

Mesh encoding for 4k intros

Iñigo 'iq' Quilez / rgba

(www.rgba.org/iq)

index

- **1 introduction**

- background
- subdivision
- compressing

- **2 quantification**

- principle
- improvements
 - dimensions
 - sampling
- conclusions for 4k

- **3 encoding**

- method 0
- method 1
- method 2
- method 3

- **4 conclusions**



index

- **1 introduction**

- background
- subdivision
- compressing

- **2 quantification**

- principle
- improvements
 - dimensions
 - sampling
- conclusions for 4k

- **3 encoding**

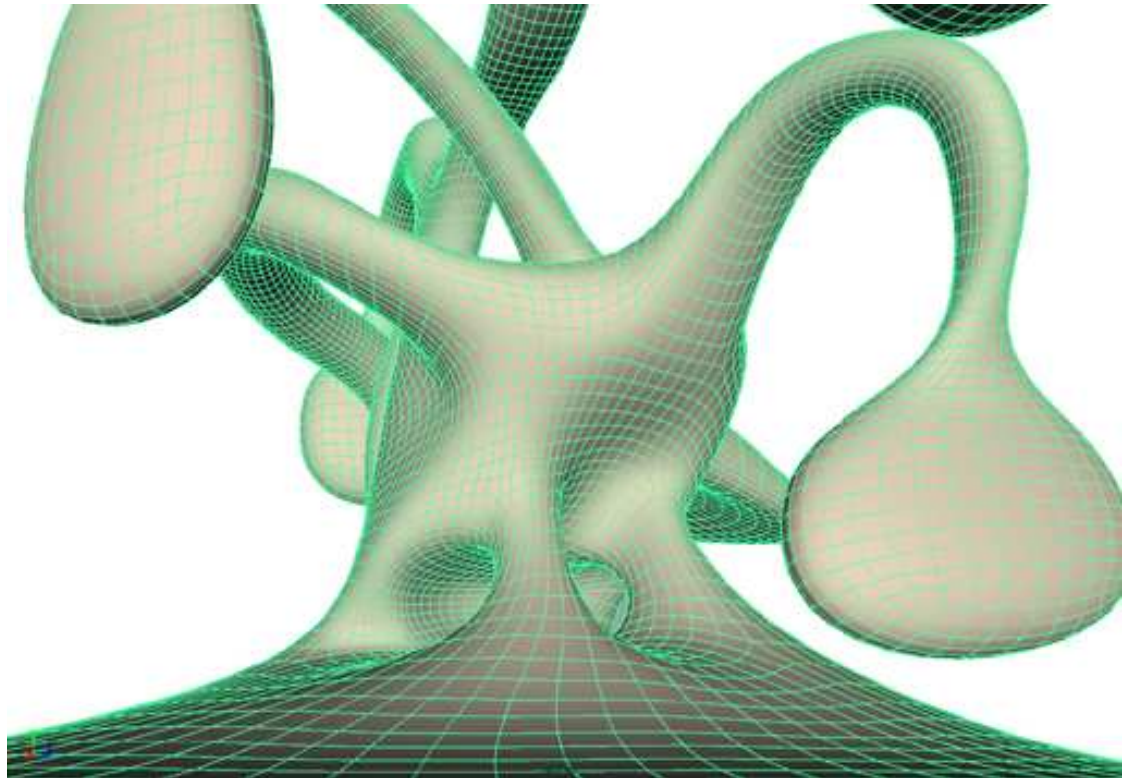
- method 0
- method 1
- method 2
- method 3

