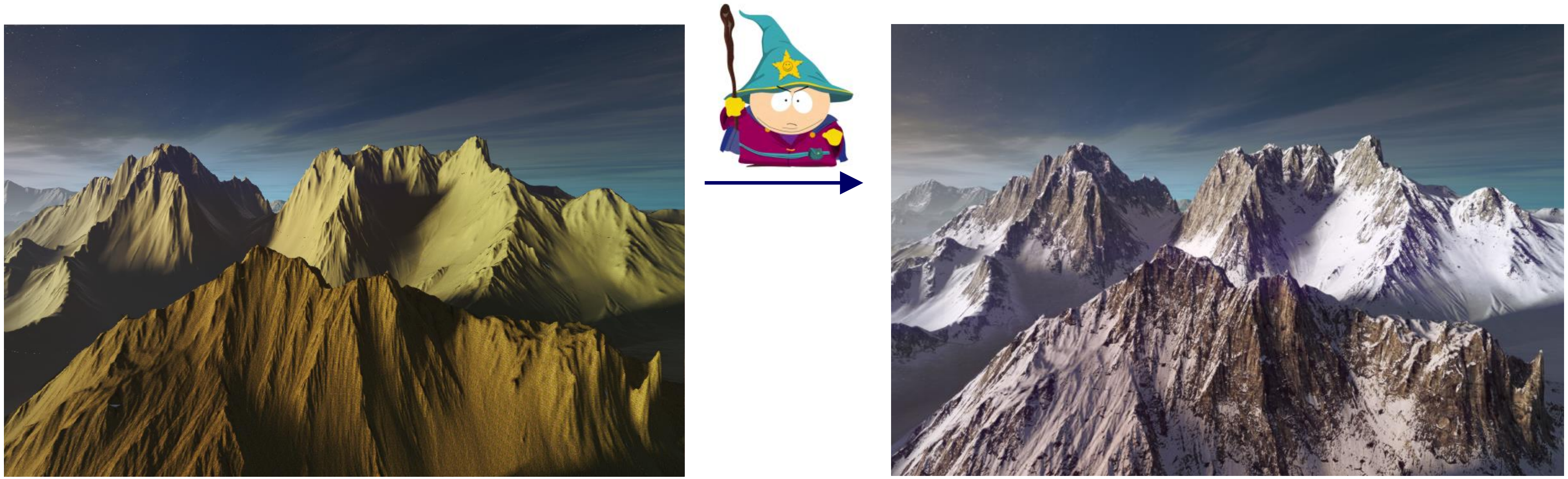
- Downsample the error maps so we can simplify on a quad-tree level prior to relying on hardware tessellation.
- Avoids rendering big areas with a dense grid of minimally tessellated blocks







Texturing goals



WORKFLOW DESIGN ver.1.0

Texturing goals (2)

- Have convincing vistas with virtually no effort
  - start the real work from there
- Have nice material blends in closeup
  - fine to adjust texel size manually
    - with UV-only brush
- Simple to implement
  - No materials streaming
  - Just one texture array
    (two including normal map)

Texturing – Initial idea

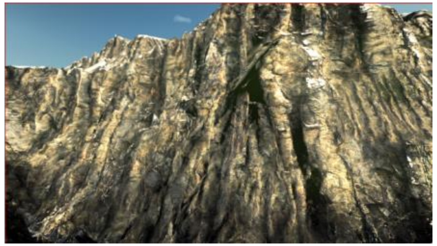
- Define a pair of default materials
  - Background and overlay
  - Set per game level
  - tangent of the background material normal defines visibility of the overlay material
- Typical material layers on top of that
  - controlMapVal <> 0
- Unfortunately - far from achieving all three goals

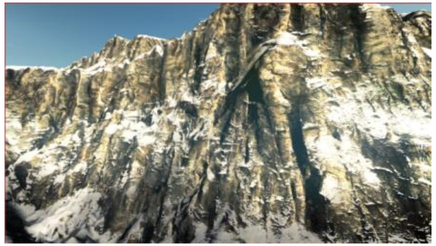
Texturing - Solution

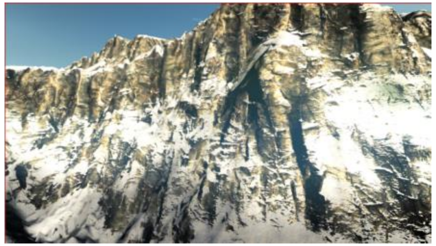
- More radical approach - paint with two freely picked textures
  - Background texture (eg. rock)
  - Overlay texture (eg. snow)
  - Painting feels differently
    - broom - when painting a pathway
    - sowing tool - when placing grass
    - a bucket of snow
    - ...
- 16 bit control-map
  - 0-4 overlay texture index
  - 5-9 background texture index
  - 10-12 UV scale (7 scale values)
  - 13-15 Slope threshold (7 threshold values)

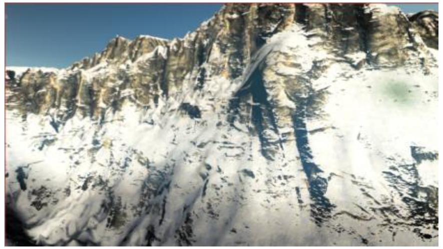
Texturing – Slope threshold

- Each of the 3-bit values assigned for slope threshold is associated with a value
- Values go from 0.0 to 1.0
- Compute slope angle of the background material (world space texture normal)
- Compare against threshold value (control map)


0.125

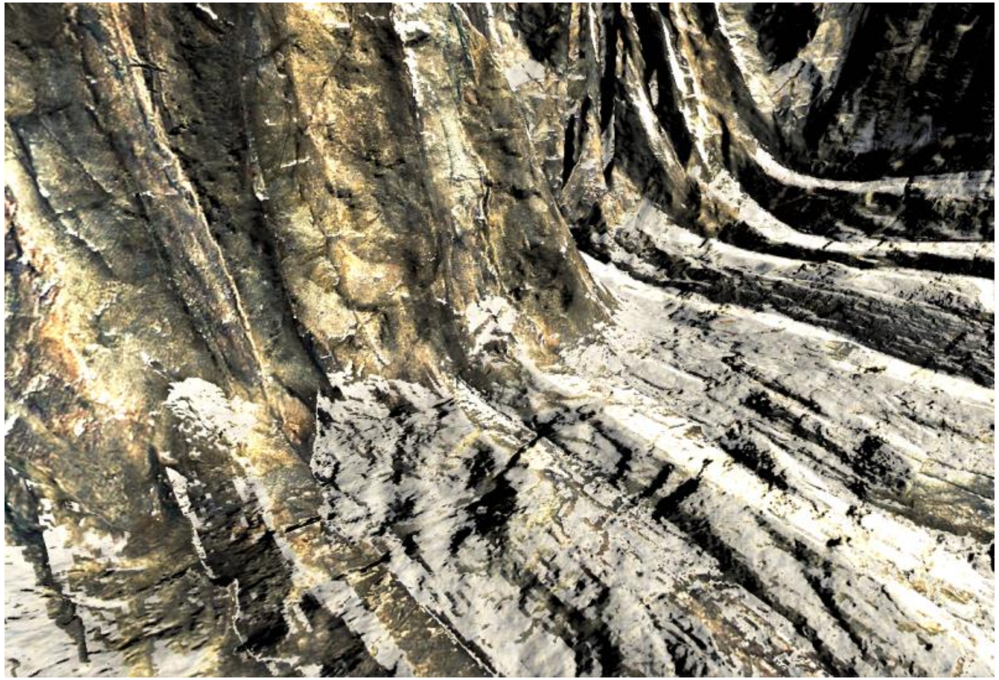

0.375


0.5


0.75

Texturing – Not enough flexibility

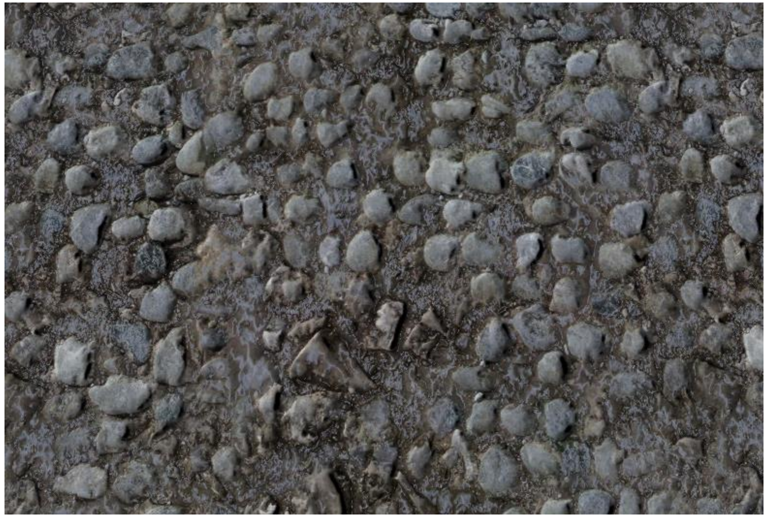
- Issue with slope thresholds logic - problematic to make a thick snow/grass/sand/* cover
- Start with simple ideas…
- Boost overlay texture when vertex normal looks up
- Might be good enough for this case, but ...



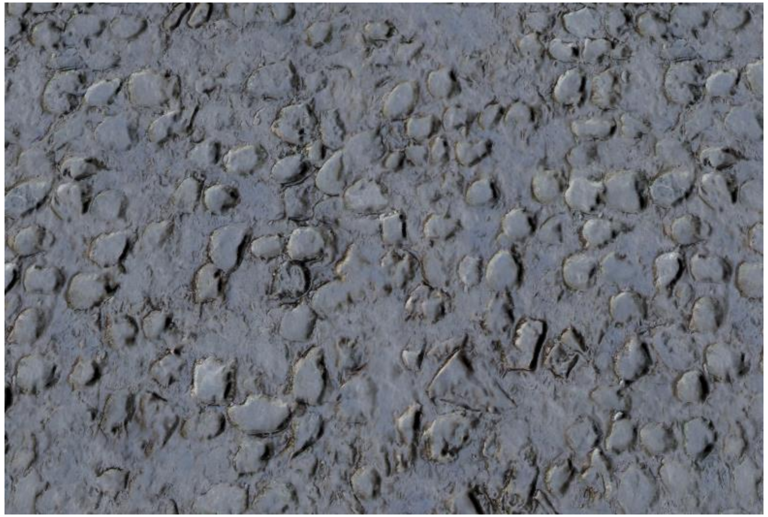
Cobblestones case

- Bad for cobblestones :(
- Smallest threshold value already makes it more than 50% covered
- We want cobblestones to have wide range of protrusion

slope threshold = 0.1 [proper distribution]



