



**SIGGRAPH** THINK  
BEYOND  
2020 S2020.SIGGRAPH.ORG

---

**USING MESH SHADERS  
FOR CONTINUOUS  
LEVEL-OF-DETAIL  
TERRAIN RENDERING**

Matthias Englert / Tinman 3D / GMT+1



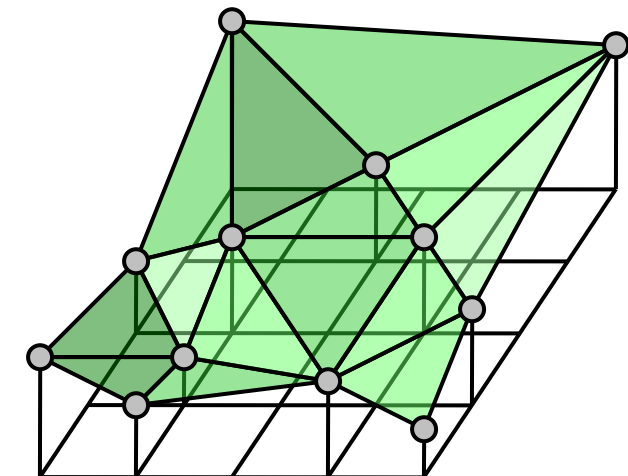
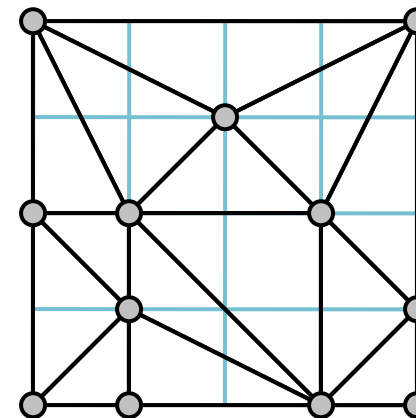
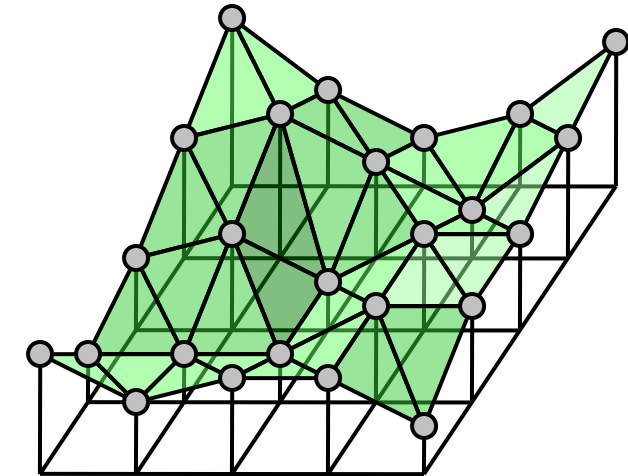
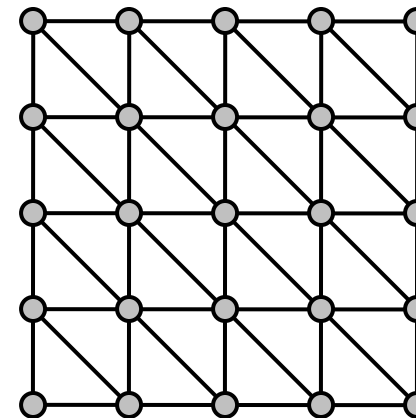
---

## PART 1: TERRAIN RENDERING

- Heightmaps (regular grids)
- Textures (image pyramids)
- Terrain mesh
- Spatial queries

# PART 1 – TERRAIN RENDERING – #1

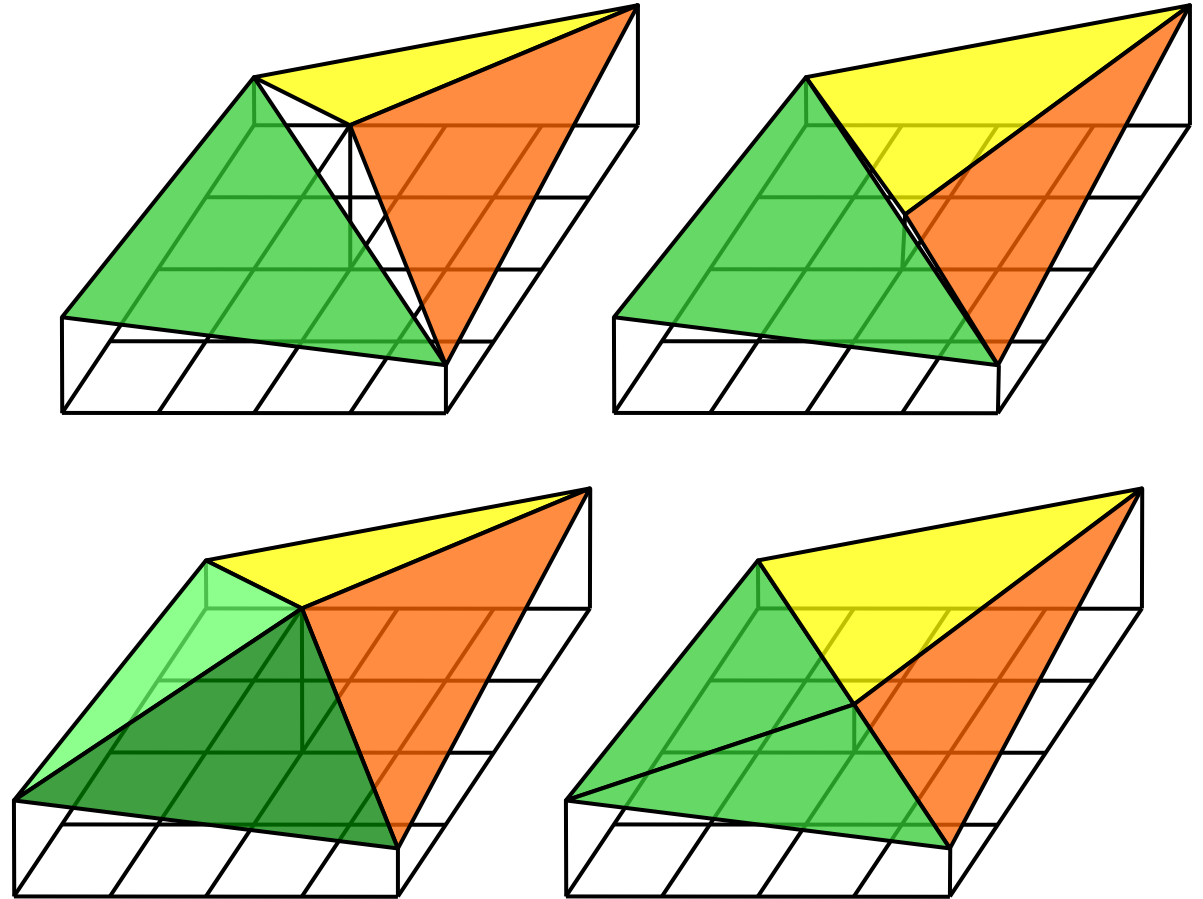
- “Huge” regular grids of terrain data samples
  - Not talking about “point soups” here
  - Cannot just load into CPU memory
    - ASTER GDEM has over  $2.8E+11$  samples
- “Huge” pyramids of terrain image tiles
  - Will probably never fit into GPU tiled texture
    - BingMaps zoom level 22 is  $1,073,741,824^2$  px<sup>2</sup>
- Want to render those in real-time
- Based on some visibility criterion, need to choose:
  - Which terrain data samples to use
  - Which terrain image tiles to use
- Use streaming to load the chosen data
- Build a triangle mesh from the chosen samples
- Map the chosen images as textures onto the mesh
- Render using your favourite API





# PART 1 – TERRAIN RENDERING – #2

- Building a triangle mesh is not trivial
  - Need to avoid gaps
  - Need to avoid T-junctions
  - Should be done by splitting triangles where necessary
  - Often done by adding extra “skirting” geometry
  - Need to be fast enough for real-time
- Mapping images is not trivial
  - Geo-referenced bounds usually do not line up with the cells of the regular grid
  - So we need to re-project them
- Having a texture-mapped triangle mesh is nice, but:
  - Also need to do spatial queries efficiently (e.g. picking)
  - Usually solved by augmenting the triangle mesh with some bounding-volume hierarchy



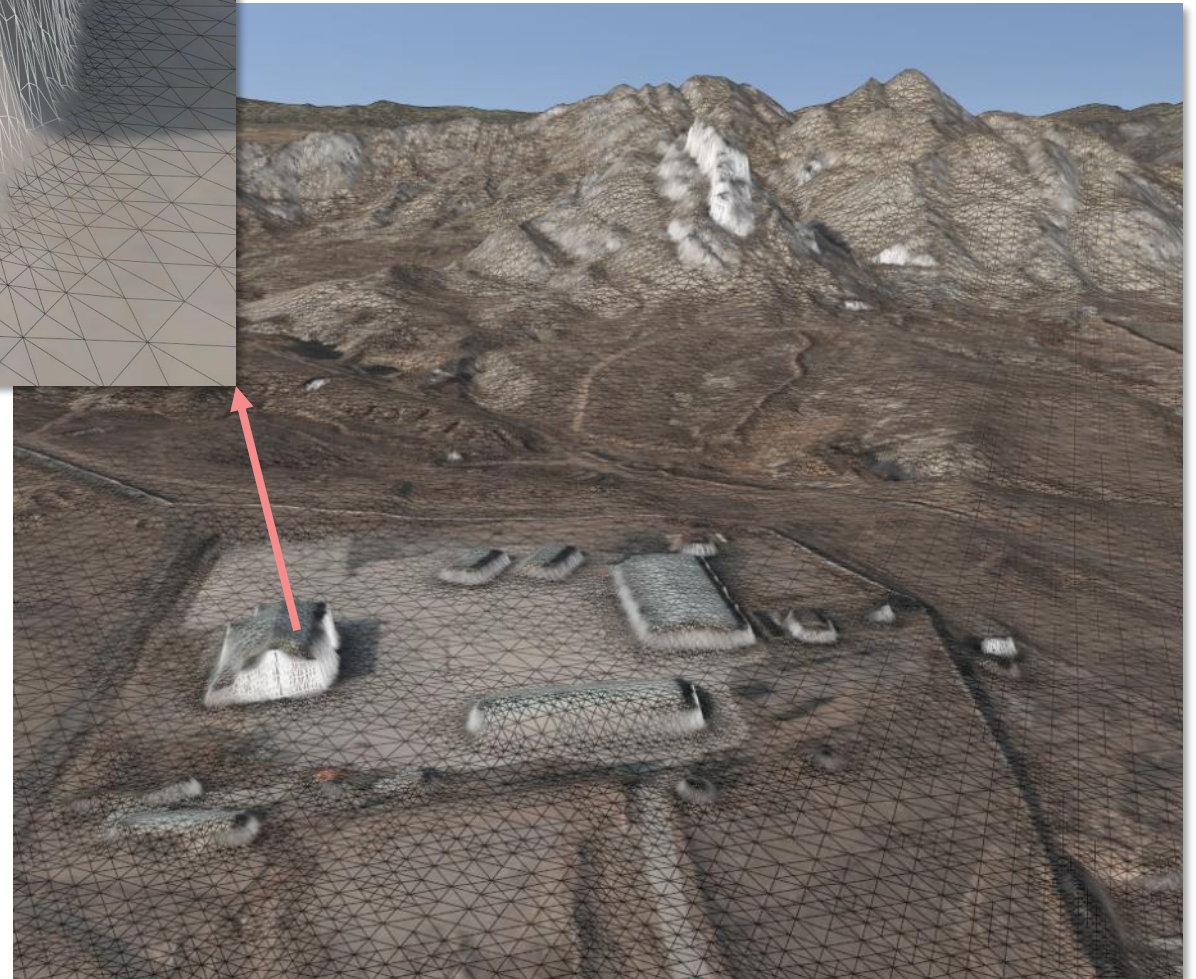
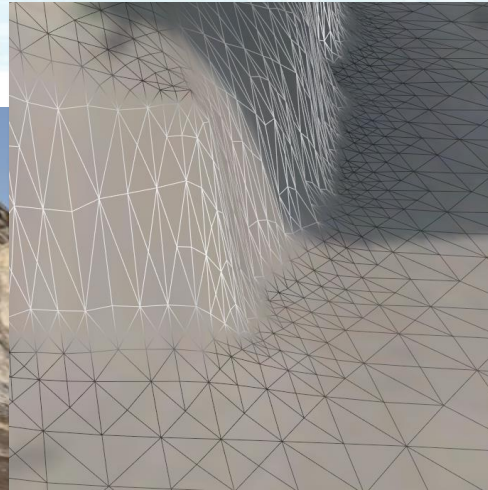
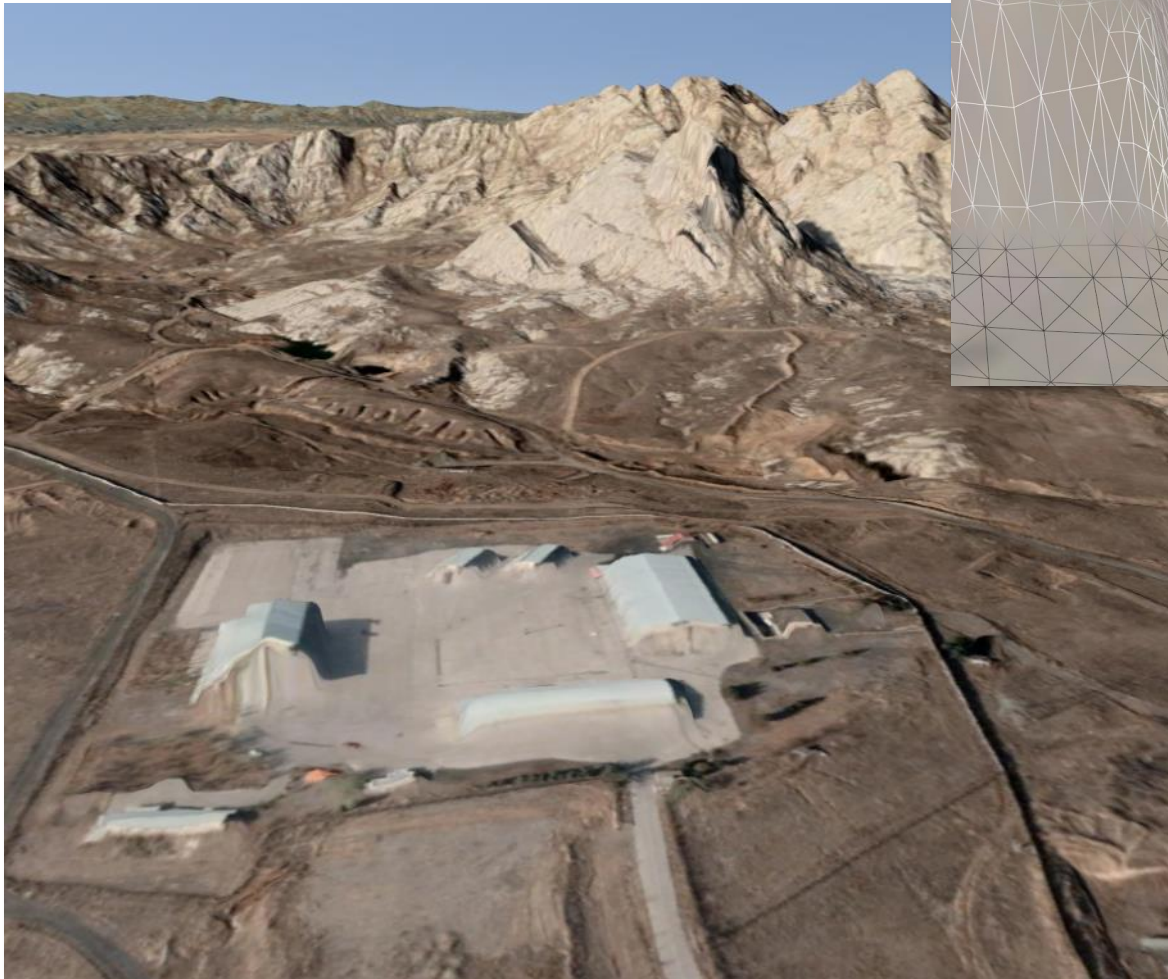
---