- **4 conclusions**



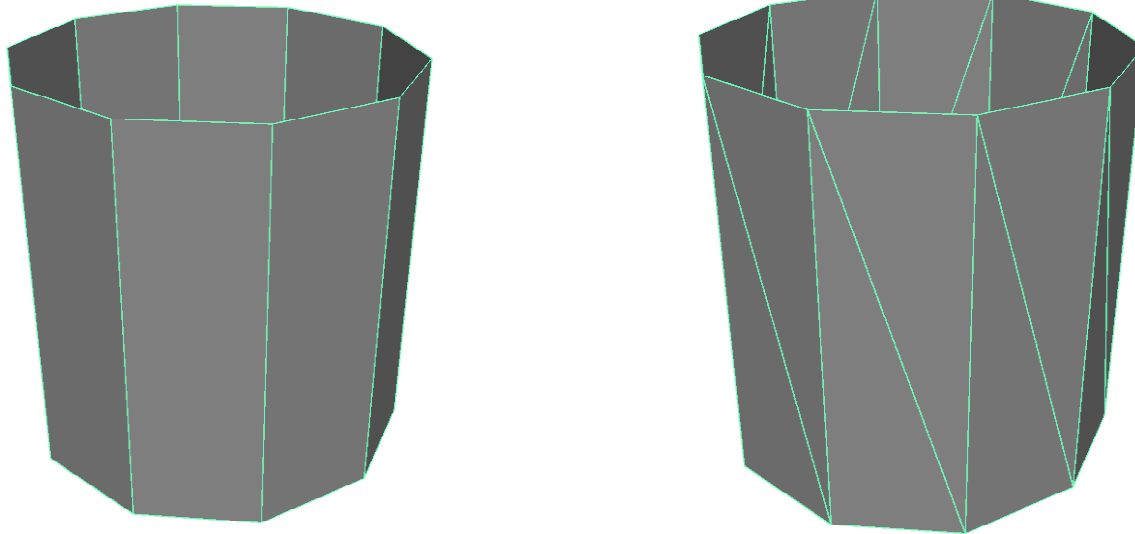
introduction.background

- This is about putting small 3d seed meshes in 4k intros.
- Yes, but *how small?*
 - *Kinderplomber*: 472 tris, 237 verts
 - *Stiletto*: 406 tris, 249 verts



introduction.background

- Well, it's better not to use triangles, but quads.
 - they better represent organic shapes.
 - they don't add useless information as triangles do, and this is important for compression.
 - the only problem is that for some models it's hard to get rid of triangles.



The extra edges introduced by the triangulation doesn't bring any extra shape information in this example, and it rarely does in organic shapes.

introduction.subdision

- *Kinderplomber* used the Catmull-Clark subdivision scheme (see <http://www.rgba.org/iq/divulgation/breakpoint2007/breakpoint2007.htm>), around 430 bytes of code.
 - It matched quite well the subdivision implemented in Maya and Max, what helped the content creation process.
- However *Stiletto* featured more code than *Kinderplomber* (animation, shadow, materials, text) and 430 bytes were too many bytes. Another approach was necessary – a big hack.

introduction.subdision

- *Stiletto* used DX tessellation code (which leads to some ugly rectangular shapes), and applied a cheap software vertex smoothing implemented in around 120 bytes. Problems:
 - of course it didn't work as any subdivision scheme to date, and thus it was impossible to predict from Maya the final shape of the mesh in the intro. The modeling process became a pain.
 - the trick happened to be very sensitive in border mesh edges, and thus they had to be predeformed in quite aggressive ways.
 - this degraded the vertex array compression.
- In future subdivision will be free, done in HW 😊



The sensitivity of border vertices forced to pre distort the model in an unnatural way. Note the self intersections of the mesh.

introduction.compressing

- Things NOT to do:
 - limit yourself to an integer amount of bits (8, 6, 4, 2). Be flexible, use as many as you need. If your values go from 0 to 41, do not try to pack them in 5 or 6 bit. Use bytes and fill them with values from 0 to 41. The arithmetic encoder (Crinkler, Kkrunchy or UPX) will take care of using the proper amount of bits (5.4 in the worst case).
 - pack unrelated data together to avoid “wasting” unused bits. If you have 4 sound volumes (2 bits) and 64 possible notes (6 bits), do NOT be smart and pack them together in 8 bits. Store them in separate byte arrays, the compressor will certainly not use 16 bit, but 8 in the worst case (as with the packing, but now there is more chance for the packer to detect biased distributions).
 - do useless work as using any RLE technique and bit magic. Don't.
- Basically, don't try to be smart packing/compressing data. These guys behind the packers do know the business very well, let their tools do the work, don't put obstacles.

index

- **1 introduction**

- background
- subdivision
- compressing

- **2 quantification**

- principle
- improvements
 - dimensions
 - sampling
- conclusions for 4k

- **3 encoding**

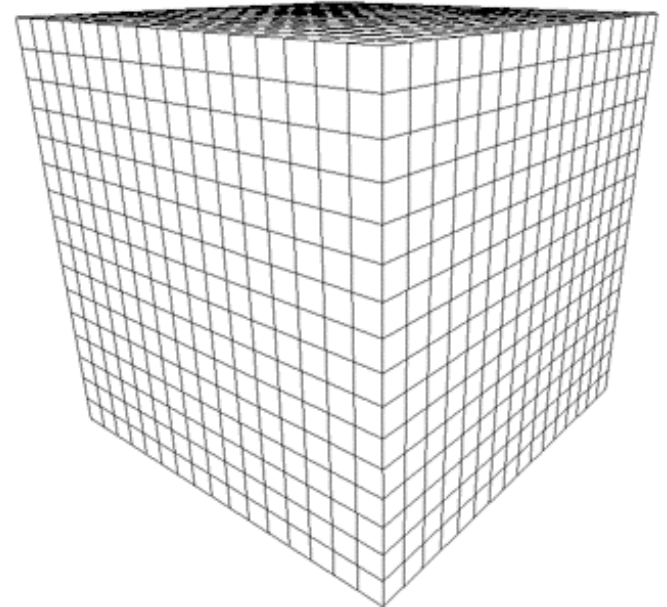
- method 0
- method 1
- method 2
- method 3