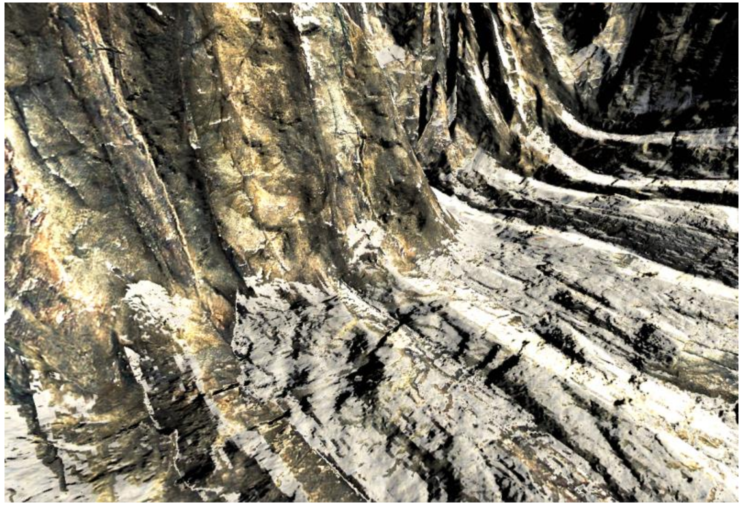
slope threshold = 0.1 with boost [bad distribution]



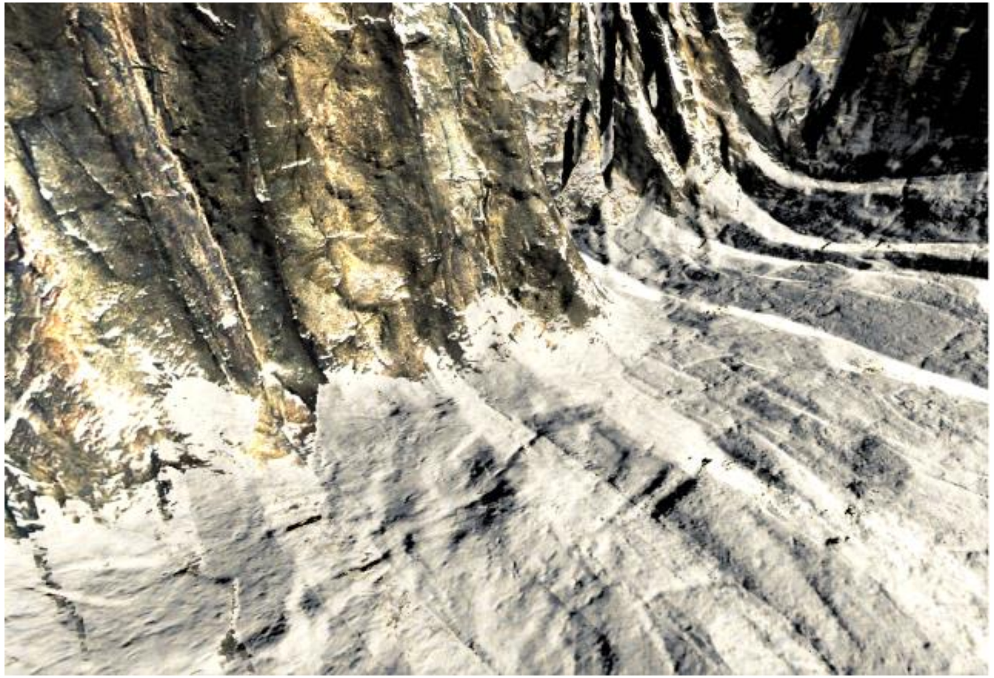
Texturing – Solution

- Provide some parameter that will save the day
  - “slope-based damp”
- We can categorize background materials as:
  - artificial materials (cobblestone, brick, path) that appear mostly on horizontal surfaces
  - natural (crust, rock, etc.) ones that appear widely in nature
- The parameter will be close to 0 for the first case and close to 1 for the second case
- Do it separately for normals

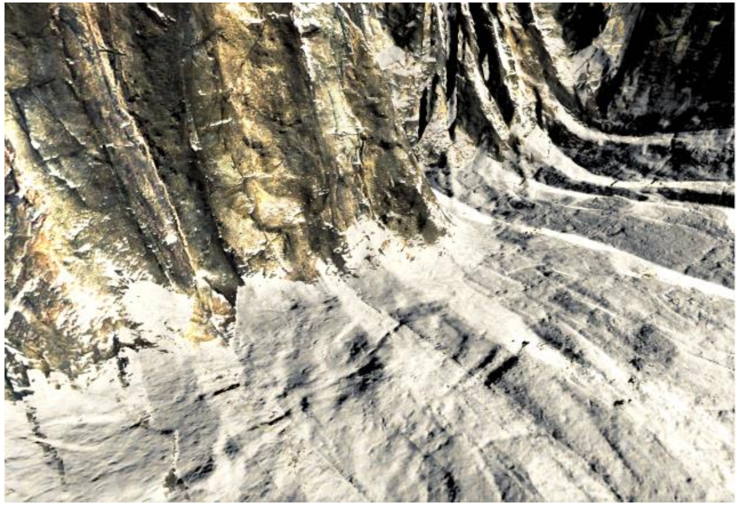
Texturing – Dampening (3)



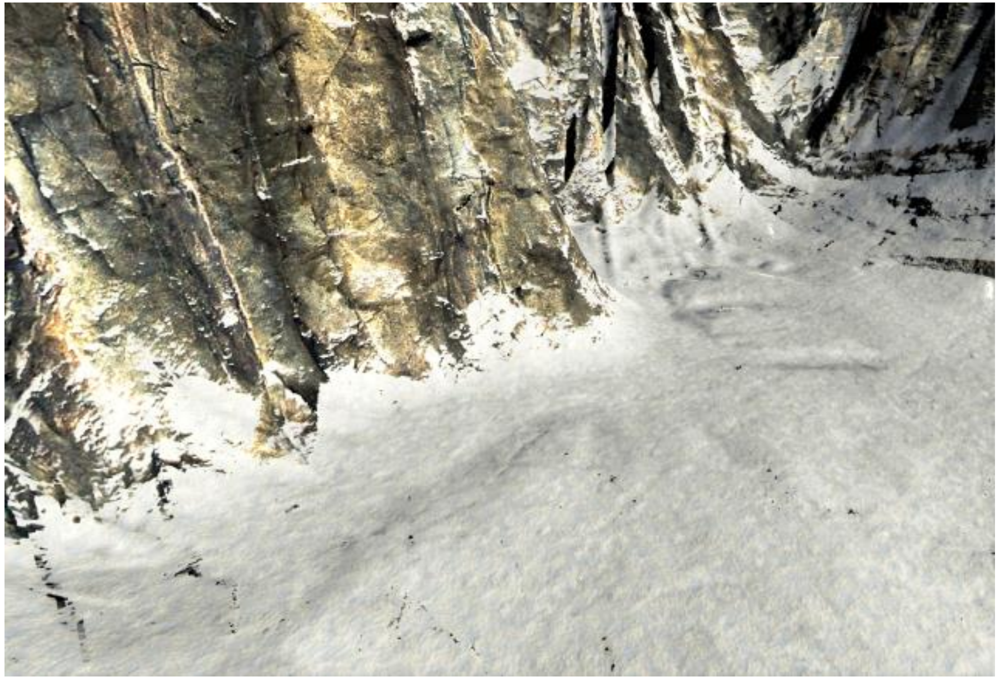
Dampen = 0.5

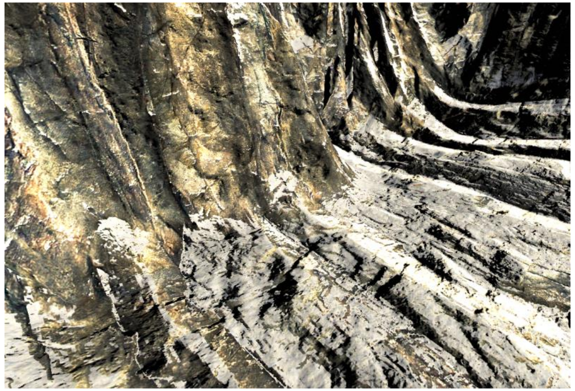


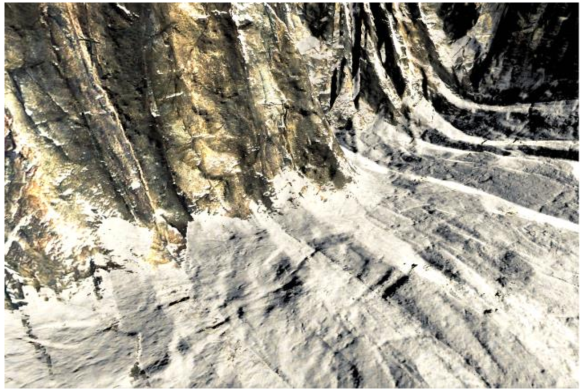
Texturing – Dampening (4)