## PART 2: RTIN + CLOD

- Classical RTIN on modern GPU
- No ray-casting techniques
- No implicit techniques
- No mesh chunking



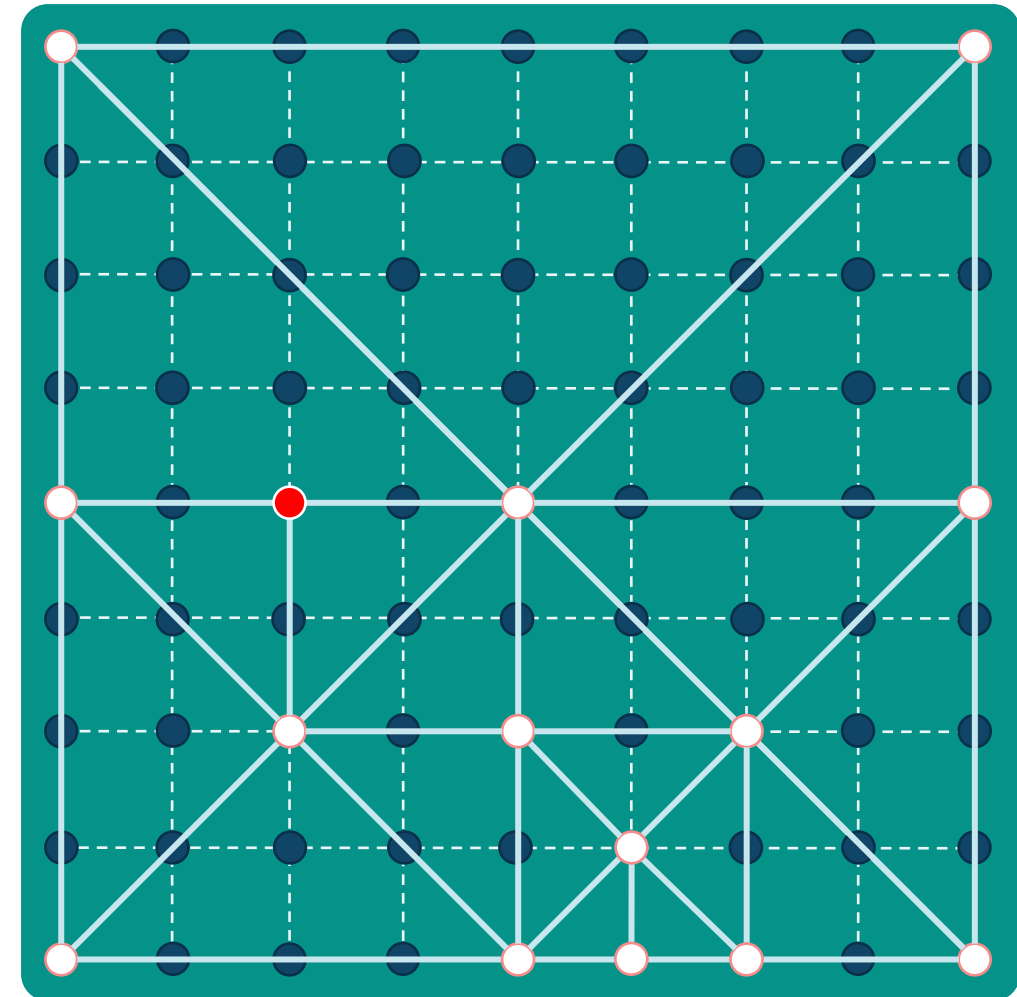
## PART 2 – RTIN + CLOD – #1





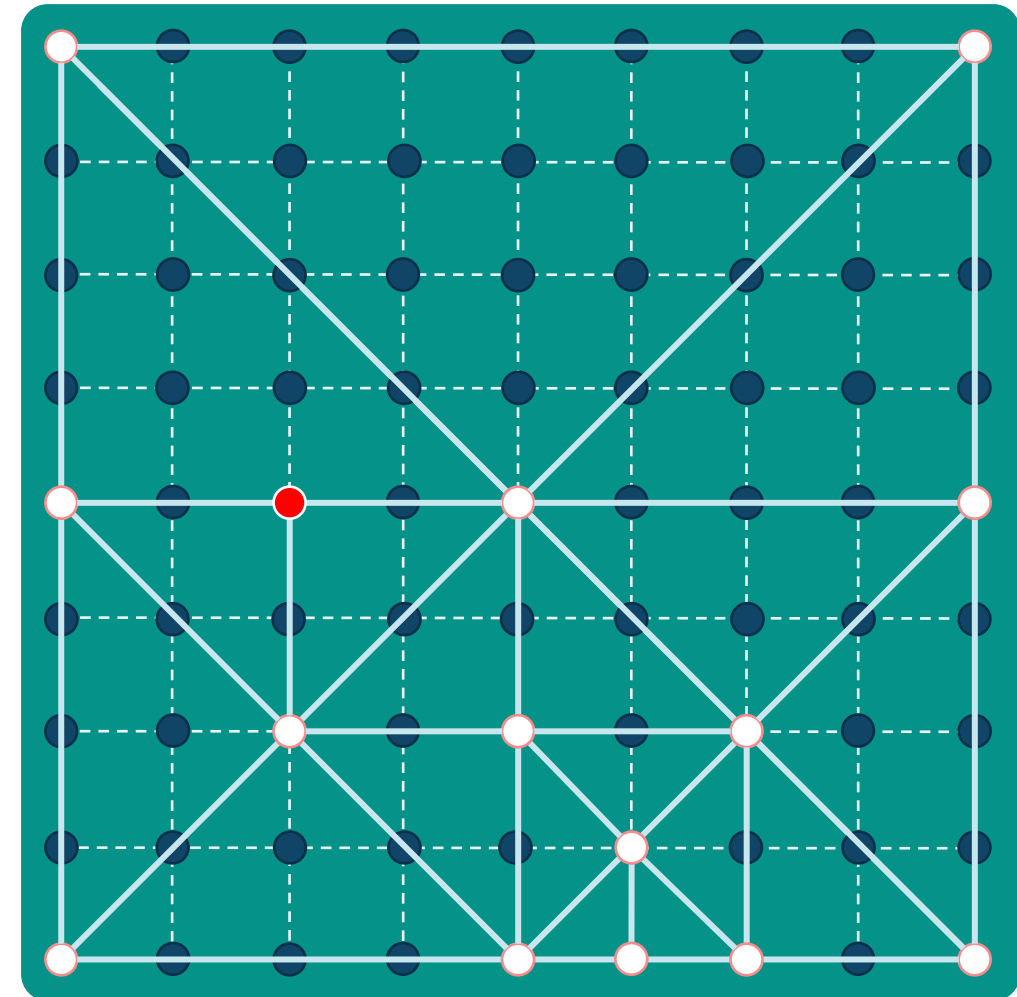
## PART 2 – RTIN + CLOD – #2

- Use of right-triangulated networks for continuous level-of-detail has been around for years:
  - “SOAR: Visualization of Large Terrains Made Easy”  
<https://computing.llnl.gov/projects/soar-visualization-large-terrains-made-easy>  
Peter Lindstrom and Valerio Pascucci, IEEE Transactions on Visualization and Computer Graphics, 8(3):239-254, July-September 2002.
  - “ROAMing terrain: Real-time Optimally Adapting Meshes”  
<https://ieeexplore.ieee.org/document/663860>  
M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich and M. B. Mineev-Weinstein, "ROAMing terrain: Real-time Optimally Adapting Meshes," *Proceedings. Visualization '97 (Cat. No. 97CB36155)*, Phoenix, AZ, USA, 1997, pp. 81-88, doi: 10.1109/VISUAL.1997.663860.
- As RTIN+CLOD not easily fits GPUs, there also is:
  - “Quad-Tree Atlas Ray Casting: A GPU Based Framework for Terrain Visualization and Its Applications”  
[https://link.springer.com/chapter/10.1007/978-3-642-29050-3\\_7](https://link.springer.com/chapter/10.1007/978-3-642-29050-3_7)  
Luo J., Ni G., Cui P., Jiang J., Duan Y., Hu G. (2012) Quad-Tree Atlas Ray Casting: A GPU Based Framework for Terrain Visualization and Its Applications. In: Pan Z., Cheok A.D., Müller W., Chang M., Zhang M. (eds) *Transactions on Edutainment VII. Lecture Notes in Computer Science*, vol 7145. Springer, Berlin, Heidelberg
  - “Adaptive GPU Tessellation with Compute Shaders”  
<https://github.com/jdupuy/opengl-framework/tree/master/demo-isubd-terrain>  
Jonathan Dupuy Jad Khoury and Christophe Riccio 2018.
  - “Planet-sized batched dynamic adaptive meshes (P-BDAM)”  
<https://ieeexplore.ieee.org/abstract/document/1250366>  
P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio and R. Scopigno, "Planet-sized batched dynamic adaptive meshes (P-BDAM)," *IEEE Visualization, 2003. VIS 2003.*, Seattle, WA, USA, 2003, pp. 147-154, doi: 10.1109/VISUAL.2003.1250366.