- **4 conclusions**



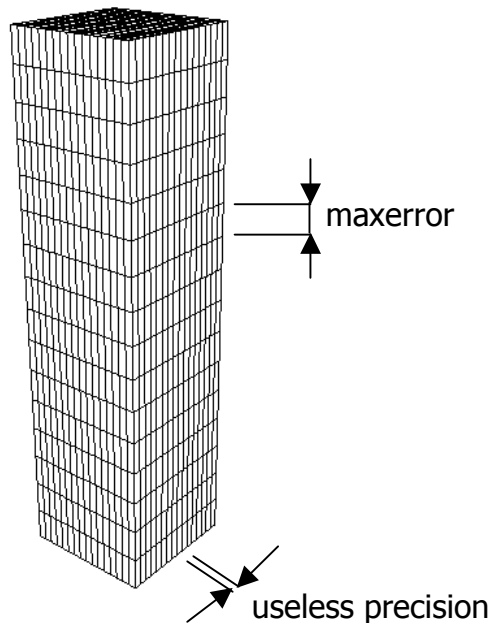
quantification.principle

- First step is quantification, using N bits (for example N=8).
- It reduces the amount of bits (from 32 to N).
- Each dimension is subdivided in $M=2^N$ segments.
- Each vertex is assigned to the closest cell sample.
- Normally a uniform distribution is used:
 - $maxerror = size / 2^{N+1}$
 - so **$N = -1 + \log_2(size / maxerror)$**
 - note N, does not necessarily have to be integer. Let the formula choose the best N based on the size/maxerror ratio, let the arithmetic encoder do the job.
 - for example, the model is 7 meters, the error we tolerate is 1cm, then we need 350 segments, meaning 8.45 bits.

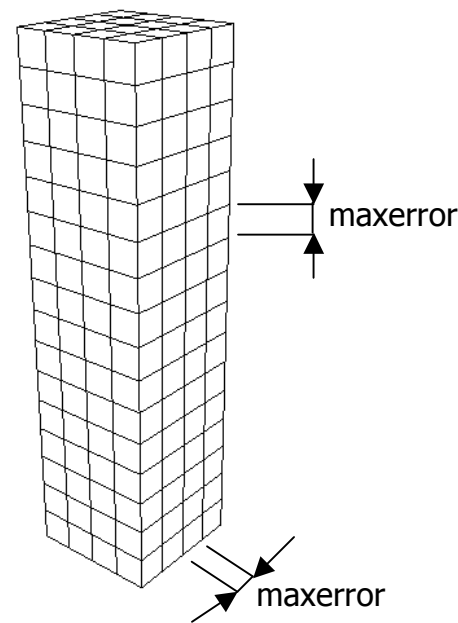


quantification.improvements.dimensions

- Subdivide each dimension in a different amount segments.
- Longer ones need more bits (see previous formula).
- Basically, do not waste bits, sample in uniform cubes.



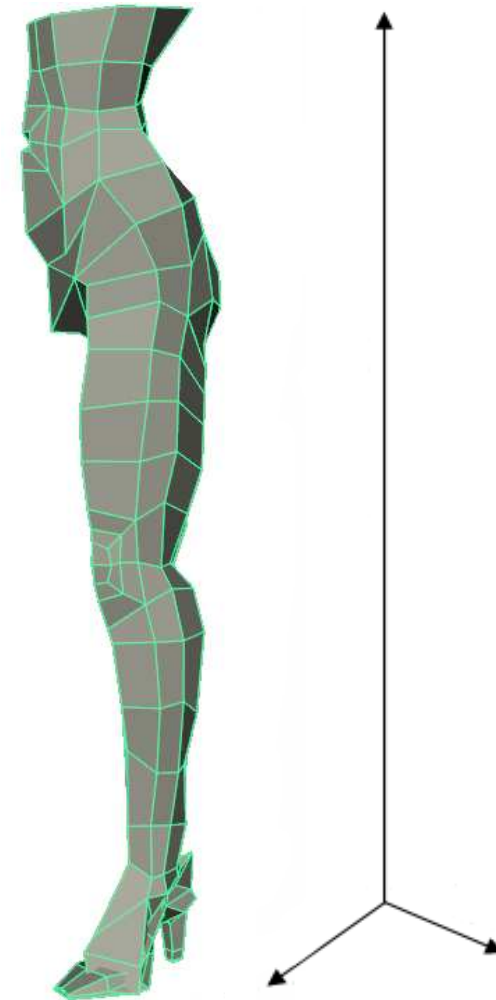
Two of the dimensions are oversampled, and do not contribute to reduce the maximum error of the quantification.