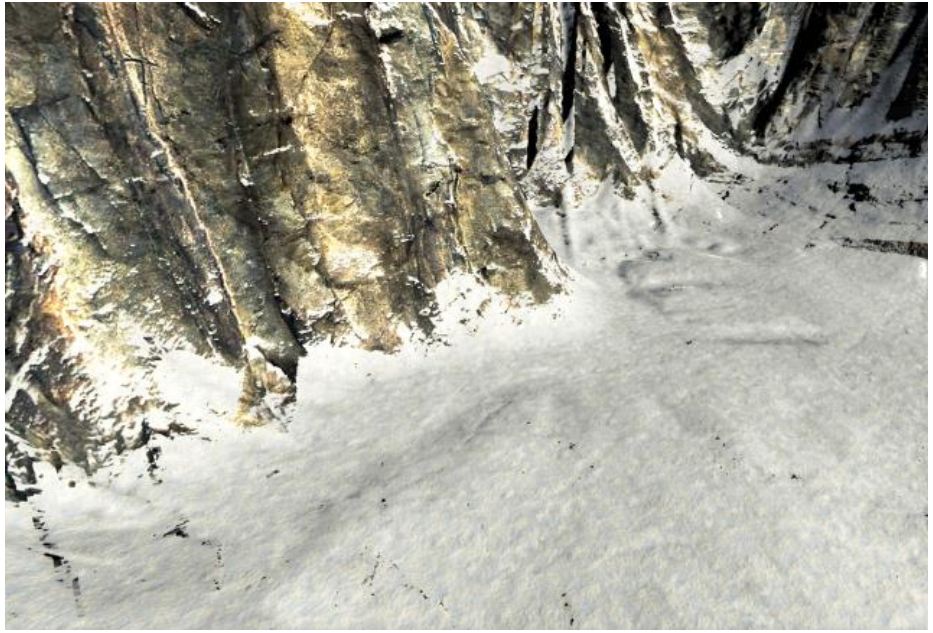


Dampen = 0.5
Normal damp = 0.8



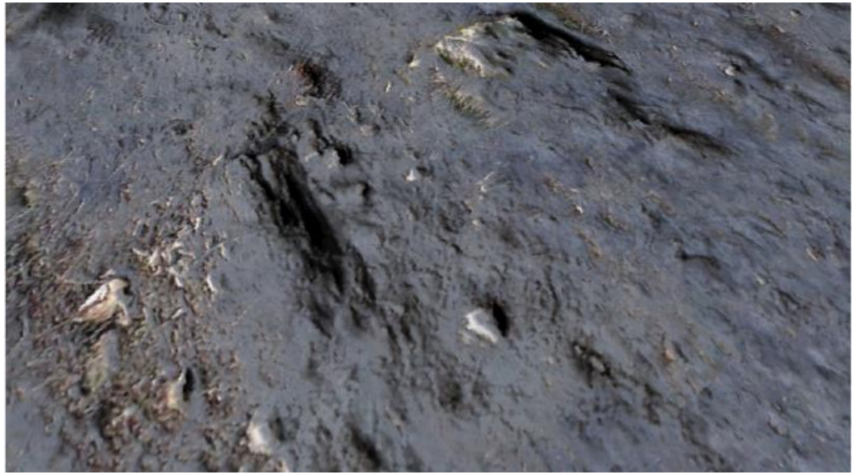

No damp

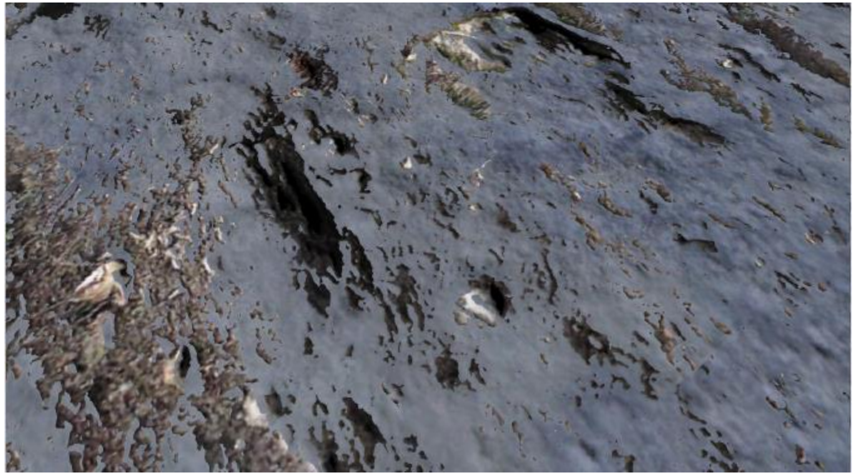

Dampen = 0.5


Dampen = 0.5
Normal damp = 0.8

Blend sharpness

- Different transitions to the background material
  - rather blurry - mud, ice, dirt
  - rather sharp - snow, sand, grass
- Simple to implement - just stretch the distance between some lerp coefficients (see next slide)





Pixel shader – combining and dampening

1. Compute world space normal of a background surface (*combinedVerticalNormal*)
2. Compute world space normal of an overlay surface (*combinedHorizontalNormal*)
3. Measure vertex-level slope angle (*vertexSlope*)
4. Compute a flattened version of pt1. based on *vertexSlope* *(flattenedCombinedVerticalNormal)*
5. Lerp between a pt.1 and pt.4 based on ***slopeBasedDamp*** (*biasedFlattenedCombinedVerticalNormal)*

```
// Combine vertex normal with background / overlay material normals
float3 combinedVerticalNormal = CombineNormalsTangent( vertexNormal, verticalNormal, worldTangent, worldBinormal );
float3 combinedHorizontalNormal = CombineNormalsTangent( vertexNormal, horizontalNormal, worldTangent, worldBinormal );

// Apply dampening to the backround normal
float vertexSlope = dot( vertexNormal, float3( 0, 0, 1 ) );
float3 flattenedCombinedVerticalNormal = lerp( combinedVerticalNormal, float3( 0, 0, 1 ), vertexSlope );
float3 biasedFlattenedCombinedVerticalNormal = normalize( lerp( combinedVerticalNormal, flattenedCombinedVerticalNormal, slopeBasedDamp ) );

// Compute slope tangent for the dampened background normal
float verticalSurfaceTangent = ComputeSlopeTangent( biasedFlattenedCombinedVerticalNormal, slopeThreshold,
                                                    saturate( slopeThreshold + horizontalToVerticalBlendSharpness ) );
float3 fullNormalCombination = CombineNormalsDerivatives( combinedVerticalNormal, combinedHorizontalNormal,
                                                          float3( 1.0f - slopeBasedNormalDamp, slopeBasedNormalDamp, 1.0f ) );

// Use slope to lerp between vertical and horizontal surface colors, normals and material params (spec, gloss, fallof)
float3 finalColor = lerp( horizontalColor, verticalColor, verticalSurfaceTangent );
float3 finalNormal = lerp( fullNormalCombination, combinedVerticalNormal, verticalSurfaceTangent );
```

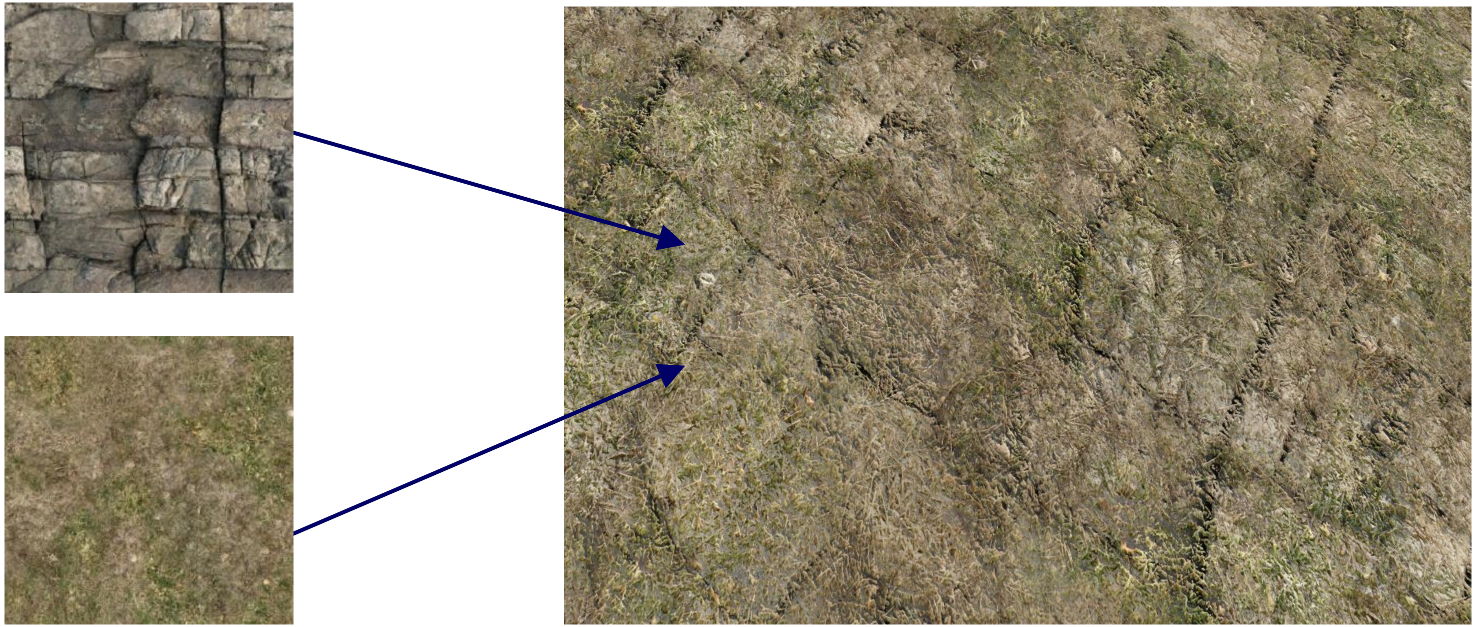
Pixel shader – blending (+ normal dampening)

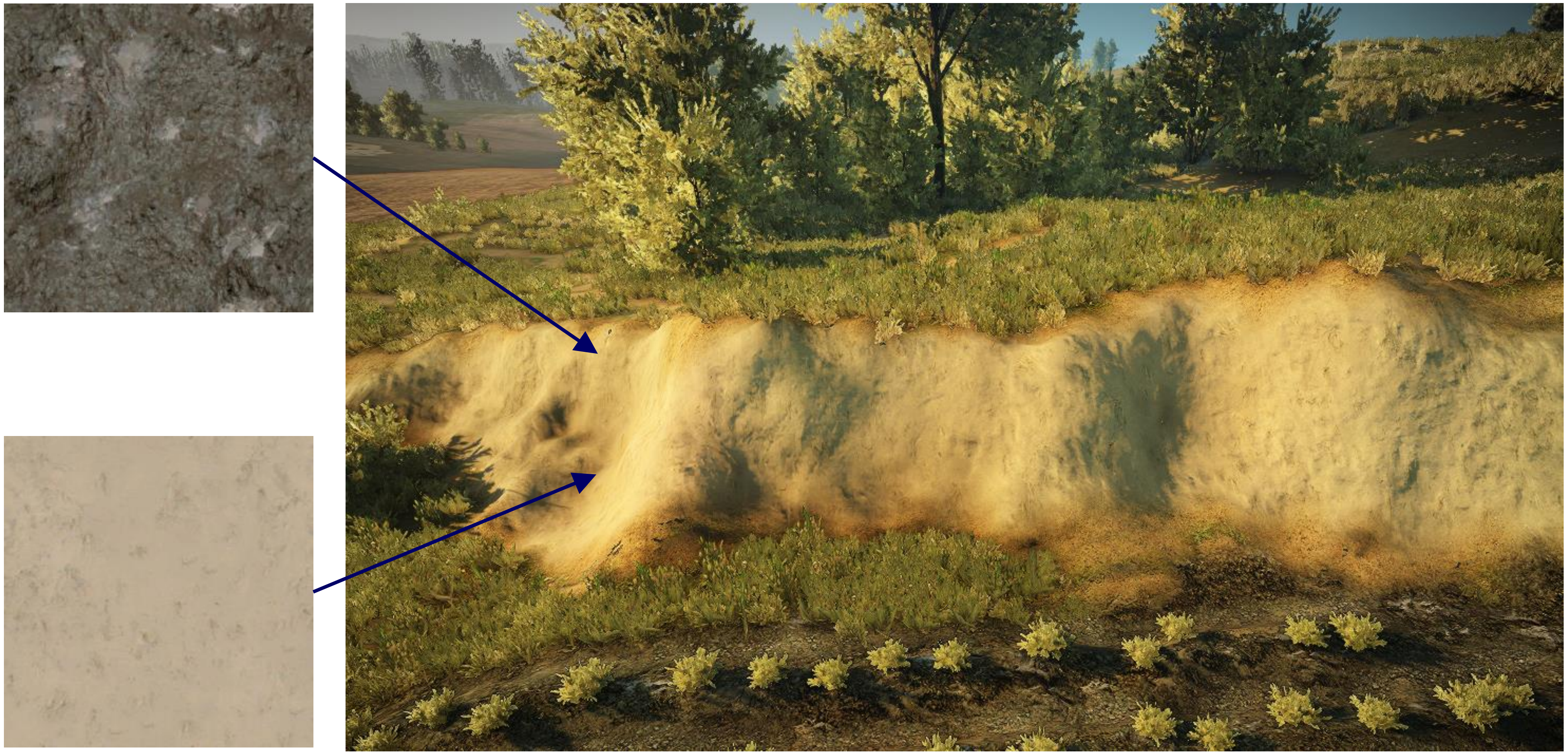
1. Compute tangent of the background surface (*verticalSurfaceTangent*)
   a. linear step between *slopeThreshold* and *(slopeThreshold + blendSharpness)*
2. Combine background and overlay normals through partial derivatives, considering *slopeBasedNormalDamp. (fullNormalCombination)*
3. Lerp colors and normals based on *verticalSurfaceTangent.*