## PART 2 – RTIN + CLOD – #3

- Using 100% classic RTIN + CLOD on modern GPUs...
  - No ray-casting on the GPU
  - No implicit subdivision on the GPU
  - No terrain chunking
- ...is great, because we can...
  - get a quad-tree for free (when encoding the RTIN properly)
  - do spatial queries on the CPU easily
  - exploit the RTIN structure on the GPU...
  - ...as well as on the CPU
  - load terrain data samples at optimal granularity
- ...is hard, because we must...
  - encode the RTIN properly and efficiently
    - no pointers or dynamic memory allocation
  - resolve T-junctions via forced triangle splits
  - need to triangulate the RTIN
  - compute per-vertex LOD data (e.g. normal vectors)





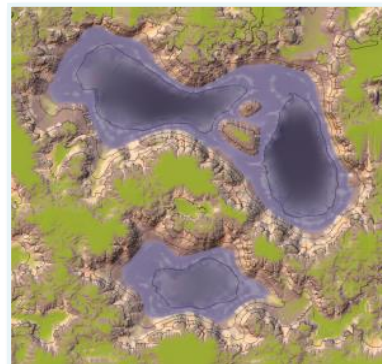
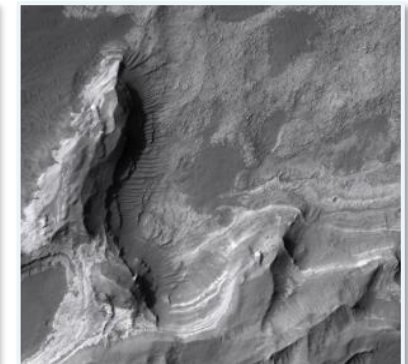
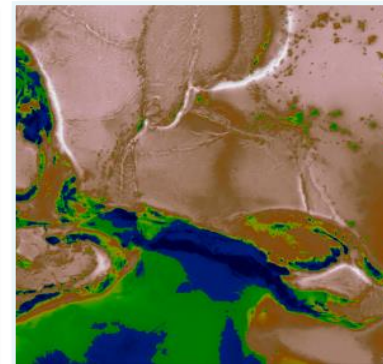
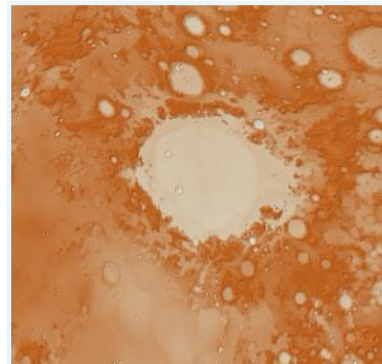
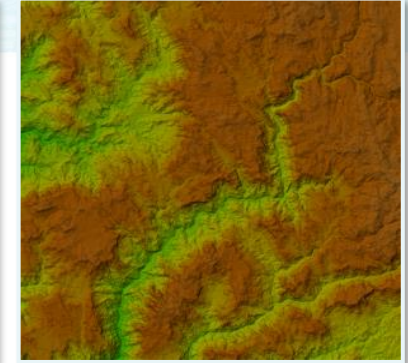
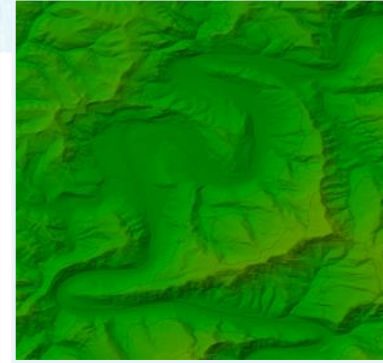
---

## PART 3: TERRAIN DATA

- Uniform data representation
- Hierarchical raster data
- Hierarchical image data
- Level-of-detail partitioning

# PART 3 – TERRAIN DATA – #1

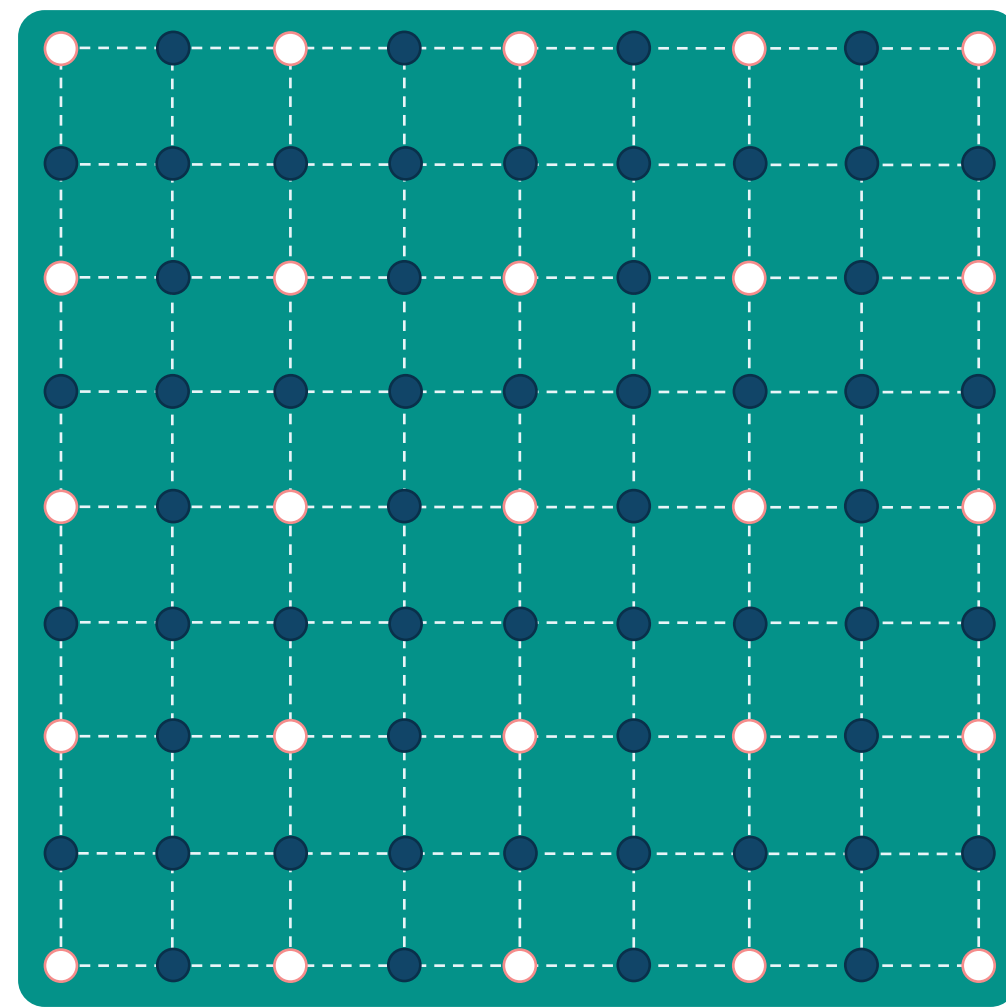
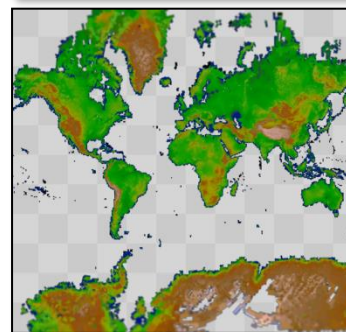
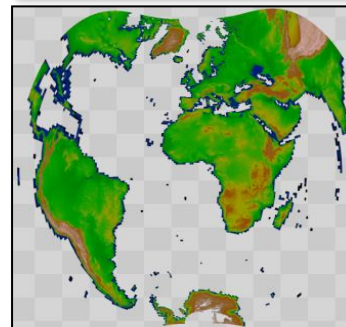
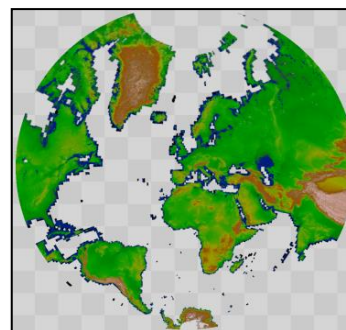
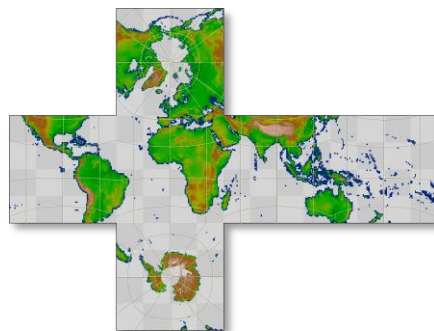
- Non-uniform terrain data sources
  - Imagery / elevation, geo-referenced
  - RAW, PNG, GeoTIFF, JPEG2000
  - Datum (WGS84/EGM96)
  - Projection (Mercator)
  - Parameters (origin, scale)
- *ASTER GDEM*  
<https://asterweb.jpl.nasa.gov/gdem.asp>
  - More than 22,000 tiles
  - Each tile has 3,601 by 3,601 pixels
- Lots of data, myriad of combinations
  - Need uniform basis for real-time!





## PART 3 – TERRAIN DATA – #2







- Uniform 2D raster for terrain data samples
  - $2^{N+1}$  by  $2^{N+1}$  (pixel-is-point)
  - $N=30$  yields 9mm ground resolution for Earth
  - Easy to interpolate / merge at runtime
- Uniform 2D texture for terrain imagery
  - $2^N$  by  $2^N$  (pixel-is-area)
    - GPU (tiled) textures only for  $N \leq 14$
  - Split into tile pyramid
    - For  $N=30$  and a tile size of 512, we get 22 levels
- Pad for rectangular maps
- Use map projection
  - Oblique Stereographic
  - Cassini
  - Mercator
  - Geographic Cubemap



# PART 3 – TERRAIN DATA – #3

- Example LOD partitioning
  - Grid: 16 x 16
  - Block: 4 x 4
- Data management
  - LOD-IDs per block
  - Per-block storage
  - Lossless compression
  - Random access
- Actual LOD partitioning
  - Grid:  $2^{30}+1 \times 2^{30}+1$
  - Block: 256 x 256



-  Block #0 on level #0
-  Block #1 on level #1
-  Block #2 on level #1
-  Block #3 on level #1
-  Block #4 on level #2
- ...
-  Block #15 on Level #3