The three dimensions are equally sampled, max error is consistent. This is the best uniform quantification for the given max error.

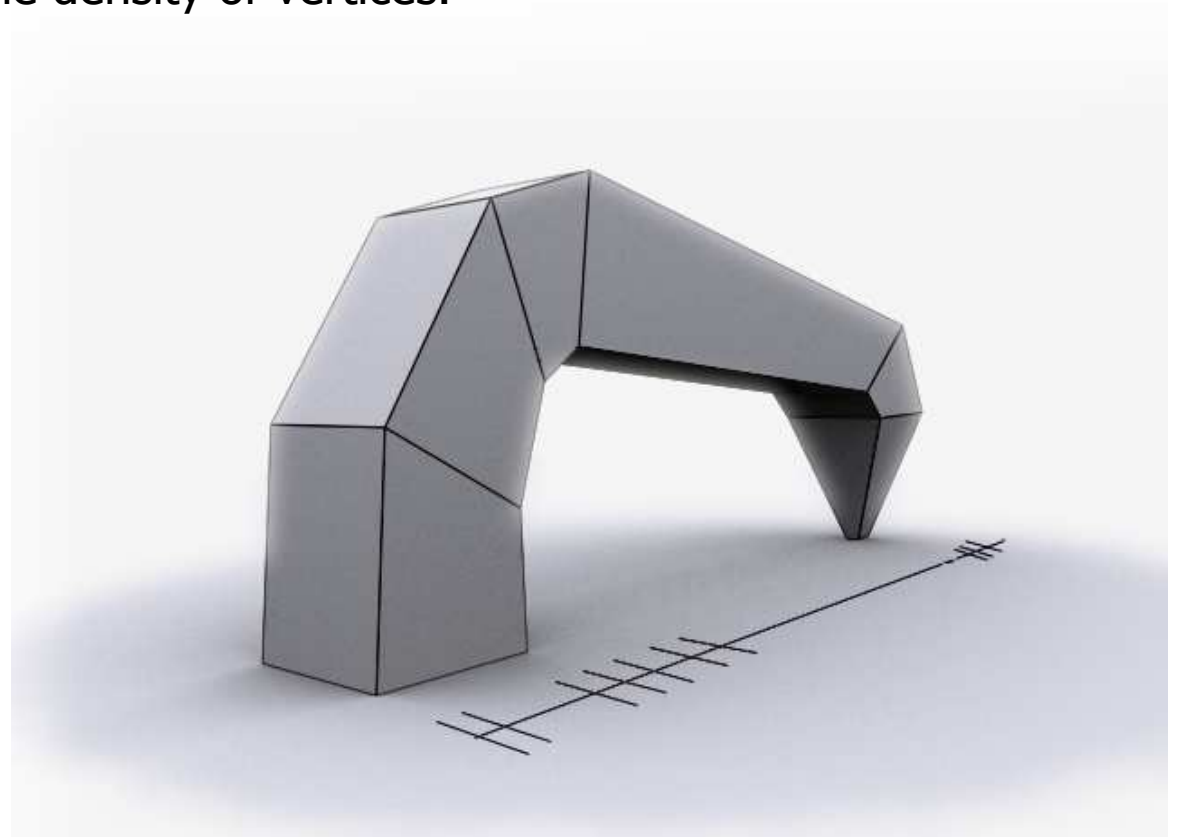
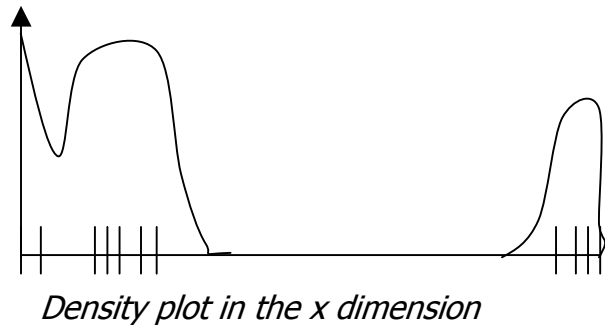
quantification.improvements.dimensions

- Example:
 - maxerr = 0.4 cm
 - bbox = { 47, 110, 41 } cm
 - subdivs = $\text{bbox}/(2*\text{maxerr}) = \{ 59, 137, 51 \}$
 - max bits = { 5.9, 7.1, 5.7 }
 - So, in the worst case we will use **18.7 bits = 2.34 bytes** per vertex. In average it will be less thou - the non uniform distribution of points will allow the entropy encoder assign less bits to the less frequent coordinates.
 - With the traditional approach of 8 bits per axis (por same max error), it would have been **3 bytes** per vertex (**24 bits**).