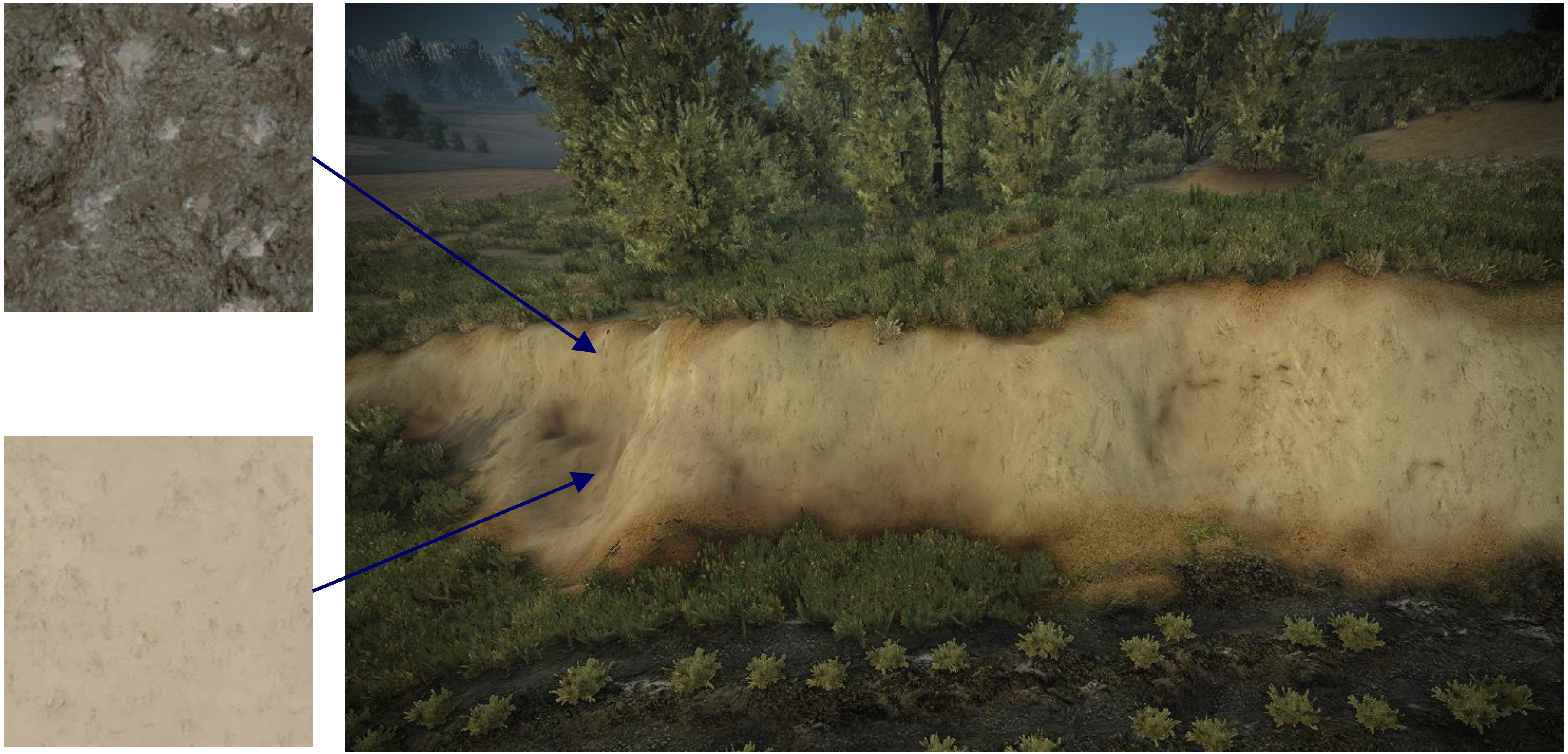
```
// Combine vertex normal with background / overlay material normals
float3 combinedVerticalNormal = CombineNormalsTangent( vertexNormal, verticalNormal, worldTangent, worldBinormal );
float3 combinedHorizontalNormal = CombineNormalsTangent( vertexNormal, horizontalNormal, worldTangent, worldBinormal );

// Apply dampening to the backround normal
float vertexSlope = dot( vertexNormal, float3( 0, 0, 1 ) );
float3 flattenedCombinedVerticalNormal = lerp( combinedVerticalNormal, float3( 0, 0, 1 ), vertexSlope );
float3 biasedFlattenedCombinedVerticalNormal = normalize( lerp( combinedVerticalNormal, flattenedCombinedVerticalNormal, slopeBasedDamp ) );

// Compute slope tangent for the dampened background normal
float verticalSurfaceTangent = ComputeSlopeTangent( biasedFlattenedCombinedVerticalNormal, slopeThreshold,
                                                    saturate( slopeThreshold + horizontalToVerticalBlendSharpness ) );
float3 fullNormalCombination = CombineNormalsDerivatives( combinedVerticalNormal, combinedHorizontalNormal,
                                                          float3( 1.0f - slopeBasedNormalDamp, slopeBasedNormalDamp, 1.0f ) );

// Use slope to lerp between vertical and horizontal surface colors, normals and material params (spec, gloss, fallof)
float3 finalColor = lerp( horizontalColor, verticalColor, verticalSurfaceTangent );
float3 finalNormal = lerp( fullNormalCombination, combinedVerticalNormal, verticalSurfaceTangent );
```

Assembling new textures

- Combining textures with each other
- Different results depending on which texture is a background





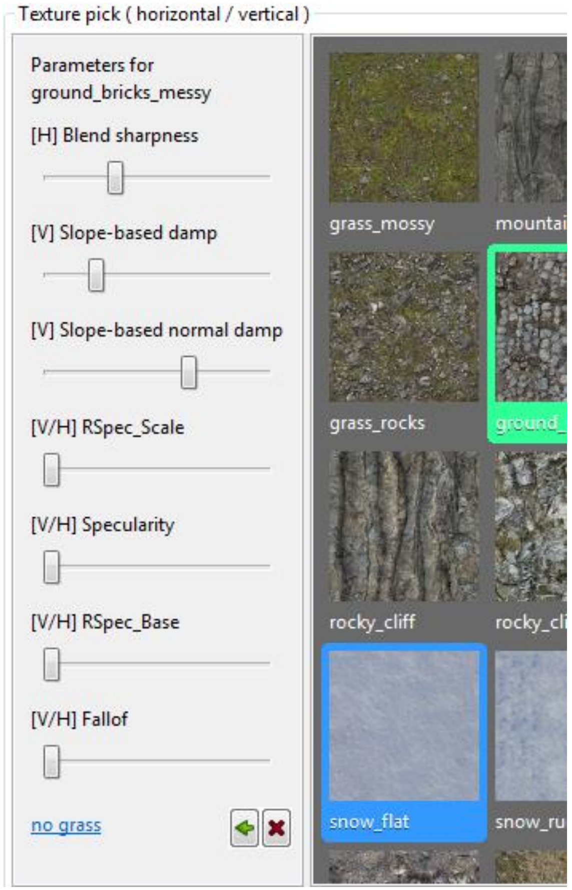


Must have tools

- Freely enable/disable each component of the brush, eg.:
  - want to influence texel size only
  - want to influence threshold only
  - want to change overlay/background texture only
- increment/decrement threshold
- optional but very useful: value falloff on brush radius

Per-texture params

- Horizontal
  - blend sharpness
- Vertical
  - dampen color on low slopes
  - dampen normals on low slopes
- Horizontal/Vertical
  - custom ones, eg.: falloff, roughness
  - grass brush (more about that later)



Texture pick (horizontal / vertical)

Parameters for ground_bricks_messy

[H] Blend sharpness

[V] Slope-based damp

[V] Slope-based normal damp

[V/H] RSpec_Scale

[V/H] Specularity

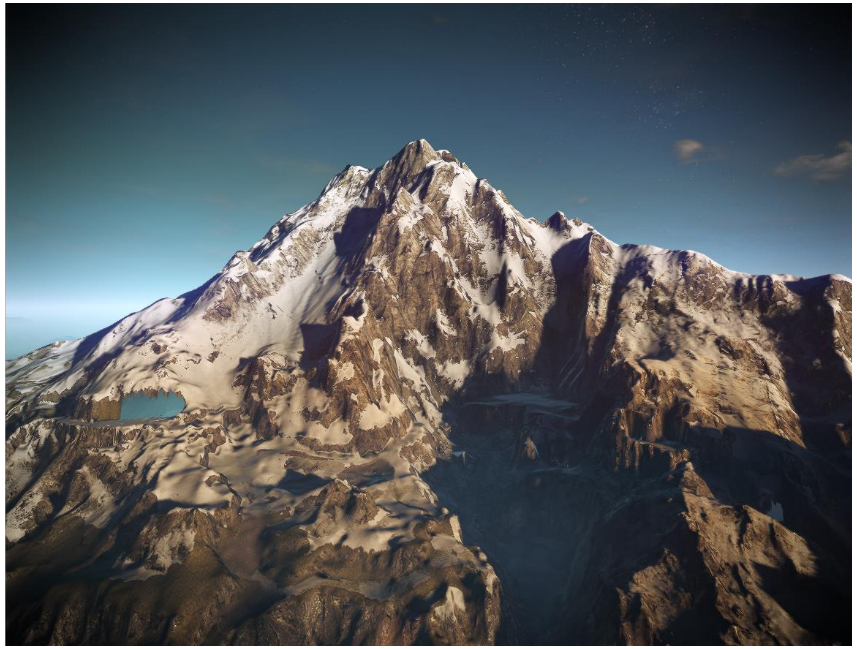
[V/H] RSpec_Base

[V/H] Fallof

no grass

Triplar mapping - performance

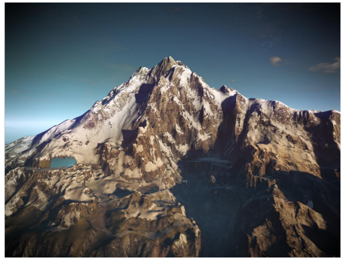
- No triplanar mapping for overlay texture
- For background texture sample with triplanar mapping
    - Choose which planes contribute (prefer branching over texture fetches)
- Tighten the blend zone as much as possible without introducing glitches