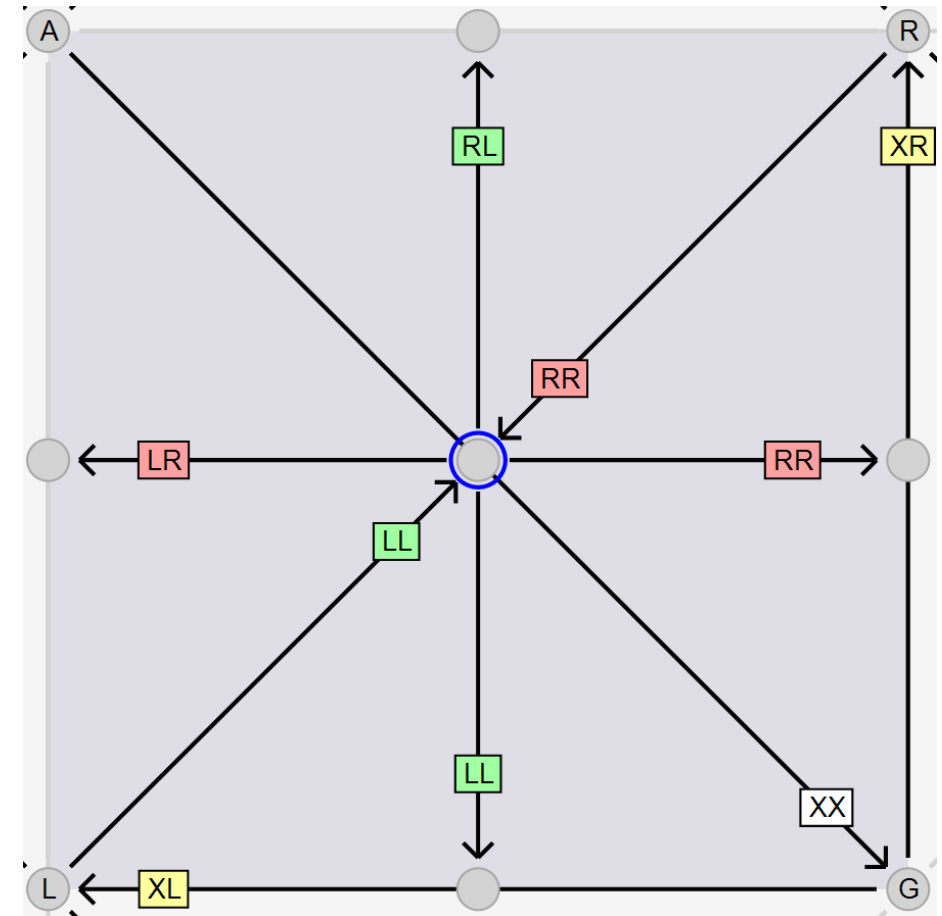
---

## PART 4: X-DAG

- eXtended Directed Acyclic Graph
- Semantics
- Vertex pooling
- Cache locality

# PART 4 – X-DAG – #1

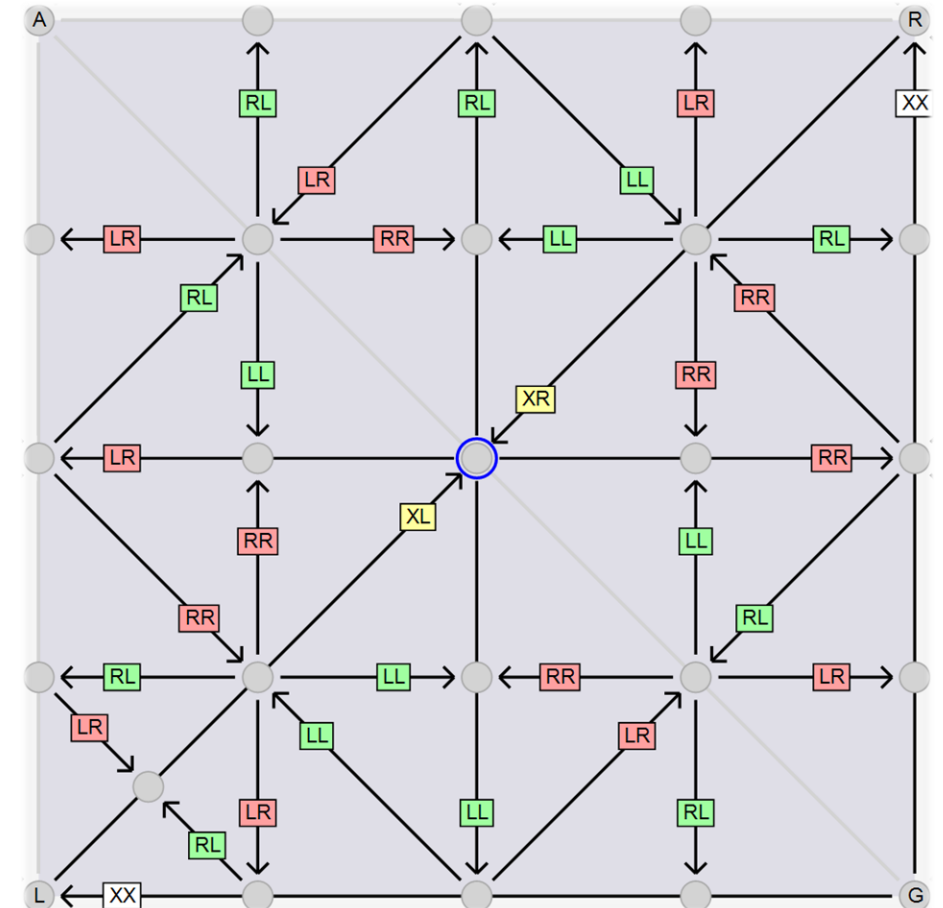
- Using an explicit **D**irected **A**cyclic **G**raph for RTIN encoding
  - Per-vertex child references (4x)
    - “Acyclic” with respect to child references
  - Per-vertex parent references (2x)
  - Per-vertex grand-parent reference (1x)
  - Per-vertex ancestor reference (1x)
- Fast traversal of the DAG is critical for many algorithms
  - T-junction removal
  - Triangulation
  - CLOD refinement
- Adding e**X**tra geometric semantic to the DAG references:
  - Downwards: **LL**, **LR**, **RL**, **RR**, upwards: **L**, **R**, **G**, **A**
  - L := left, R := right
  - Based on triangle split (apex up): left half and right half.





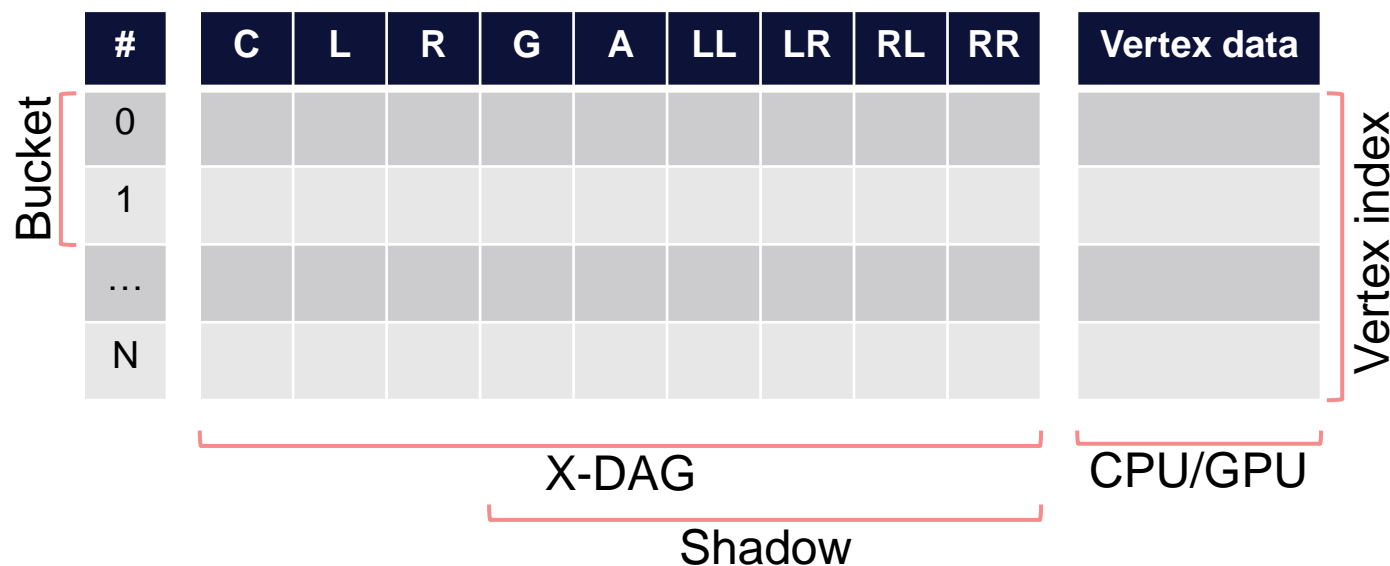
## PART 4 – X-DAG – #2

- Applying the geometric semantic for a vertex **V** yields...
  - Parent / child relationships  
 $V.LL.L = V$     $V.LR.R = R$     $V.RL.L = V$     $V.RR.R = V$
  - Centre of child quadtree node  
 $V.LL.LL$     $V.LL.LR$     $V.RL.LL$     $V.RL.LR$
  - Nearest descendant in line to L-parent  
 $V.LL.LL.(LL.LR.(RL.LR)^*)?$
  - and much more...
- Can exploit this by writing CPU/GPU programs that take hard-coded paths through memory, without traversal arithmetic
  - Use nearest descendants to compute normal / tangent vectors
  - X-DAG traversal for refinement, hitting each vertex exactly one
  - Move along space filling curve through X-DAG for triangulation
  - Quad-tree traversal for spatial queries
  - Move between adjacent vertices for path-finding



# PART 4 – X-DAG – #3

- Vertex pooling
  - Fixed-size vertex buffer,  $0 \leq \text{index} \leq N$ 
    - Divided into equally-sized buckets
  - Per-vertex lock count C
    - +1 for each child vertex
    - +1 if visible to CPU render-thread / GPU
  - Free store contains unlocked vertices
    - Linked list of free vertex indices
    - Binary heap of free vertex indices
- Take out of free store:
  - Compute LOD-ID of block that contains the new vertex
  - Hash LOD-ID to a bucket index
  - Search heap to find free index nearest to bucket. Take it.
  - If heap is empty, take least-recently used index from linked list.



- Put vertex into free store:
  - Append to linked list (most-recently used)
  - Maintain maximum list size (e.g. 1024)
    - Take least-recently used index from list
- X-DAG shadow copy
  - Contains all visible vertices
  - Only LL, LR, RL, RR, G, A references are shadowed
    - CPU (terrain work in render thread)
    - GPU (triangulation / traversal)

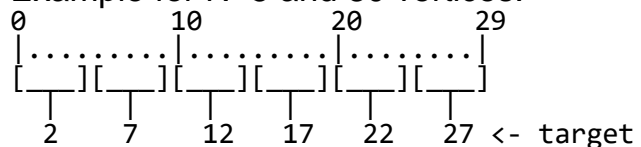


# PART 4 – X-DAG – #4

- Cache Locality

- Spatial buckets have N vertices each

- N=64 seems to be a good choice
- Example for N=5 and 30 vertices:

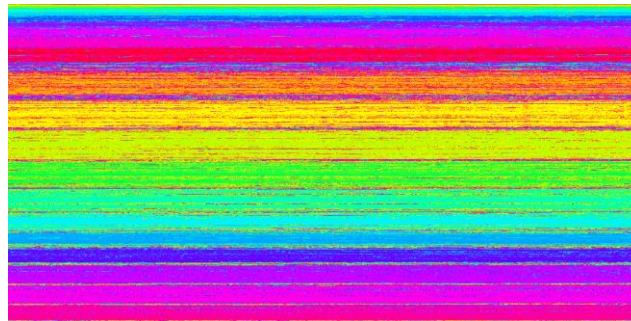


- Bucket centre vertex is used as ideal target when looking for nearest free slot

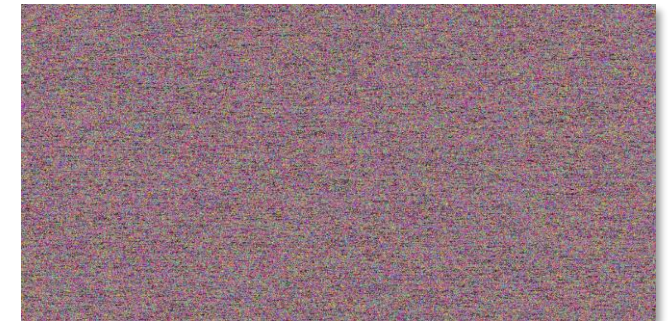
- Colours correlate with LOD-IDs

- Rainbow-look means “good”
- Noisy-look means “bad”
- Dark just means “unused” and is OK
- Top row shows naïve implementation
  - LRU only, no spatial buckets
  - “Optimal” initial performance, but dramatically drops later.
- Bottom row shows the described impl.
  - “Near-optimal” initial performance, no significant drop later.

Initial (naïve LRU)



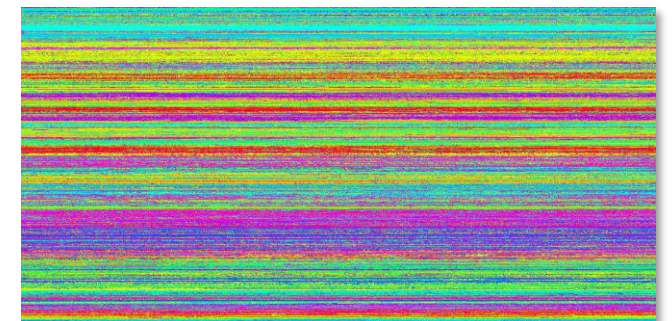
Later (naïve LRU)



Initial (spatial buckets)



Later (spatial buckets)



---

## PART 5: GPU SETUP

- Buffers
- Shaders
- Triangulation
- From DX9 to DX12



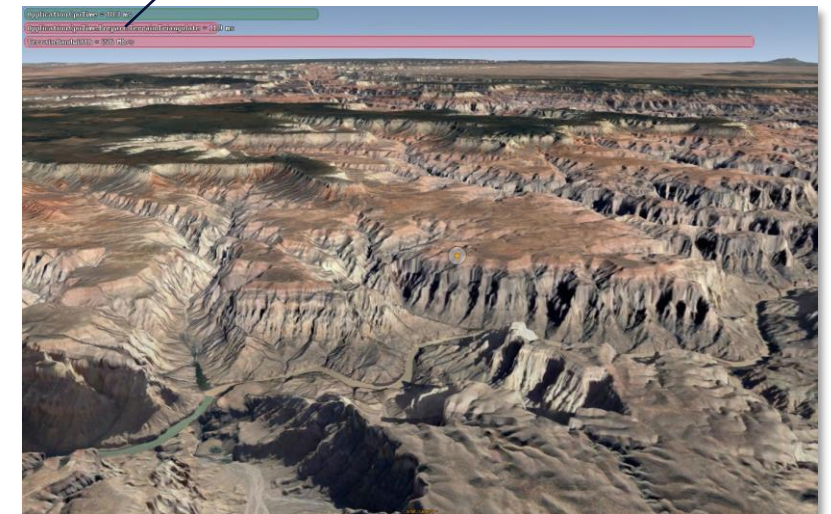
# PART 5 – GPU SETUP – DX9

- Dynamic vertex buffer in shared GPU memory
  - Update with NO\_OVERWRITE
- Volatile index buffer in shared GPU memory
  - Update with DISCARD
- Texture atlas with  $N$  separate 2D textures
- 1 draw call per terrain batch
- CPU triangulation + GPU upload
  - Triangle list: 12 bytes per triangle
  - Triangle strip: ~6.5 bytes per triangle
  - **Bottleneck for detailed terrains**
    - **Bandwidth usage > 200 MB/s**
    - **CPU render thread 50% busy**
- CPU visibility culling
  - View frustum
  - Horizon
- No displacement mapping
  - Not counting D3DRS\_ENABLEADAPTIVETESSELLATION here...