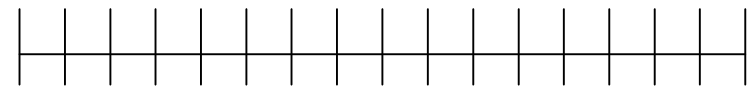
quantification.improvements.sampling

- Non uniform divisions has lot of potential to reduce the amount of segments, and thus the amount of bits.
- Ideally we want to put more segments where vertices are closer together, and less segments where vertices are away from each other.
- Hence, we have to look to the density of vertices.

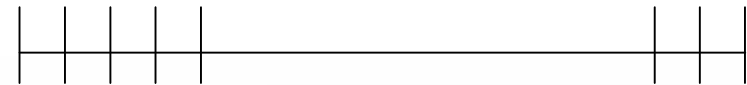


quantification.improvements.sampling

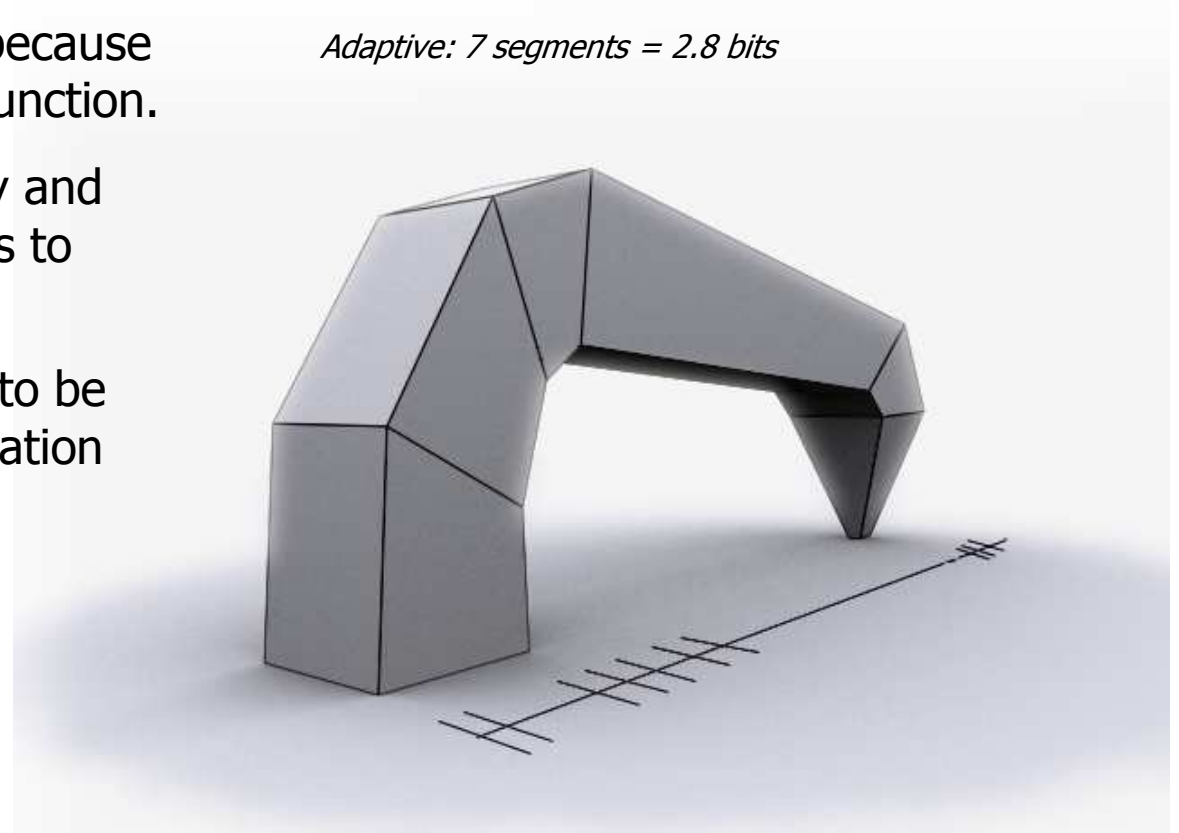
- Putting more segments where density is higher we can save a lot of precious bits.
- With the same max error, ie, same quality.
- We have a bit of overhead because we have to save the density function.
- So, measure the real density and use some parametric functions to match the density.
- This matching doesn't have to be exact, so any cheap approximation will be fine.



Uniform distribution: 16 segments = 4 bits



Adaptive: 7 segments = 2.8 bits



quantification.improvements.sampling

- Density can be also adjusted to subjective criteria, like
 - lower precision for bigger objects.
 - or lower samples to parts of objects we know are far from camera.
 - not always possible to predict of course.
 - or some other heuristic (like u-law for PCM sound).

quantification.conclusions for 4k

- Meshes in 4k intros are very low poly, and density estimations are difficult.
- So far all the low poly meshes for 4k I have used where quite uniform.
- No space for saving density functions.