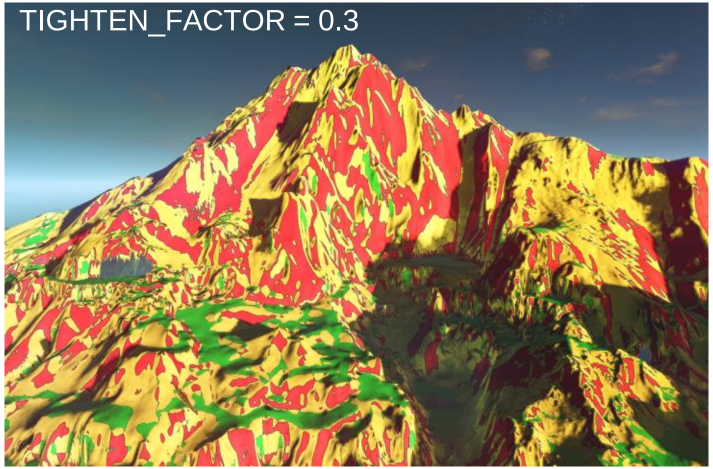


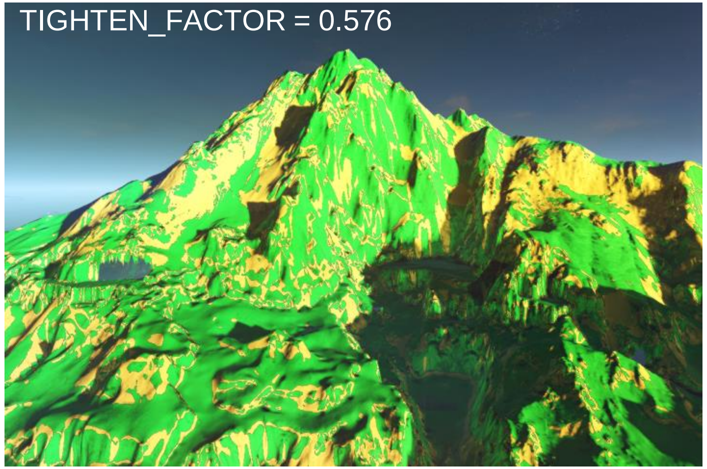
Blend zone tightening

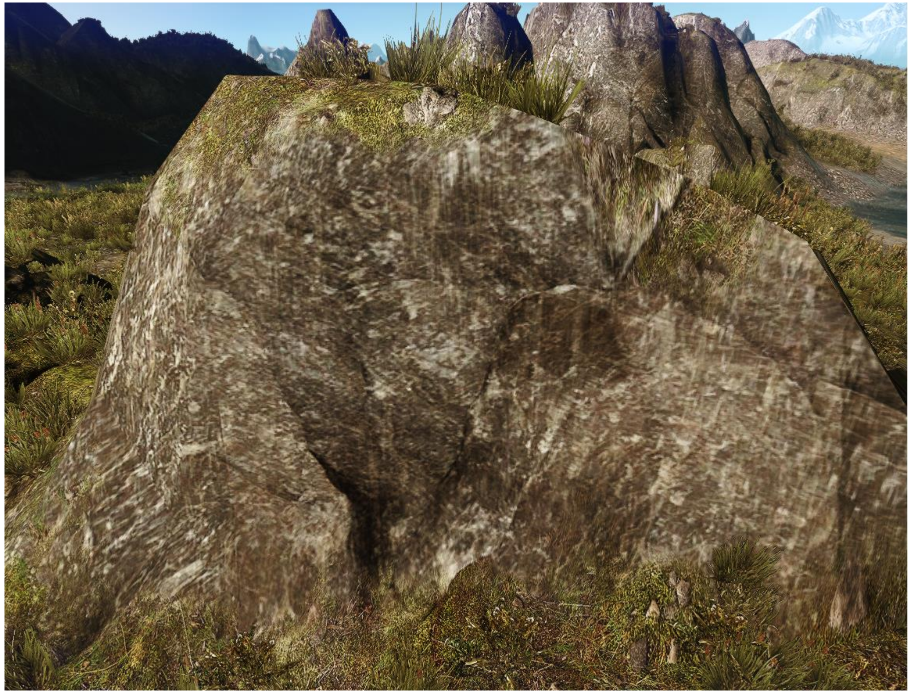
```
// Compute triplanar mapping weights
const float3 tighten = (float3)TIGHTEN_FACTOR;
const float3 absVertexNormal = abs( normalize( vertexNormal ) );
float3 weights = absVertexNormal - tighten;
```

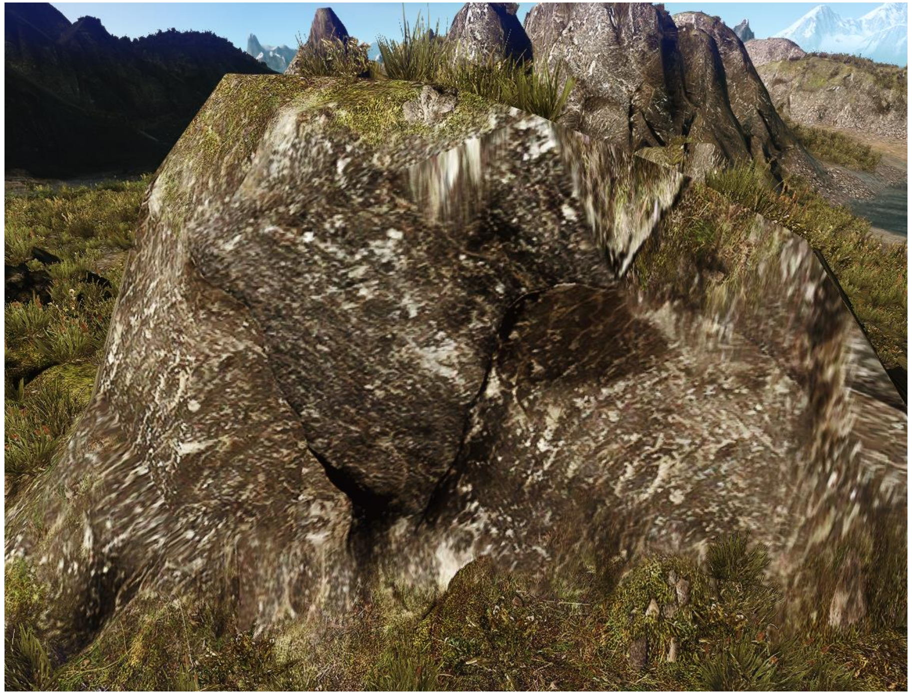
RED - three samples YELLOW - two samples GREEN - one sample





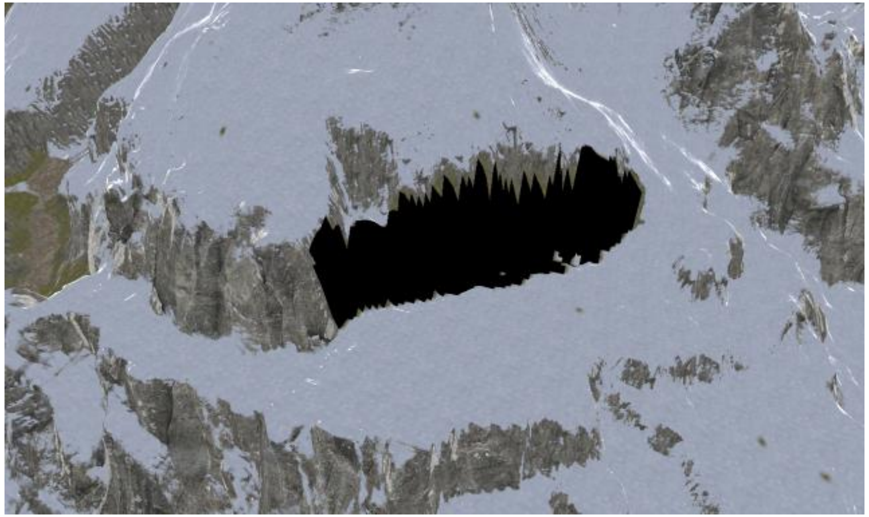



TIGHTEN_FACTOR=0.3


TIGHTEN_FACTOR=0.576

Terrain holes performance

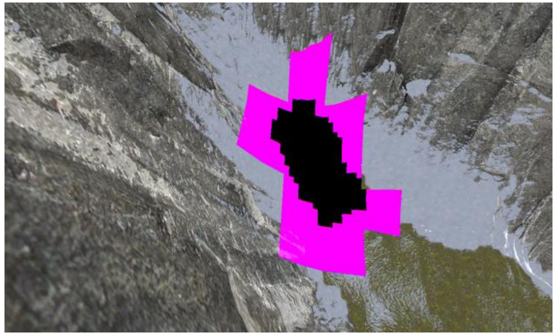
- Solutions unavailable:
  - Can’t just render terrain with “discard” instruction inside, as it disables early-Z and hi-Z.
  - Can’t manipulate indices when using hardware tessellation

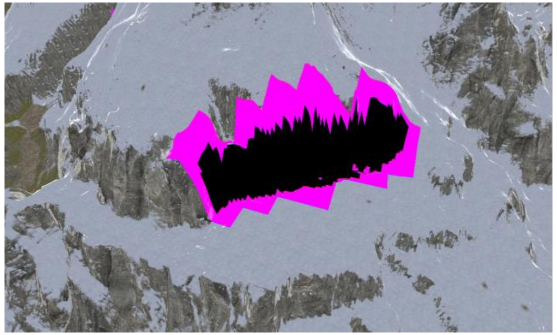


Terrain holes - our solution

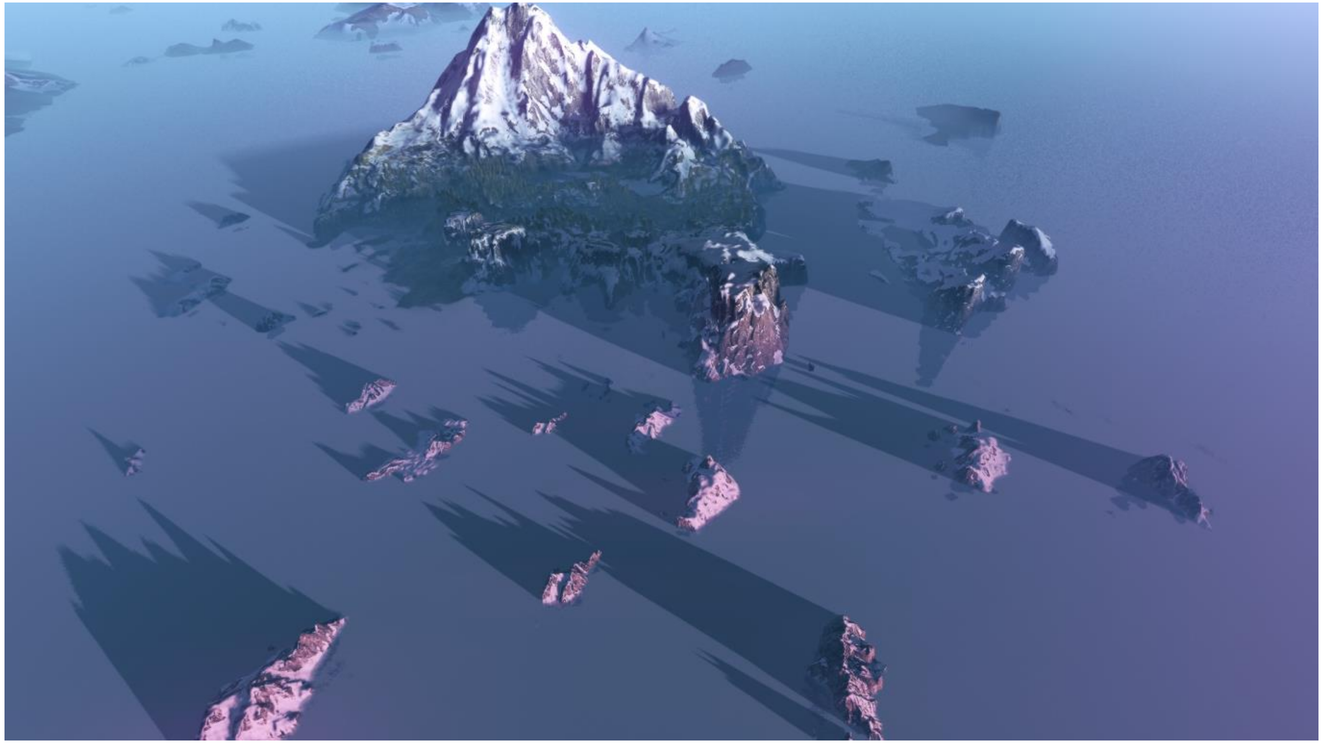
- Know that there is a hole inside a given tessellation block
  - Determine that during vertical error clipmap update
  - Check that during hull shader processing
- Split terrain drawcall into two drawcalls (#ifdef'd shaders)
  - 1: cull tessellation block if it contains a hole
    - usually covers over 99% of the terrain
    - no **discard** instruction - hiZ on
  - 2: draw tessellation block if it contains a hole
    - with **discard** instruction
    - best/closest clipmap level only

```
    // Assume that the stamp is going to need discards.
    // MS_DISCARD_PASS is only used for level 0 patches
#ifdef MS_DISCARD_PASS
    if ( patchCenterErrors.w == 0.0f && !isInStamp )
#else
    if ( patchCenterErrors.w > 0.0f || ( isInStamp && clipmapLevel == 0 ) )
#endif
    {
        leftTessFactor = rightTessFactor = bottomTessFactor = topTessFactor = 0;
    }
```