Application.CpuTime = 18.3 ms

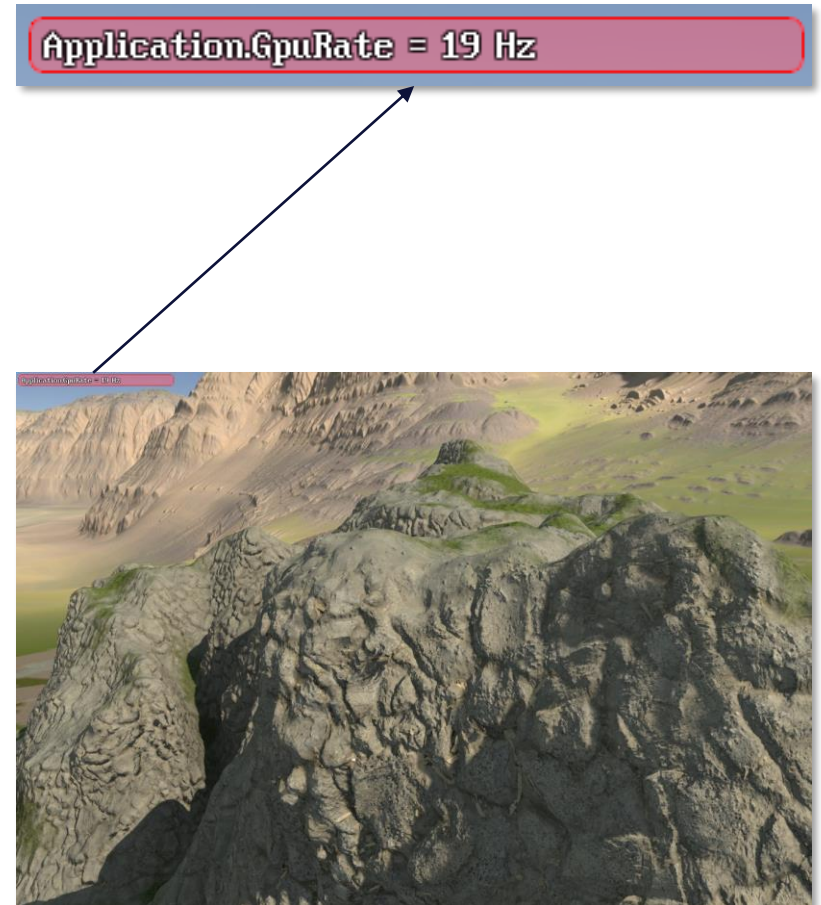
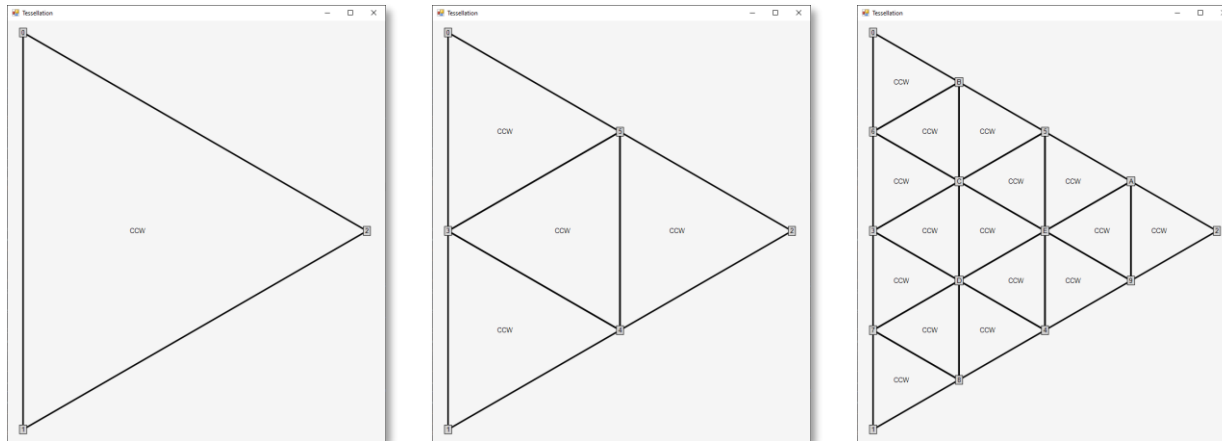
Application.CpuTime.Prepare.Terrain.Triangulate = 11.9 ms

Terrain.Bandwidth = 276 Mb/s



# PART 5 – GPU SETUP – DX10

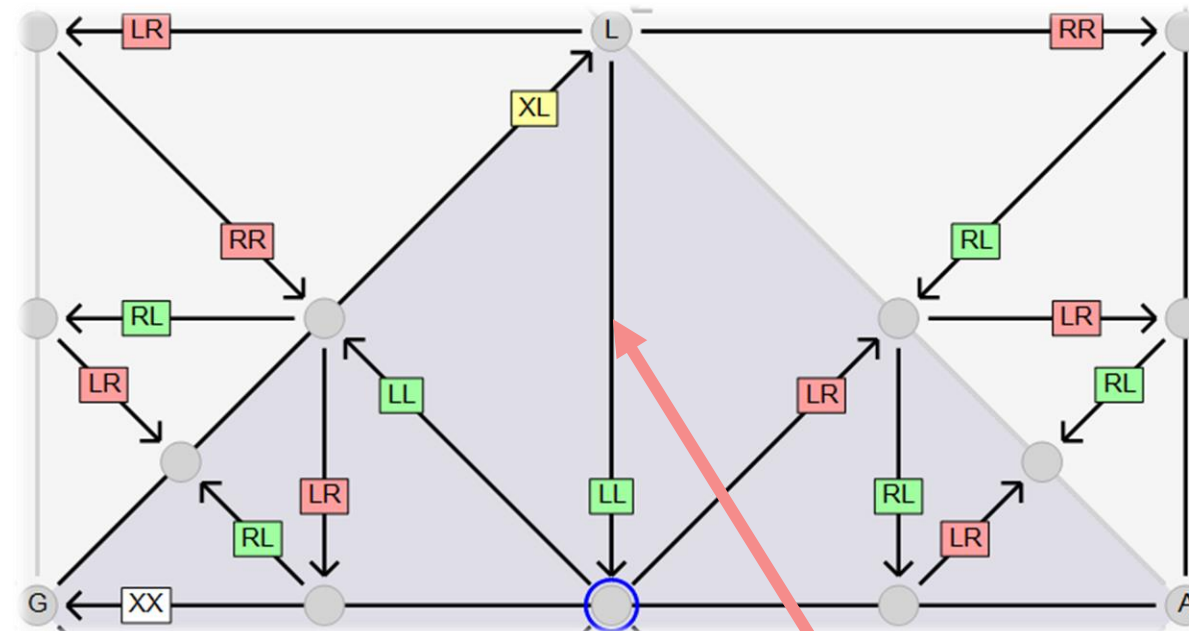
- Texture atlas with 2D texture arrays
  - No longer need to swap textures per terrain batch
- Displacement mapping with Geometry Shader
  - Simple 1-to-4 triangle subdivision
  - Up to two subdivision levels
  - **Bottleneck for detailed terrain meshes**
    - **< 20 FPS, even on NVIDIA GeForce RTX 2080 Ti**





# PART 5 – GPU SETUP – DX11

- CPU pre-triangulation + GPU upload
  - Terminal triangle list: ~1 byte per triangle
  - Bandwidth & CPU usage at ~10% of DX9 levels
- GPU triangulation
  - Compute Shader expands terminal triangles to a list of leaf triangles and generates indirect draw calls.
  - Dynamic structured buffer in shared GPU memory, holding the X-DAG shadow
    - Update with NO\_OVERWRITE
- 1 compute dispatch call per frame
- 1 indirect draw call per terrain batch
- Displacement mapping with Hull and Domain Shaders



# PART 5 – GPU SETUP – DX12

- CPU pre-triangulation + GPU upload
  - Sector triangle list: < 0.1 byte per triangle
  - Bandwidth & CPU usage at irrelevant levels
- GPU triangulation
  - Amplification Shader expands sector list to terminal triangle list
    - Shader output limit is a problem
    - CPU pre-triangulation must traverse deeper to reduce the number of terminal triangles per sector
  - Mesh Shader expands terminal triangles to leaf triangles
    - Shader output limit is fine for vertex sizes of ~256 bytes or less, as there is an inherent upper limit to the number of leaf triangles in a terminal triangle of ~128
- 1 mesh dispatch call per frame
- Dynamic GPU buffers in L1 GPU memory
  - Use D3D12\_HEAP\_TYPE\_UPLOAD to accumulate incremental updates
    - Vertex data
    - X-DAG shadow
  - Use CopyBufferRegionUpdate to transfer the updates .
- Displacement mapping with Mesh Shader
  - Using modified DX10 subdivision code
    - Can exploit inherent adjacency of terminal triangles
    - Shader output limit is a problem: the number of subdivided leaf triangles in a terminal triangle must not exceed 256
    - Amplification Shader must traverse deeper to reduce the number of leaf triangles
    - Need to escalate up to CPU pre-triangulation, if Amplification Shader would run into its own output limit
- Difficult to implement ☹
- But parallel processing is worth it 😊



---

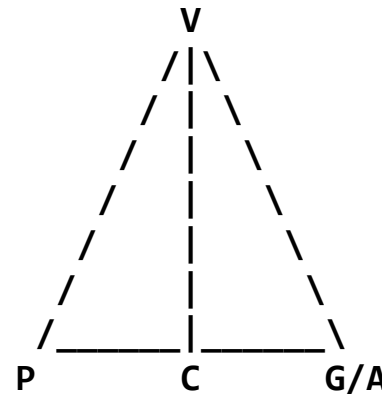
## PART 6: IMPLEMENTATION DETAILS

- “Terminal Triangle List” primitive
- “Sector List” primitive
- Culling with inline ray-tracing

# PART 6: IMPLEMENTATION DETAILS #1

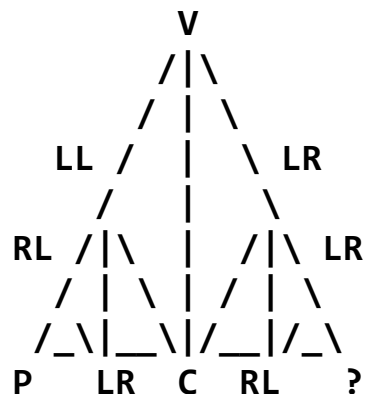
## GPU primitive: “Terminal Triangle List”

- One 32-bit word per terminal triangle
- CPU performs regular triangulation but stops at terminal triangles
- GPU decodes terminal triangle, traverse the X-DAG shadow and generates triangles accordingly

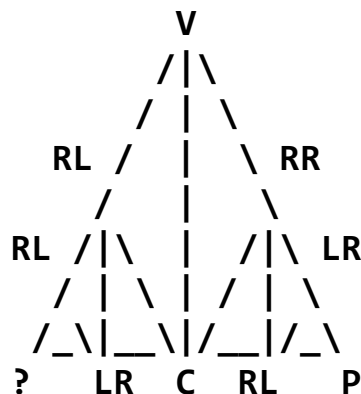


bits [0.. 1] : child index of C in V  
(LL=0, LR=1, RL=2, RR=3)  
bits [1.. 2] : child index of V in P  
bits [3.. 3] : triangulate (V,P,C)?  
bits [4.. 4] : triangulate (V,C,G/A)?  
bits [5.. 5] : flip triangle winding?  
bits [6..31] : index of vertex P