- So use uniform distribution, based on maxerror decision for each axis (no necessarily in powers of two to have best bit usage).

index

- **1 introduction**

- background
- subdivision
- compressing

- **2 quantification**

- principle
- improvements
 - dimensions
 - sampling
- conclusions for 4k

- **3 encoding**

- method 0
- method 1
- method 2
- method 3

- **4 conclusions**



codification.method 0

- The *Paradise* way:
 - first the x coord of all vertices
 - then y coord of all vertices
 - then z coord of all vertices
 - then for each quad
 - vertexID 0
 - vertexID 1
 - vertexID 2
 - vertexID 3
- Problems:
 - does not explode geometric redundance.
 - does not explode topologic redundance.
- Close to **2.8 bytes/tri** on average for small models...

codification.method 1

- The *Kinderplomber* and *195/95/256* way (brute delta encoding):
 - first the delta x coord of all vertices
 - then delta y coord of all vertices
 - then delta z coord of all vertices
 - then for each quad
 - delta vertexID 0
 - delta vertexID 1
 - delta vertexID 2
 - delta vertexID 3
- Problems:
 - poor geometric redundance utilization.
 - still not good topologic redundance utilization.
- Close to **2.4 bytes/tri** on average for small models...
- Very simple to implement, no more than 50 bytes of additional code.

codification.method 2

- We are not correctly using the fact that topology and geometry are related and thus have some redundant information about each other.
- It would be better if we could compress both at the same time and take benefit of this fact.
- Choose, geometry or topology guided method.
 - geometry guided methods are apparently better, but harder to code.
 - topology guided methods are simple enough for 4k intros, and are quite ok compression wise. Have to choose a strategy.
 - iterative, recursive?
 - face, edge based?
 - ...

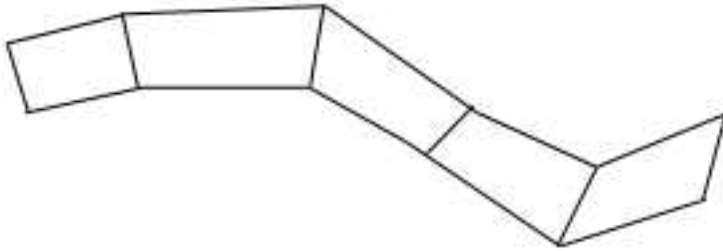
codification.method 2

- The *Stiletto* way, at export time:
 - 1 split the model in smallest amount of quad strips.
 - quads better than tris.
 - strips, compact topology representation: 2 ids per quad vs the usual 4!.
 - 2 start with the longest strip
 - 3 encode strip topology, and as new vertices are visited by the strips, encode them (in a separate data stream of course).
 - apply your preferred geometric compression trick here (more later).
 - 4 when done with the first strip, pick the next (longest) one, and goto 2.

codification.method 2

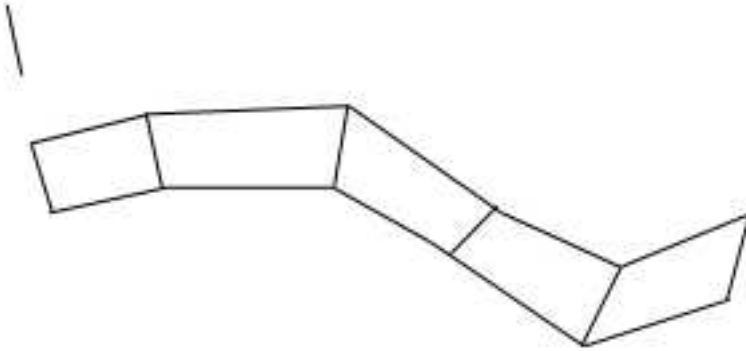
- encoding the quad strips
 - encode vertices in pairs, as in GL_QUAD_STRIP
 - each time we submit a vertex id
 - if vertex already encoded,
 - output 1 in an "visited" array
 - output vertex id in "strips" array
 - if new vertex
 - output 0 in the "visited" array
 - output compressed vertex to "vertices" array (more on this later)
 - no need to output anything else, vertex id can be recovered by the amount of vertices already exported.