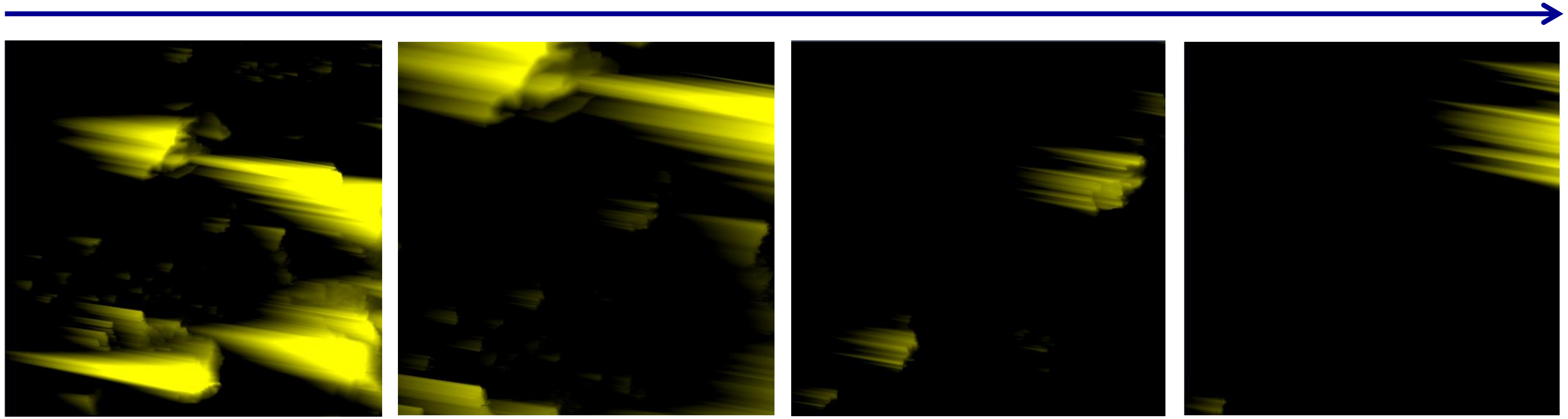


Terrain shadows



Terrain shadows clipmap

- Store maximum height that's in shadow
- Update when streaming clipmap or changing time of day
- Has to be tightly coupled with clipmap computations to avoid shadows shimmering
- Allow a few huge meshes to cast terrain shadows



Terrain shadows algorithm

1. Lay down terrain depth - sun perspective
2. Render to the slices of shadows clipmap

   For each texel
   - At a corresponding elevation texel, calculate a full world space position (*wsPos*)
   - For i=0 to n   // n=13 works just fine ☺
     - transform *wsPos* to the sun-space position
     - compare z value of the sun-space position with the one fetched from pt.1 texture
     - if position is occluded, increase *wsPos.z* by *step,* halve *step*
     - if position is not occluded, decrease *wsPos.z* by *step*
   - Store the last z component value in the terrain shadow map

Terrain – memory footprint

5 clipmap levels, 1024x1024 window size

- elevation + normal + control map = 30 MB
- color map = 12 MB
- vertical errors map = 327 KB
- shadow map = 10 MB
  +10 for meshes casting terrain shadows

= ~53 (63) MB

+ Texture Arrays

Vegetation - overview

- Using SpeedTree Forest library for culling and LODing
  - Shared culling grid for all tree types
  - Separate culling grid per grass type (grass layer)
    - various cell sizes and draw distances
- Track grass visibility cells as they come in and out of view
  - Populate newly visible cells in two steps:
    - pass prebaked instances
    - generate and append dynamic ones
  - Update instance buffers
- Introduce a **vegetation brush** resource
  - wraps a set of vegetation types along with their densities and scales/scale vars.

Grass/debris - goals

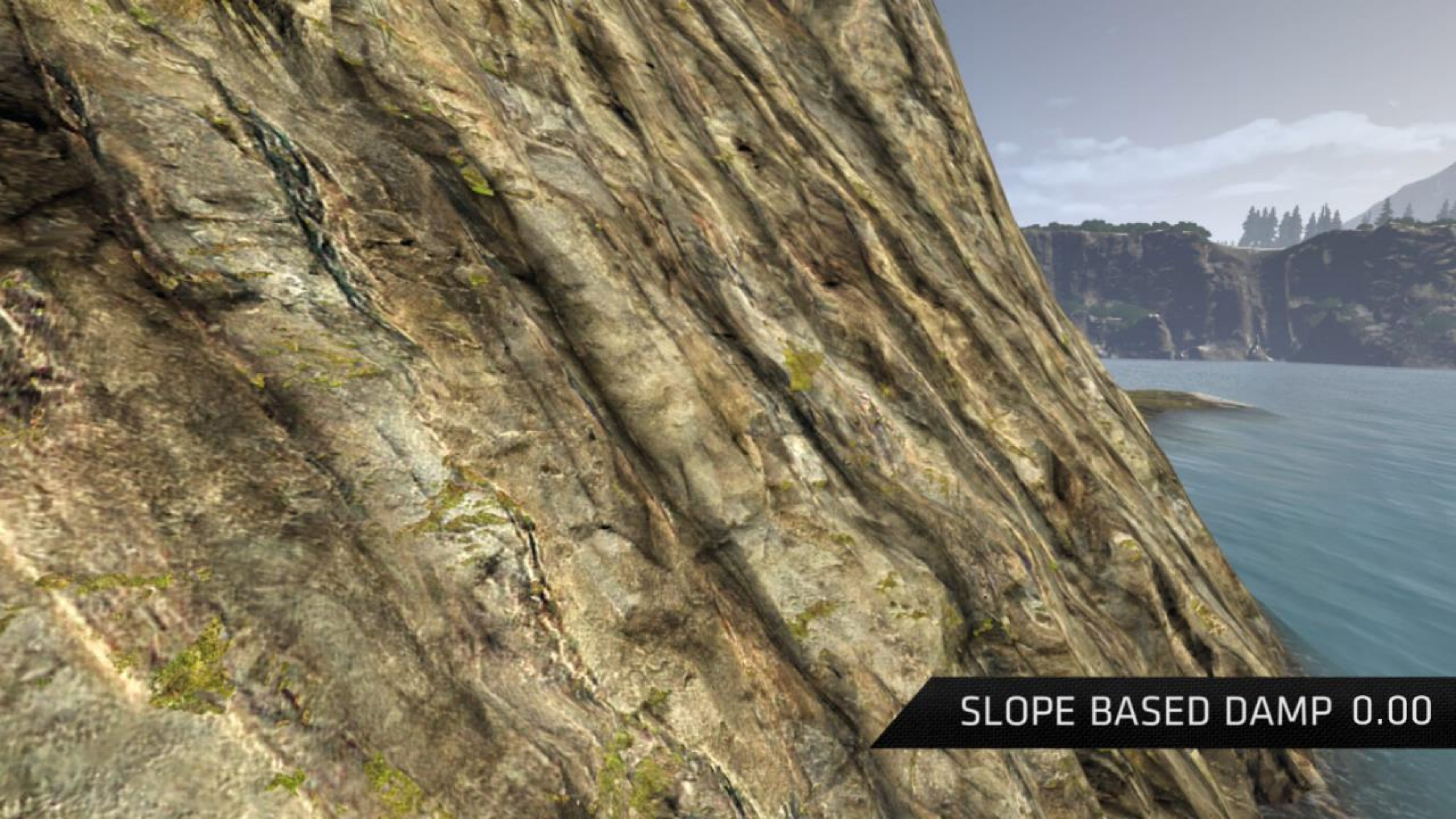
- Distribute procedurally because
  - … instances take a lot of memory
  - … painting takes a lot of time
- Match terrain material
- Make it look diverse
- Allow for masking out
- Make it fast
  - CPU or GPU?