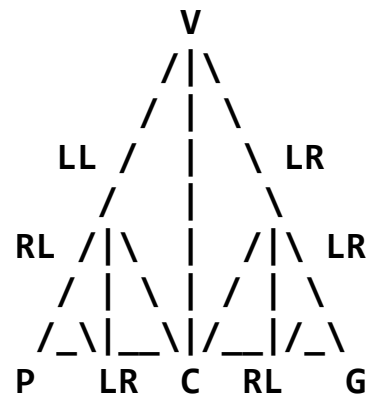
V.?L == C



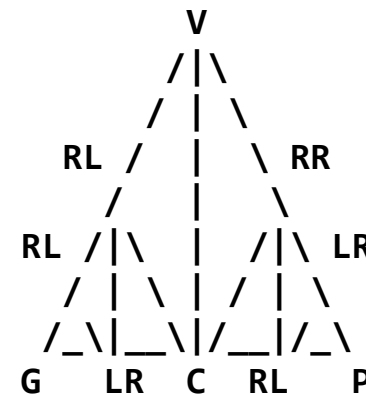
V.?R == C



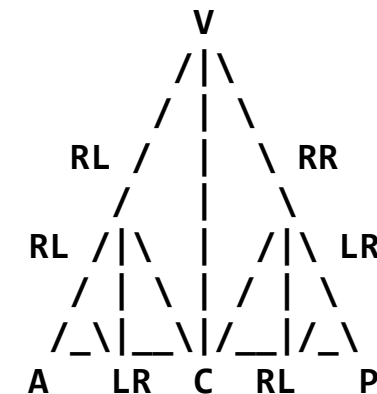
V.LL == C



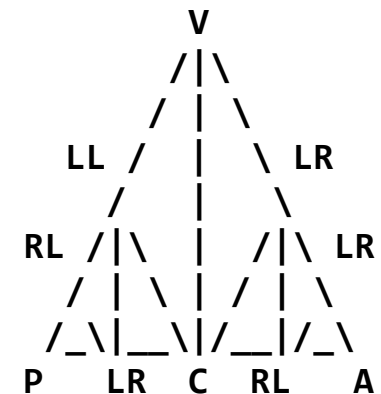
V.RR == C



V.LR == C



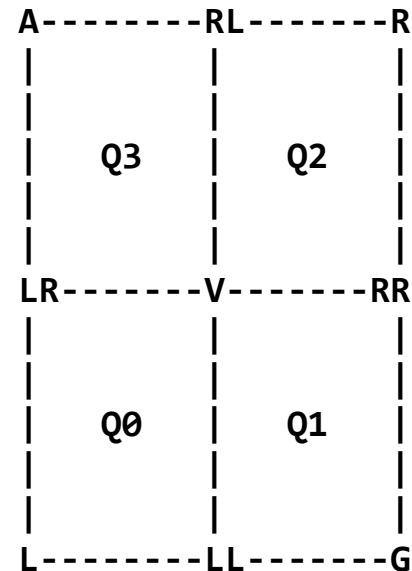
V .RL == C



## PART 6: IMPLEMENTATION DETAILS #2

### GPU primitive: “Sector List”

- One 32-bit word per sector code
- CPU performs regular triangulation but stops at sectors that the application wants to render separately
- GPU decodes sector code, traverse the X-DAG shadow and generates terminal triangles accordingly



bits [0.. 2] : index of leaf quadrant Q  
in the range [0..3]  
or 4 for whole sector  
bits [3.. 3] : flip triangle winding?  
bits [4..31] : index of vertex V

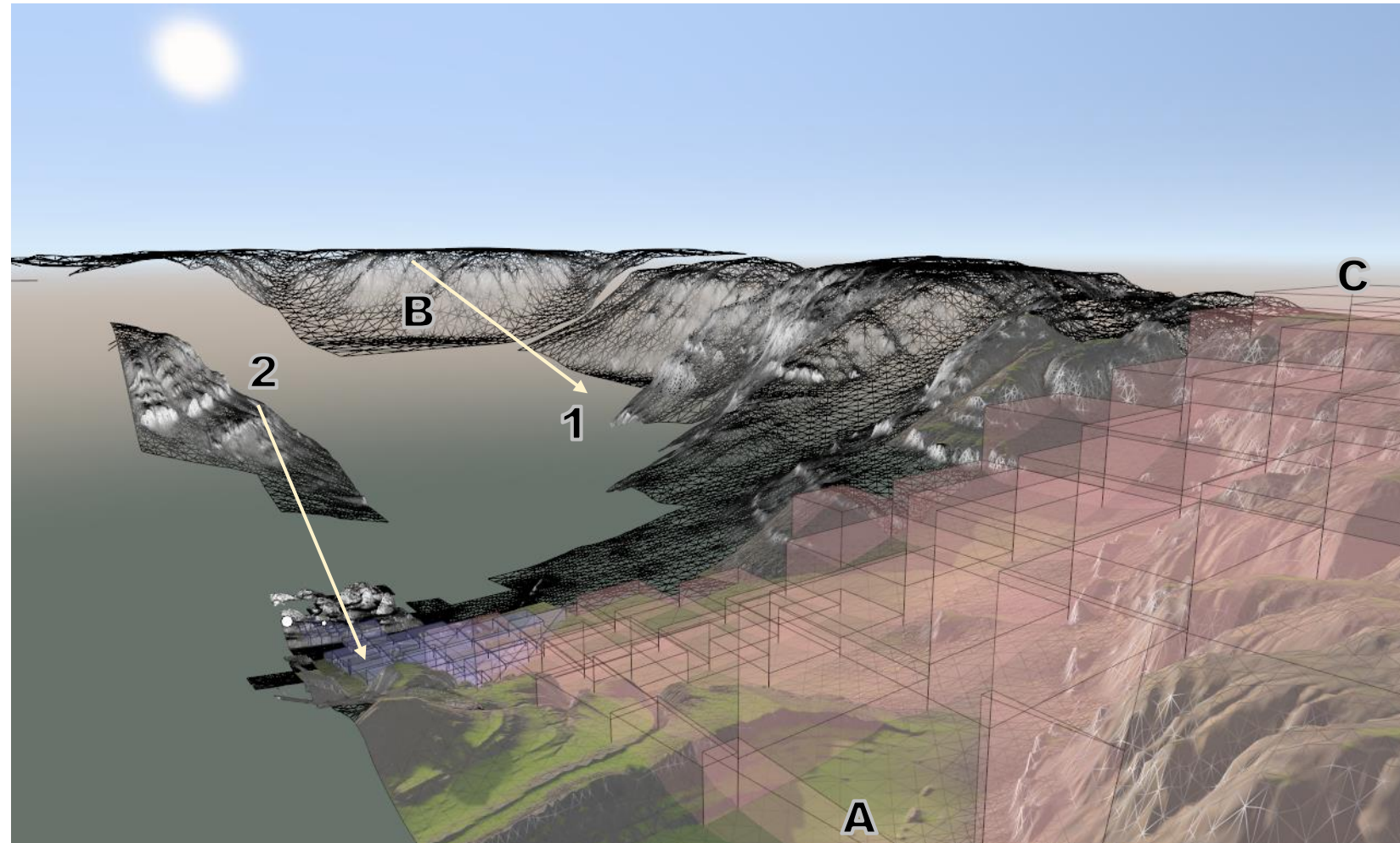
V.L = L	V.LL.LL = Q0
V.R = R	V.LL.LR = Q1
V.G = G	V.RL.LL = Q2
V.A = A	V.RL.LR = Q3



# PART 6: IMPLEMENTATION DETAILS #3

## Culling with inline raytracing

- Potential shadow-casters are found by frustum culling in light-space
  - **A** shows the shadow-receivers
  - **B** shows potential shadow-casters
  - **C** is the bounding-volume-hierarchy
- Reduce **B** by casting inline rays:
  - Add BVH to acceleration structure, two cuboids per mesh sector, representing minimum resp. maximum elevation
    - Use instance masking to distinguish
    - Update via transformation matrix
  - In Amplification/Mesh Shader, shoot light-direction rays onto maximum-BVH:
    - **1** Skip geometry when rays miss
    - **2** Process geometry otherwise
- Also works for horizon culling
  - Shoot reverse eye-vectors
  - Use minimum-BVH



---

## PART 7: DEMONSTRATION

- Try it yourself!
- New GPU primitives:  
less bandwidth usage  
less CPU work
- Inline ray-tracing:  
advanced visibility culling  
w/o “heavy” CPU work



# PART 7 – DEMONSTRATION

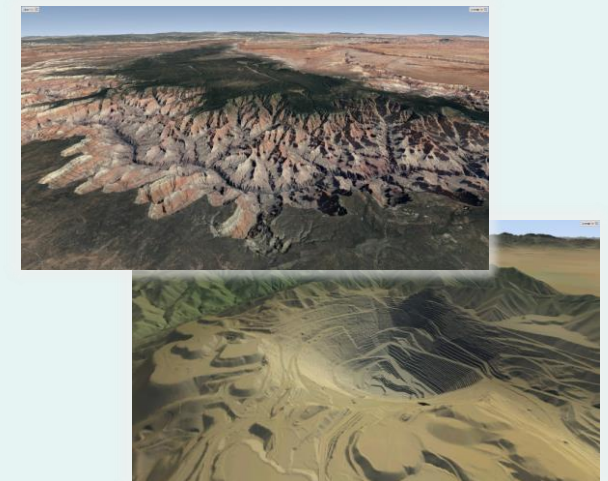
## Try it yourself!

1. Browse “Algorithms.cs”
2. Download the SDK  
<https://www.tinman3d.com>  
- *no registration required* -
3. Run the “Demo Application”  
- *no license required* -
4. Inspect the HLSL shaders  
- *full source code included* -
5. Questions? Please ask:  
[me@tinman3d.com](mailto:me@tinman3d.com)

## Artificial Terrain

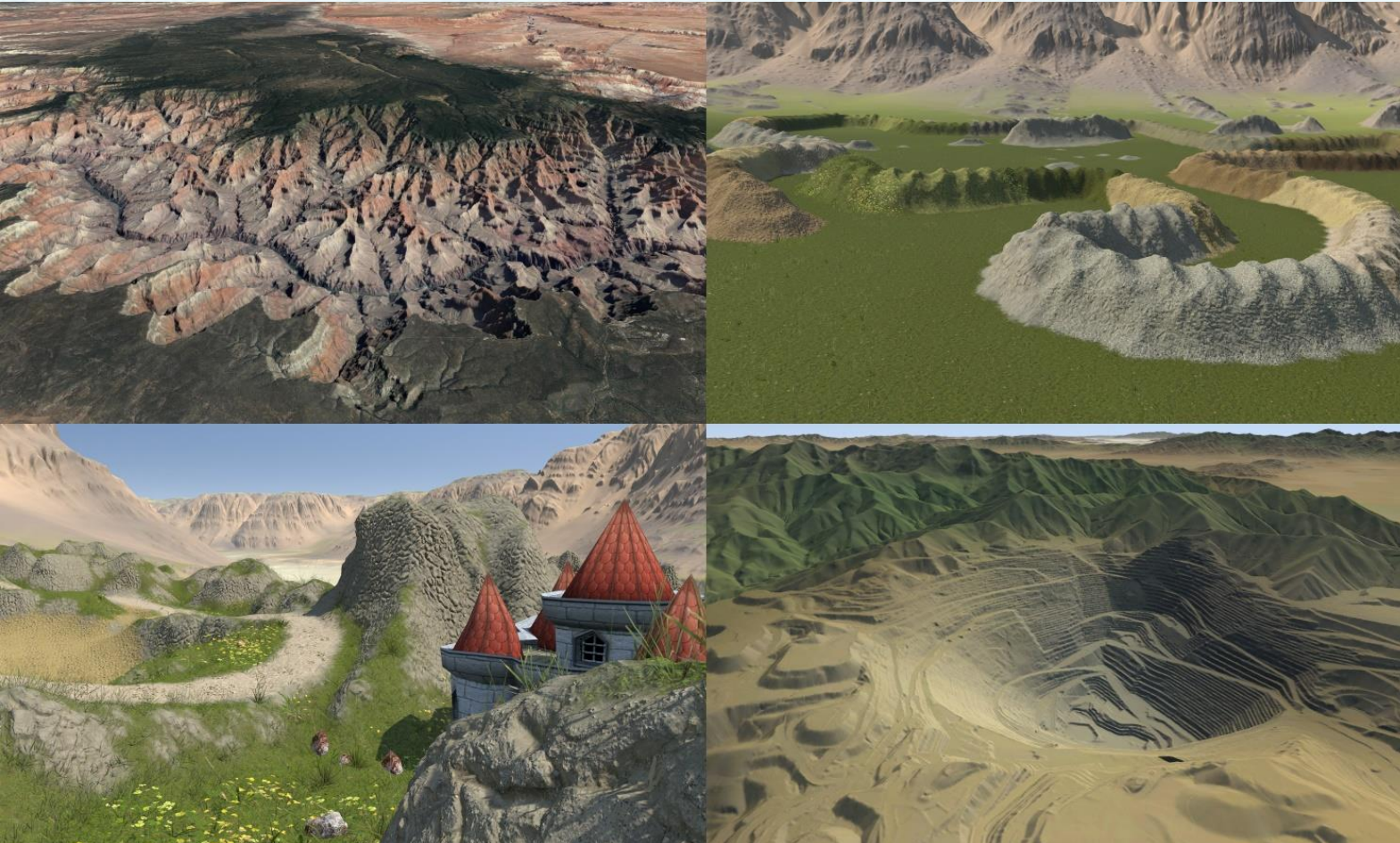


## Earth





# THE END



Thank you for listening!

Questions?