codification.method 2



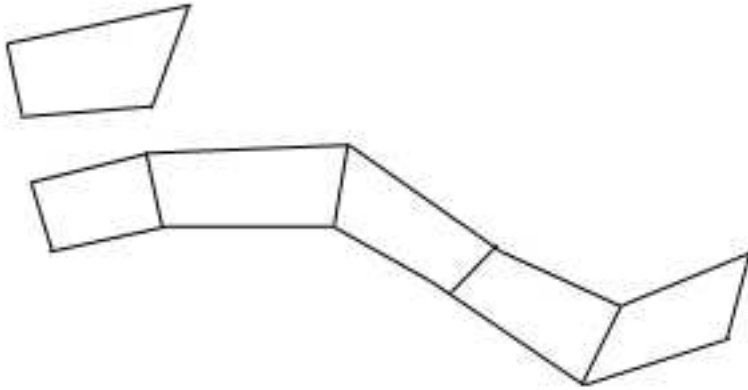
- #exported vertices so far: 12
- **vertices**: { 12 xyz somehow encoded }
- **visited**: { 0,0,0,0,0,0,0,0,0,0,0,0 }
- **strips**: { empty }
- **slenths**: { 5 }

codification.method 2



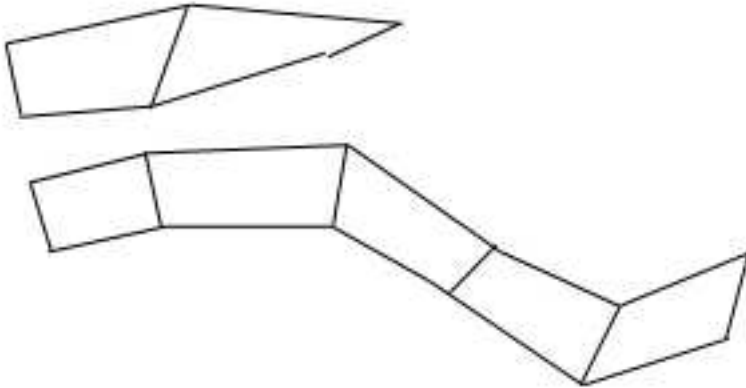
- #exported vertices so far: 14
- vertices: { 14 xyz somehow encoded }
- visited: { 0,0,0,0,0,0,0,0,0,0,0,0,0,0 }
- strips: { empty }
- slenths: { 5 }

codification.method 2



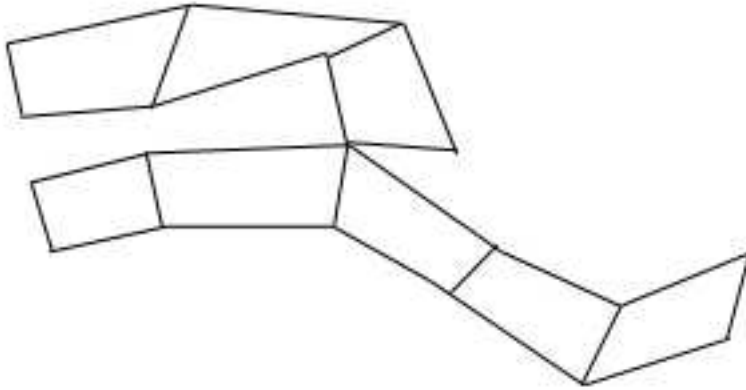
- #exported vertices so far: 16
- **vertices**: { 16 xyz somehow encoded }
- **visited**: { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 }
- **strips**: { empty }
- **slenths**: { 5, 1 }

codification.method 2



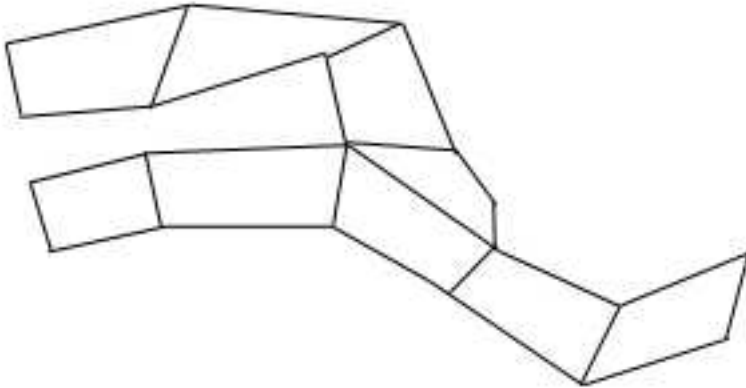
- #exported vertices so far: 18
- vertices: { 18 xyz somehow encoded }
- visited: { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 }
- strips: { empty }
- slenth: { 5, 2 }