Grass/debris – distribution overview (1)

- Assign brushes to terrain materials
  - grass types from those brushes become auto-distributed
  - many to many relation
  - ~10 types are auto-distributed on our levels
- When a grass cell comes into view, lookup the prebaked **occurrence map**
  - each auto-distributed grass type has one
  - if bit is not set -> skip the cell processing
  - 125kB per map given 100 m^2 cell size on a $10km^2$ world
  - Skips almost all the work compared to the uncooked game

Grass/debris – distribution overview (2)

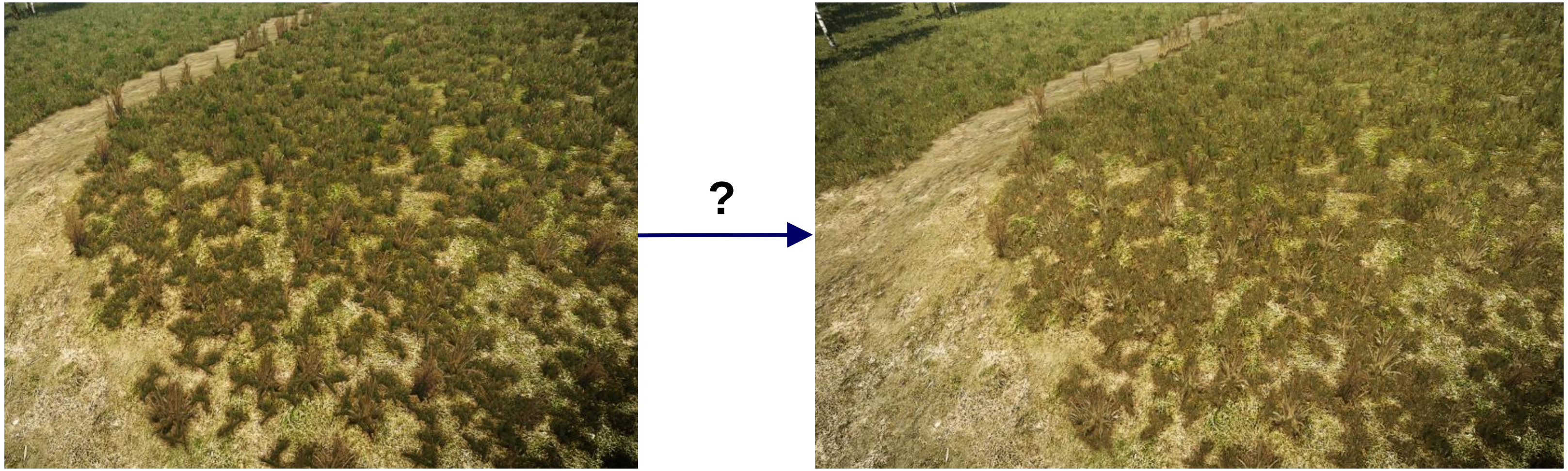
- For a given cell, perform a number of iterations
  - pick a spot within cell's bounds
  - verify if the type is assigned there
  - compute background/overlay visibility for that spot
    - replicates terrain PS behaviour, but with vertex-level precision
  - consume the amount of "attempts" based on brush settings
- Optimize by rendering a grass map after a clipmap update
  - fetch instead of the normal map and avoid most of the computation



SLOPE BASED DAMP 0.00

Grass doesn’t „sit” on the terrain

- Conventional problem of grass standing out visually
  - Takes a lot of artist’s time to hide it
  - Critical when populating procedurally

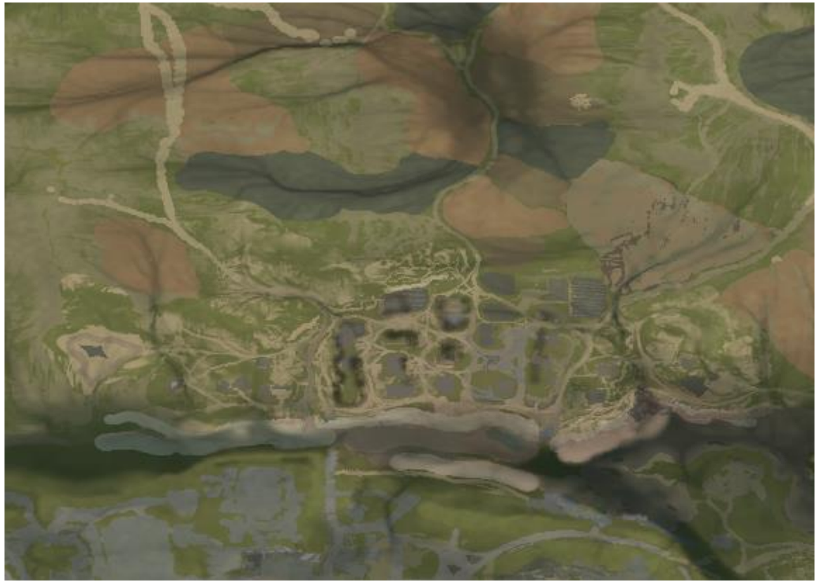


what we have

what we want

Pigment map

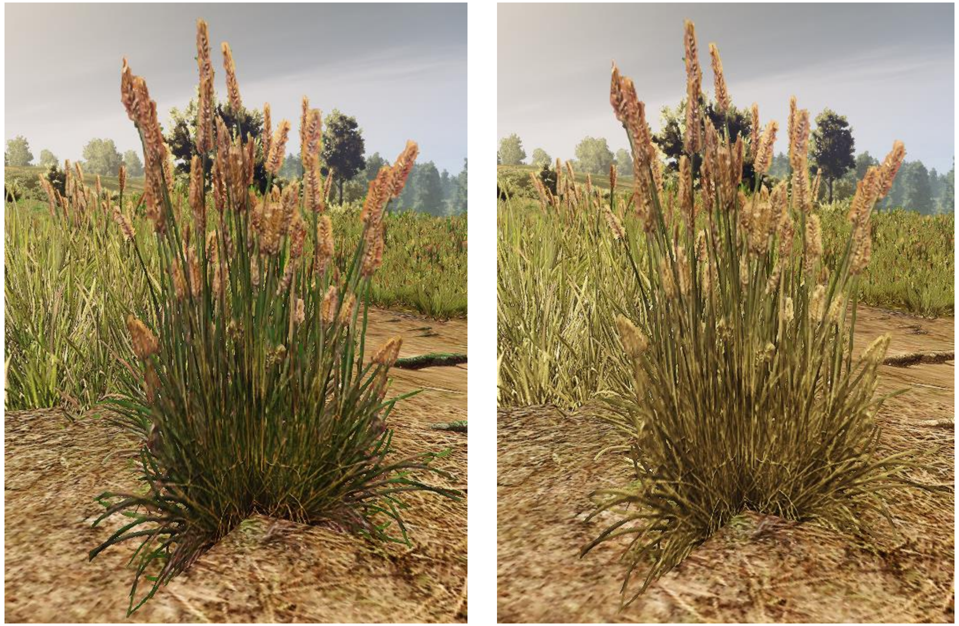
- Render top-down view of the terrain
  - 1’st clipmap level (closest area)
  - Use lowest mipmaps
  - No interpolation
  - No triplanar mapping
  - Account for color map
  - No tessellation - just one huge quad
    - provided that pregenerated normal maps are enabled

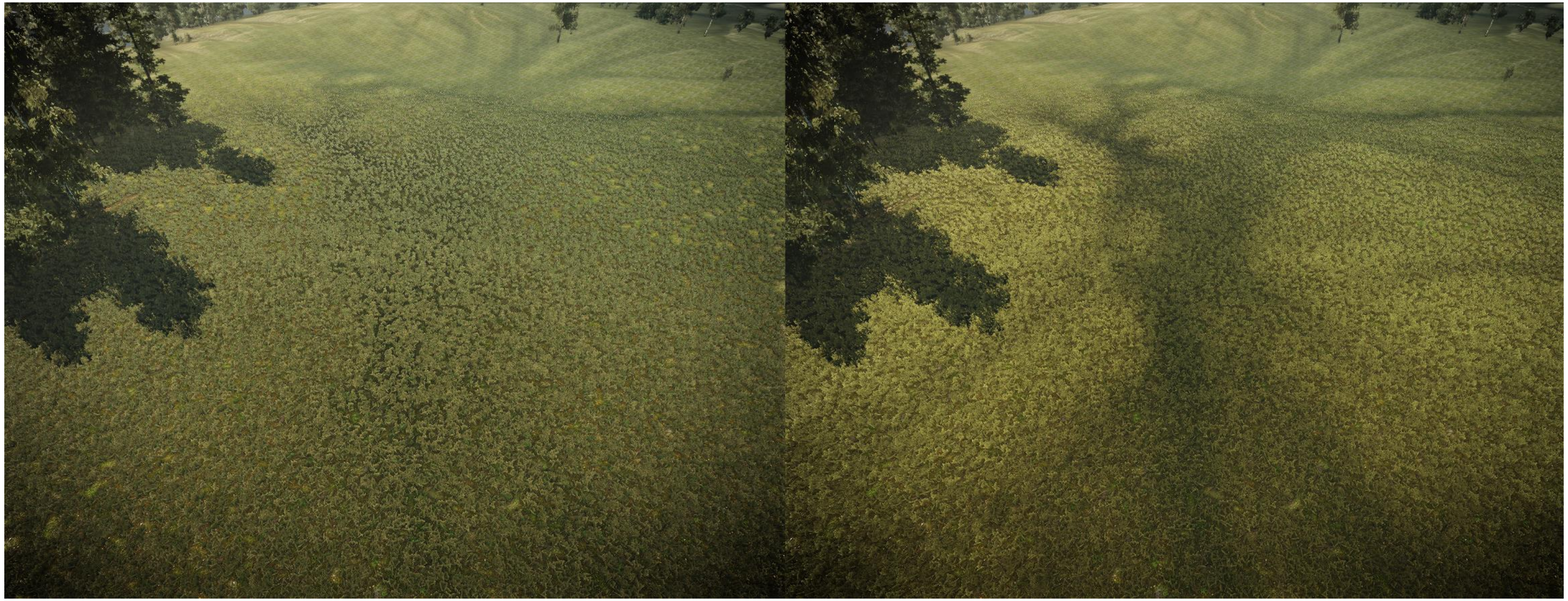


Pigment map (2)

Colorize instances by sampling pigment map in vertex shader

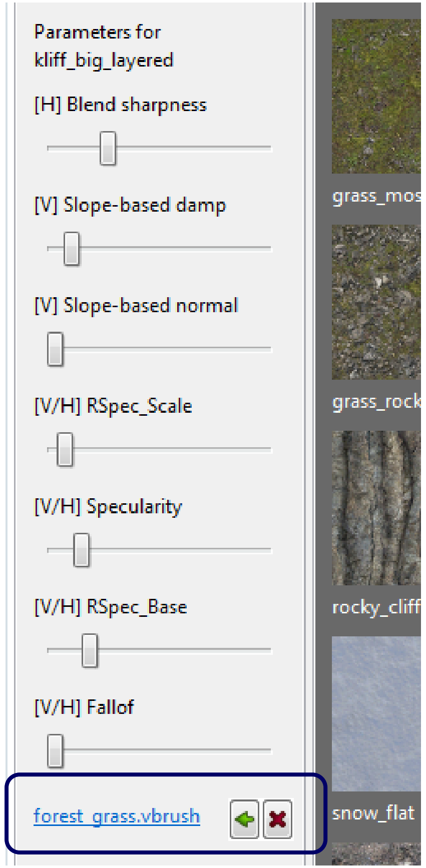
- Bottom-up falloff, adjustable per type
- Slightly less for close instances
- Slightly more for far instances
- Exclude some types (eg.: heather)





Grass/debris – tool

- Assign a brush to the material
- Ignore non-grass types
- Account for density, scale, scale var.