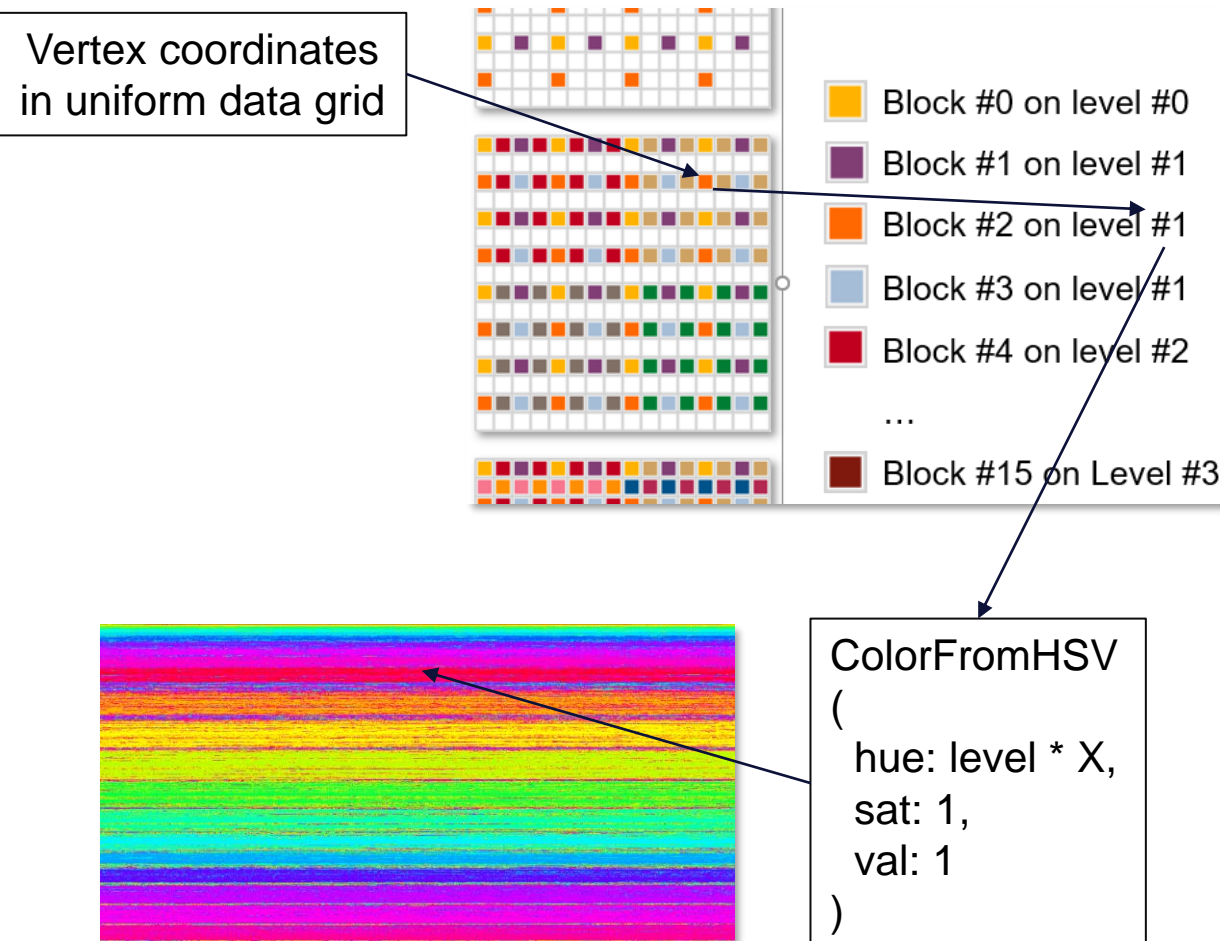
Please ask:

Matthias Englert  
[me@tinman3d.com](mailto:me@tinman3d.com)

# QUESTIONS & ANSWERS – #1

*“In the cache locality colour diagram, the colours represent some sort of heuristic for locality, can you explain what exactly is that heuristic for our locality graph?”*

- Take X/Y coordinates of vertex in uniform data grid
- Apply LOD partitioning to compute block ID / level
- Take block level, scale by some nice-looking value and map to hue cycle.
- “Rainbow look” means:
  - Vertices on same LOD level have spatial coherence in vertex buffer.
  - Grouping by LOD block ID is not apparent in this simple colouring scheme.

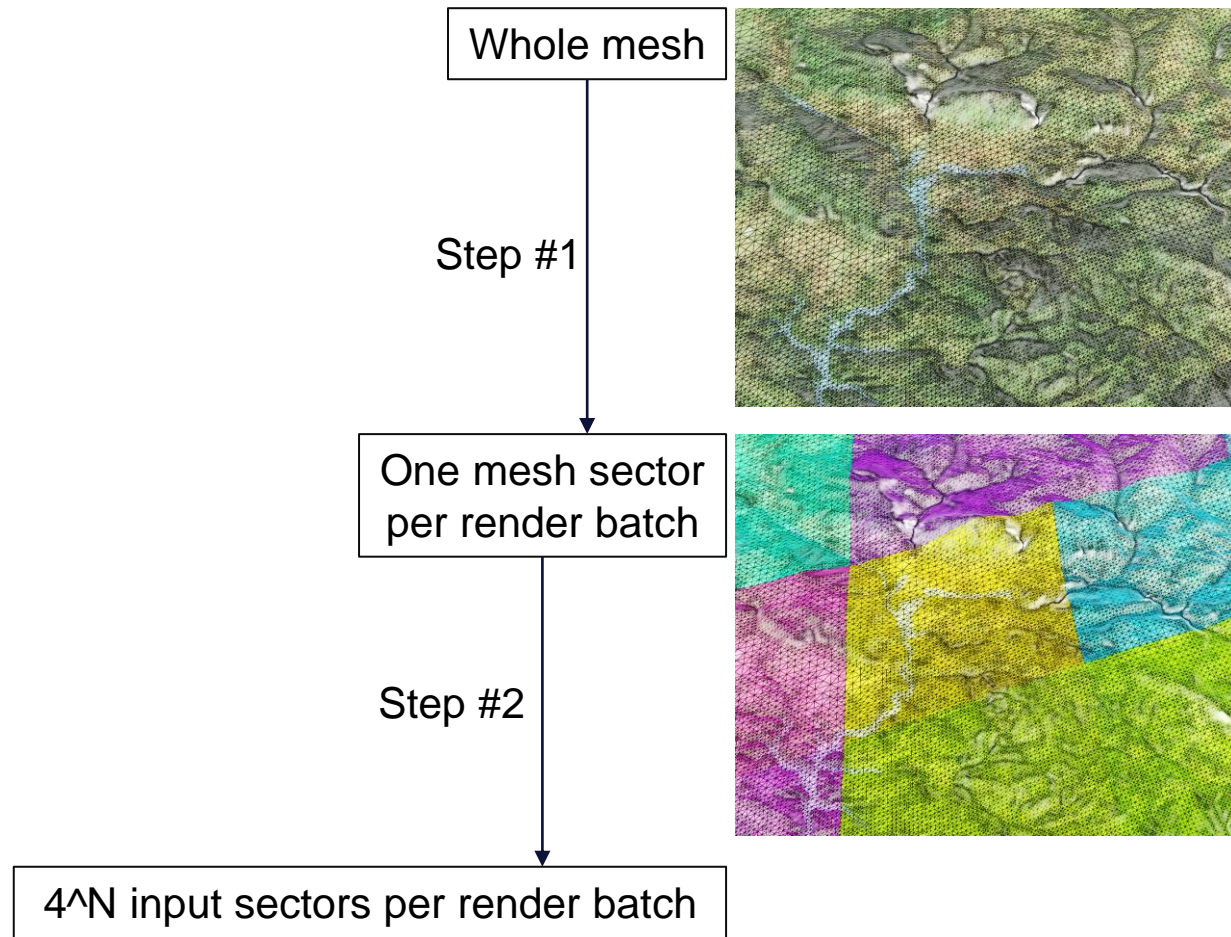




## QUESTIONS & ANSWERS – #2

*“You mentioned that in DX12 for amplification/mesh shaders the CPU was traversing the Sectors. What sort of approach did you use to determine how much the CPU must traverse the network? Is this done automatically or manually by trial and error?”*

1. CPU traverses the terrain mesh, as required by culling, texturing, etc. This yields ~100..200 sectors, having roughly the same screen size and roughly the same triangle count, assuming the terrain surface “complexity” is similar.  
=> automatic
2. CPU traverses N quadtree-levels deeper.  
N too low: no parallel processing  
N too high: CPU / bandwidth usage too high  
=> trial and error, N=3 seems to be a good value

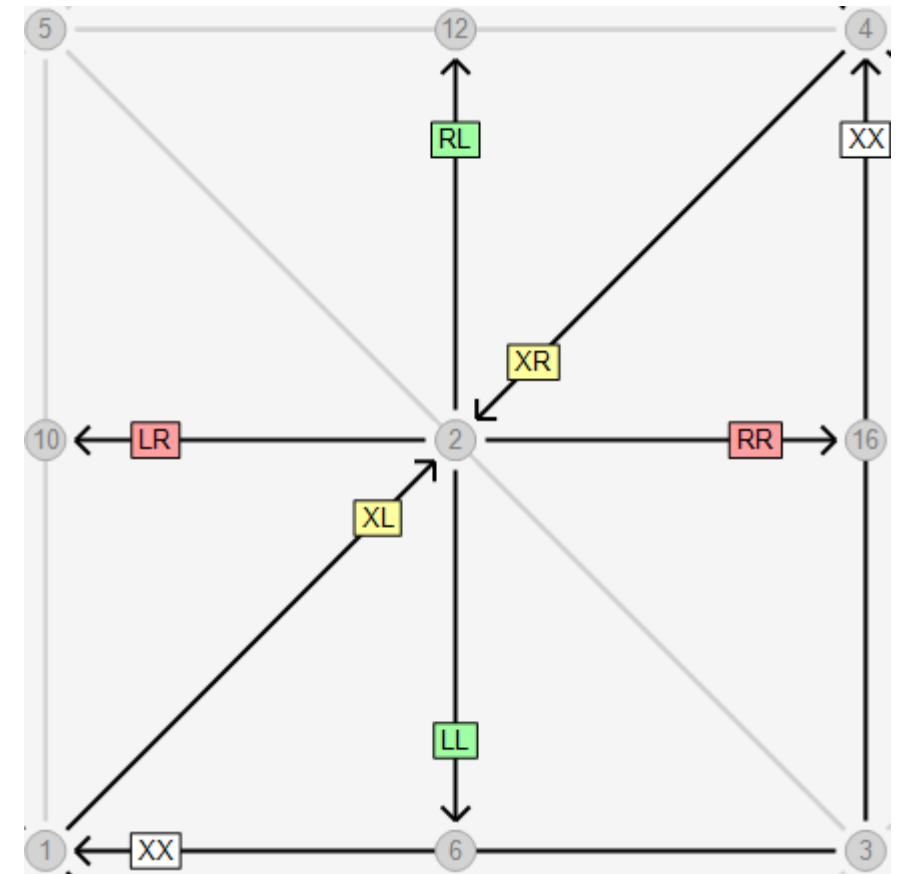




# QUESTIONS & ANSWERS – #3

*“The encoding of a terminal triangle, Slide # 24 you show that bits 0..1 means the ‘child index of C in V’ what exactly does this index mean?”*

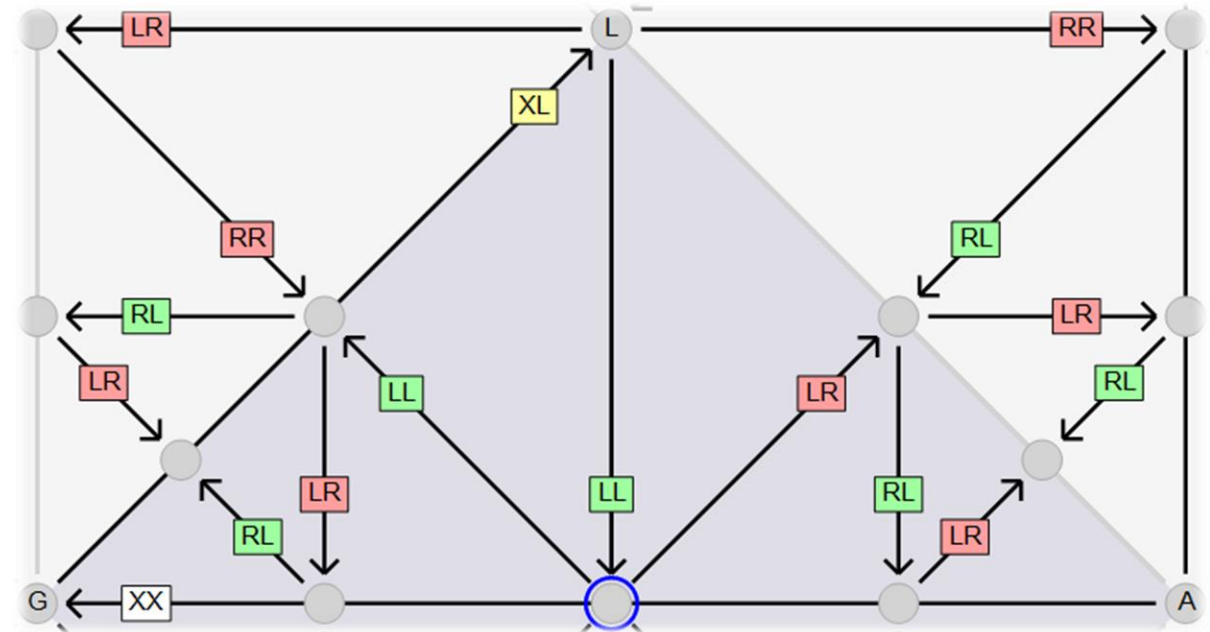
- Take vertex “2” in the figure as V
- Map LL to 0, LR to 1, RL to 2, RR to 3
- Then:
  - “6” is the LL-child of V, child index is 0
  - “10” is the LR-child of V, child index is 1
  - “12” is the RL-child of V, child index is 2
  - “16” is the RR-child of V, child index is 3
- The terminal triangle code also encodes the “X” bits of the child links from “1” / “4” to V.
- Just a nomenclature:
  - LL, LR, RL, RR are “human-readable”
  - 0, 1, 2, 3 are “machine-readable”