Parameters for kliff_big_layered

[H] Blend sharpness

[V] Slope-based damp

[V] Slope-based normal

[V/H] RSpec_Scale

[V/H] Specularity

[V/H] RSpec_Base

[V/H] Fallof

forest_grass.vbrush

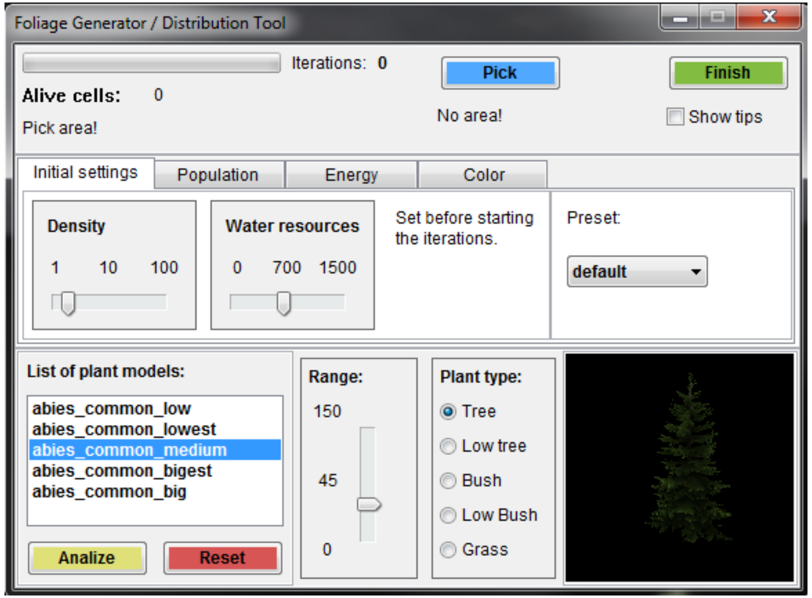
Grass/debris – reminder

- The technique doesn’t exceed the cost of 1ms when running in a cooked game
- Hits the editor performance badly when going overboard with the amount of auto-distributed grass types
- Users must be able to mask some areas out
  - currently we have a separate mask for that
  - will use occurrence map in a cooked game
- A thing to try: move the distribution to the GPU
  - eg.: run it on a compute shader with limited resources, while processing draw calls

Vegetation generator

- Offline tool, not limited to grass and debris
- Can populate the whole level or chosen contiguous areas
- Simulates water accumulation and light distribution

Vegetation generator tool



Foliage Generator / Distribution Tool

Iterations: 0

Pick

Finish

Alive cells: 0

No area!

Show tips

Pick area!

Initial settings | Population | Energy | Color

Density		
1	10	100

Water resources		
0	700	1500

Set before starting the iterations.

Preset: default

List of plant models:

- abies_common_low
- abies_common_lowest
- abies_common_medium
- abies_common_bigest
- abies_common_big

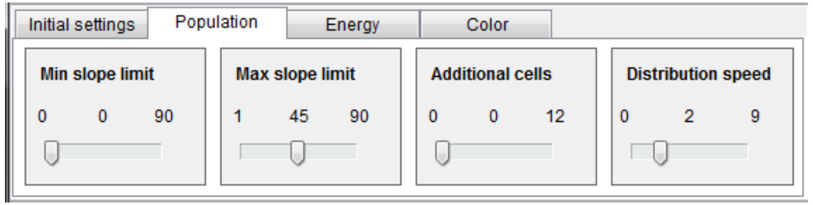
Analize

Reset

Range: 150 45 0

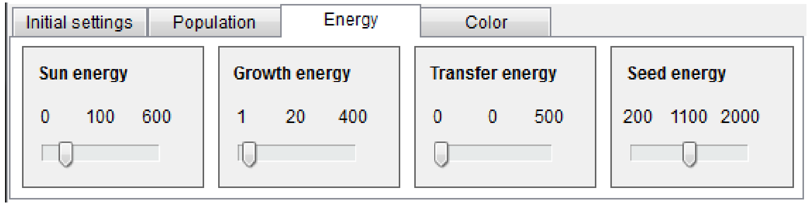
Plant type:

- [✓ Tree]
- Low tree
- Bush
- Low Bush
- Grass



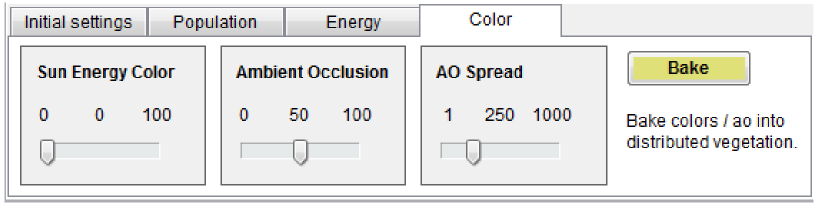
Initial settings | Population | Energy | Color

	Min	Value	Max
Min slope limit	0	0	90
Max slope limit	1	45	90
Additional cells	0	0	12
Distribution speed	0	2	9



Initial settings | Population | Energy | Color

	Min	Value	Max
Sun energy	0	100	600
Growth energy	1	20	400
Transfer energy	0	0	500
Seed energy	200	1100	2000



Initial settings | Population | Energy | Color

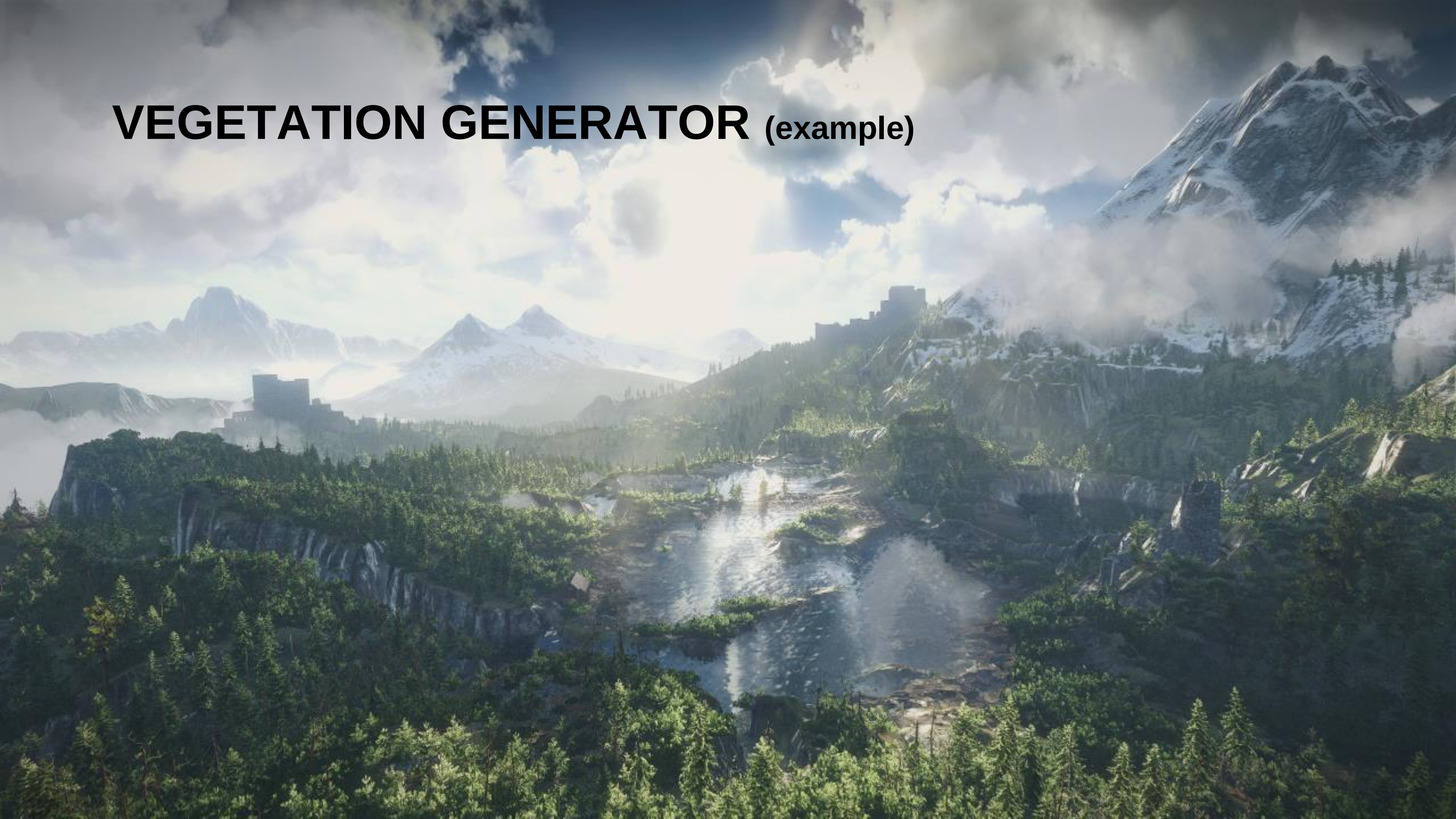
	Min	Value	Max
Sun Energy Color	0	0	100
Ambient Occlusion	0	50	100
AO Spread	1	250	1000

Bake

Bake colors / ao into distributed vegetation.

Vegetation generator algorithm

- For the whole region, allocate a 2d array of “resource” values.
- Perform a number of following iterations:
  - For each cell, check the cell and it’s neighbours elevations
  - Compute the slopes to/from neighbours, and decrease/increase resource values (sedimentation of resources)
- For all cells
  - Check the trajectory of the sun and compute the percentage of time when the cell is in shadow. Compare this value with values assigned to vegetation types to filter them out (for that cell)
  - If the cell’s resource value is bigger than some threshold, then pick a vegetation type, and spawn it.
    - Additionally scale the object based on how it’s environment matches “perfect conditions”, namely the amount of resource and sunlight



VEGETATION GENERATOR (example)

Stamp tool

- Photoshop-style
- Copy-paste with full package
  - elevation, control map, color map, vegetation
  - rotating, scaling, paste-combining
  - real-time preview
- Allow saving/loading stamps
- Became the most popular tool!

Workflow

- Import the terrain from World Machine.
- Select any area to export, process, and import back.
  - Done very rarely since stamp tool reached polished stage.
- Edit mostly with stamps.
  - Artists move away from rise/lower tool
    - They keep a set of stamps, with different landscape characteristics.
    - It’s critical to make this tool robust, ergonomic, real-time, have undo for it, etc. but it pays off.
    - Falling back to rise/lower for quest fine-tuning purposes mostly.
- Painting grass manually only in selected areas
  - some intensive action/cutscene/dialogue zone
  - when applying knowledge about relations between foliage types for a more convincing result
- Run vegetation generator iterations for chosen areas.
  - Settings are easy to reapply for given area, so redesigning some area is not a big problem.
- When in need to manually place grass and debris, mask out the automatic one.

Thanks to the team!! (in alphabetical order)

PROGRAMMERS
Adam Cichocki (Pigment map)
Przemek Czatrowski (Color map, Stamp preview, tools)
Tim Green (Stamp, tools)
Tomasz Jonarski (Shadow map)
Krzysztof Krzyścin (Vegetation Generator)
Konstantinos Michalopoulos (tools)

ARTISTS
Michał Buczkowski
Marcin Michalski
Daniel Olejnik

References

- Egor Yusov, "Real-Time deformable Terrain Rendering with DirectX 11" , Gpu Pro 3
- Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones “The Clipmap: A Virtual Mipmap”, Silicon Graphics Computer Systems
- Frank Losasso, Hugues Hoppe, „Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids”
- Nicolai de Haan Brøgger, „Real-time Rendering of Large Terrains with Support for Runtime Modifications”
- Michal Valient, „Efficient Real-Time Shadows”

Q&A