# QUESTIONS & ANSWERS – #4

*“Mesh shaders only work on a small group of triangles. How is determined how much work is needed to be done in the CPU to traverse the hierarchy before a group of triangles can be dispatched to be processed in a mesh shader?”*

- Amplification Shaders take CPU-generated sectors (see question #2) and expand them to terminal triangle codes.  
=> N=3 is no guarantee for “always good”  
=> might overflow buffer in extreme cases  
=> will result in missing terrain parts
- “Chicken-and-egg” style of problem: you know how much work to do after having done the work.
- Mesh Shaders expand each terminal triangle code to actual triangles.
- At most N triangles per terminal triangle (for a 9x9 grid, N = 10).



# QUESTIONS & ANSWERS – #5

*“Could he use compute shaders to spawn the mesh shaders instead of using the CPU to do so. I mean, can he use a hybrid Compute / Mesh shader combination and if there is an advantage to do so?”*

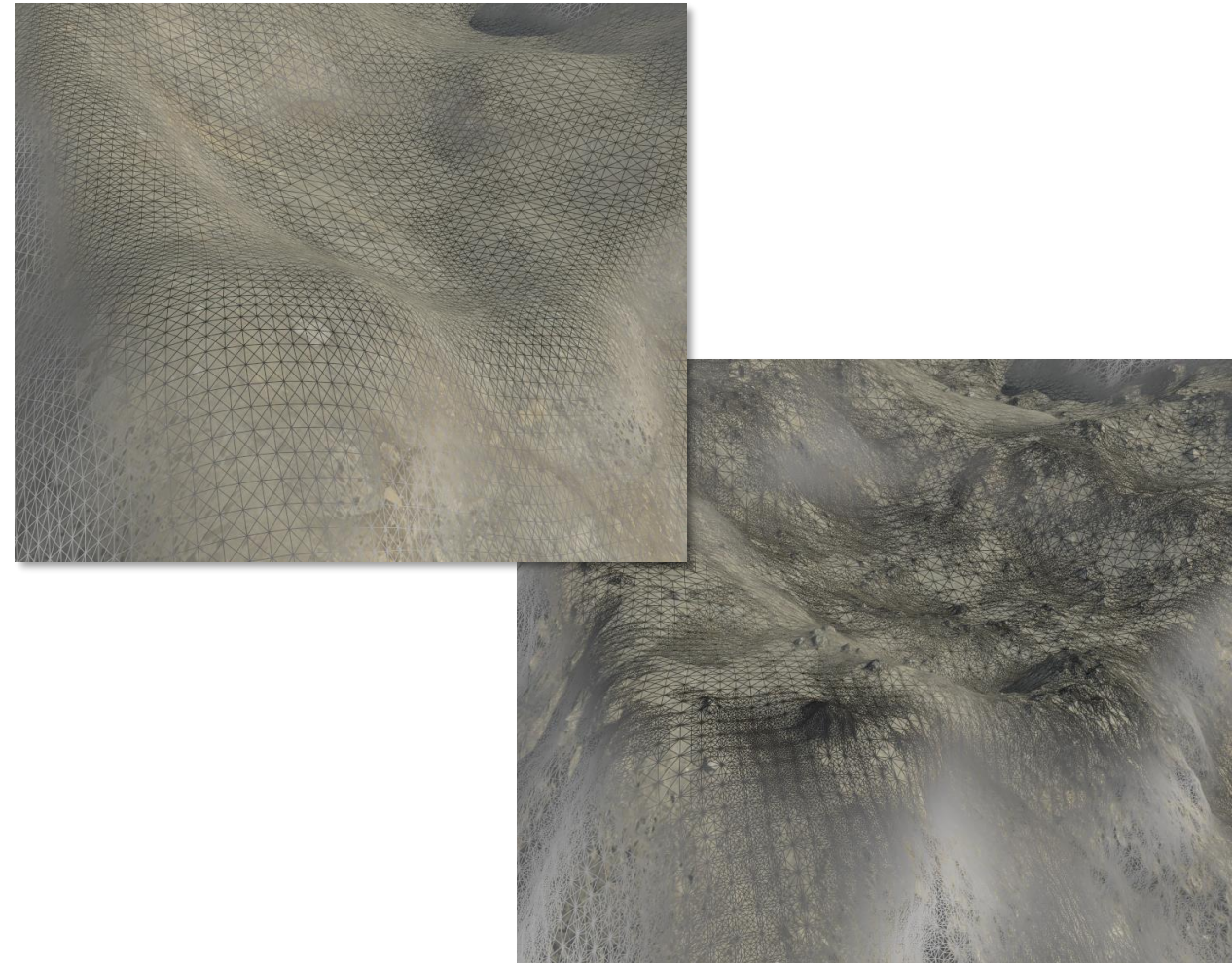
- Using Compute + Amplification + Mesh shaders is an interesting idea.
- The CPU could output just the minimal pre-triangulation (see question #2, ~100..200 sectors).
- Compute Shader could perform “pre-traversal”, in order to determine a “correct” value for N.
- Amplification Shader could consume output of Compute Shader, using the correct N for each sector.
  - or -
  - Compute Shader outputs indirect mesh dispatch commands, bypassing Amplification Shader.
- Sounds like “Best of Both Worlds”...
- Will definitely try to implement this approach 😊



# QUESTIONS & ANSWERS – #6

*“For example, when there is a tall mountain with steep sides, how is the tessellation level determined to avoid stretched triangles? Is the decision done using a view dependent or measuring the area projected into the screen, or is it by just using the vertex to camera distance?”*

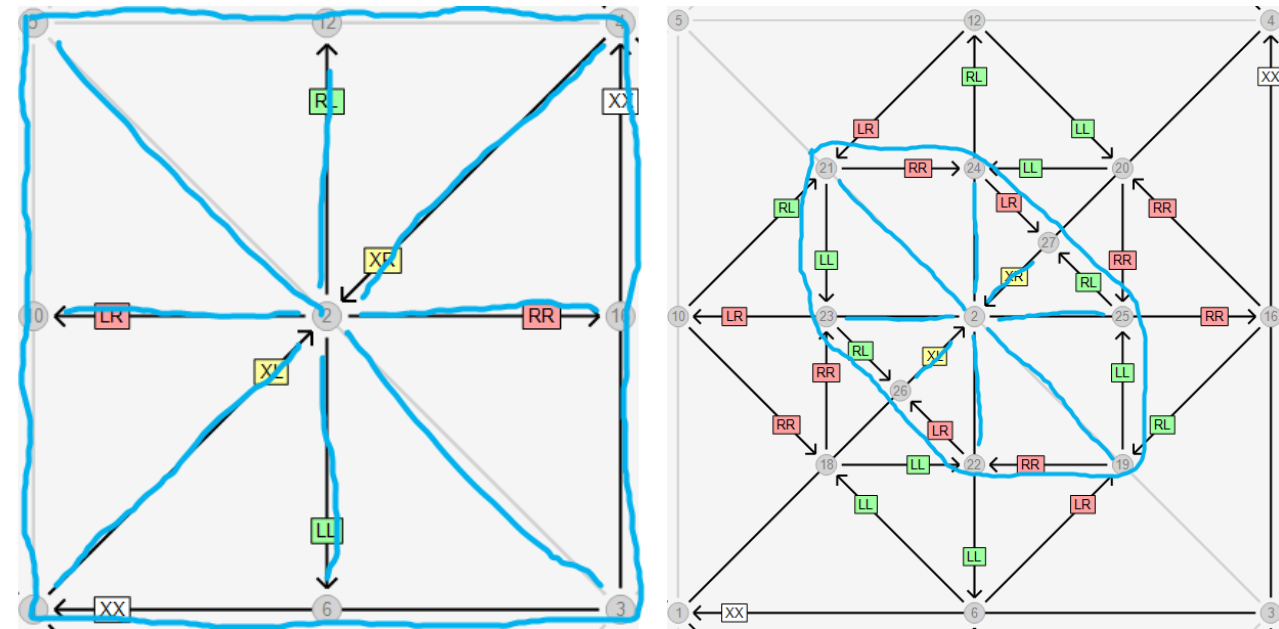
- Both CPU mesh refinement and GPU tessellation are based on projected screen size.
- CPU:  
RTIN subdivision is triggered when projected triangle areas exceed a given quality threshold.
  - “Horizontal” or “real” triangles: area is used to apply quality metric for textures, materials, normal
  - “Vertical” or “virtual” triangles: area is used to measure the geometric error that is introduced by a new vertex.
- GPU:  
Tessellation amount is specified as maximum edge length in screen-space (for maximum texture surface “complexity”), tessellation factors are computed to get the subdivision that gets as close as possible.



# QUESTIONS & ANSWERS – #7

*“When transitioning between levels of detail, the triangulation is changing. The normal seems to be fixed, is this right? At what level of the hierarchy it is computed? When new triangles are introduced, rendering artifacts might appear (e.g. a difference in shading). How you minimize those artifacts? Are you using some type of geo-morphing to smooth out the transitions between levels and how it works with different tessellations forms?”*

- Normal are computed from the vertex “neighbourhood”, must be updated when triangulation changes.
- Compute “base” and “detail” normal per-vertex:  
“base”: using neighbourhood with lowest detail  
“detail”: using neighbourhood with highest detail
- Blend between “base” & “detail” in Vertex Shader



- Project area of “base” neighbourhood to screen-size.
- Project area of “detail” neighbourhood to screen-size.
- Compute blend factor for “base” and “detail” normal.
- Vertex Shader outputs smooth normal vectors; from there, any kind of tessellation (HS/DS/GS) may take over.





## MATTHIAS ENGLERT

### FOUNDER

Fulda (Germany) - Tinman 3D - <https://www.tinman3d.com/>

Matthias Englert is a software enthusiast that grew up programming CGA, EGA and VGA boards - being fascinated by the possibilities of real-time computer graphics.

In 2012, Matthias created the Tinman3D SDK, a CPU-based 3D terrain-pipeline and rendering-engine, which uses modern GPUs for visual output.

Being thrilled by the advent of Nvidia's Turing architecture, Matthias is now working on ways to integrate new GPU features into the SDK, for example Mesh Shaders and Ray Tracing.

To reach out for Matthias, just write him an email: [me@tinman3d.com](mailto:me@tinman3d.com)



## RECORDING POLICY

It is important to recognize that many of the words, images, sounds, objects, and technologies presented at SIGGRAPH are protected by copyrights or patents. They are owned by the people who created them. Please respect their intellectual-property rights by refraining from making recordings from your device or taking screenshots. If you are interested in the content, feel free to reach out to the contributor or visit the ACM SIGGRAPH Digital library after the event, where the proceedings will be made available.

## CREDITS

The images shown in this presentation have been generated with the Tinman 3D software by processing 3<sup>rd</sup> party geodata: “*ASTER Global Digital Elevation Map*” <https://asterweb.jpl.nasa.gov/gdem.asp>, “*EGM96 - The NASA GSFC and NIMA Joint Geopotential Model*” <https://cddis.nasa.gov/926/egm96/egm96.html>, “*Global Multi-resolution Terrain Elevation Data 2010*” [http://topotools.cr.usgs.gov/gmted\\_viewer](http://topotools.cr.usgs.gov/gmted_viewer), “*The National Map - Elevation Layer*” <https://nationalmap.gov/elevation.html>, “*GoogleMaps*” <https://maps.google.de>, “*BingMaps*”, <https://www.bing.com/maps>, “*Elevation1 DSM Technical Information*”, <http://www.intelligence-airbusds.com/en/4367-elevation1>