codification.method 2



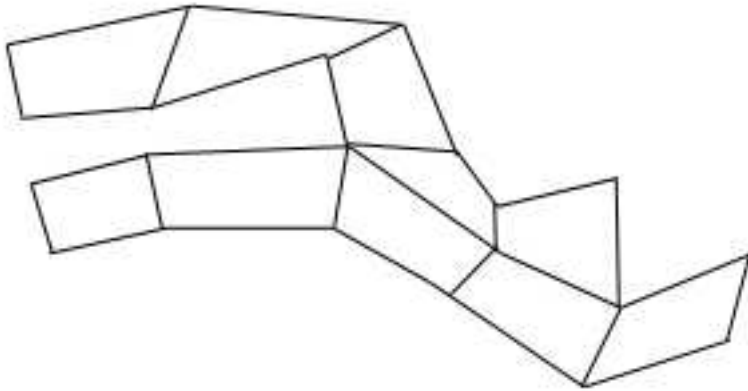
- #exported vertices so far: 19
- **vertices**: { 19 xyz somehow encoded }
- **visited**: { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0 }
- **strips**: { 5 }
- **slenths**: { 5, 3 }

codification.method 2



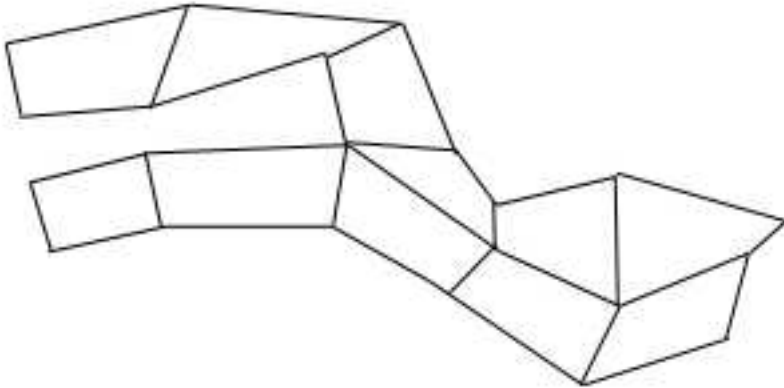
- #exported vertices so far: 20
- vertices: { 20 xyz somehow encoded }
- visited: { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0 }
- strips: { 5, 7 }
- slenths: { 5, 4 }

codification.method 2



- #exported vertices so far: 21
- vertices: { 21 xyz somehow encoded }
- visited: { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0 }
- strips: { 5, 7, 9 }
- slenth: { 5, 5 }

codification.method 2

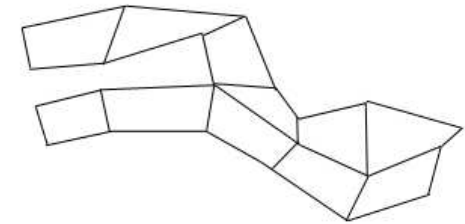


- #exported vertices so far: 22
- **vertices**: { 22 xyz somehow encoded }
- **visited**: { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,1,0 }
- **strips**: { 5, 7, 9, 11 }
- **slenths**: { 5, 6 }

codification.method 2

- Compare this

- **visited**: { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,1,0 }
- **strips**: { 5, 7, 9, 11 }
- **slenths**: { 5, 6 }



- to the naïve method (*method 0*):

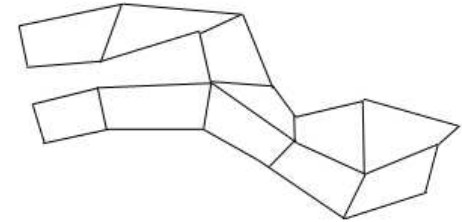
- **quads**: { 0, 1, 2, 3, 2, 3, 5, 4, 4, 5, 7, 6, 6, 7, 9, 8, 8, 9, 11, 10, 12, 13, 15, 14, 14, 15, 17, 16, 16, 17, 4, 18, 18, 4, 6, 19, 19, 6, 8, 20, 20, 8, 10, 21 }

- and to the delta method (*method 1*):

- **quads**: { 0, 1, 1, 1, -1, 1, 2, -1, 0, 1, 2, -1, 0, 1, 2, -1, 0, 1, 2, -1, 2, 1, 2, -1, 0, 1, 2, -1, 0, 1, -13, 14, 0, -14, 2, 13, 0, -13, 2, 12, 0, -12, 2, 11 }

codification.method 2

- Not only topology is better compressed, but we ensure a coherent vertex traversal.
- This will help all kind of vertex codification tricks as prediction.
- Remember on *"method 0"* we were delta encoding without any geometric consideration.
- Now deltas are most probably going to be small, at least within the same strip (more on this later).
- So, time for vertex codification!



codification.method 2.final

- Finally, the *Stiletto* way:
 - Quad strip guided compression.
 - visited, strips and slenths arrays.
 - Brute force coded vertices ☺

```

static unsigned char stilettoWoman[] = {
// numStrips
27,
// strip lengths
27, 26, 16, 16, 12, 12, 11, 9, 9, 7, 7, 7, 7, 5, 4, 4, 4, 3, 3, 2, 2, 2, 2, 2,
2, 1, 1,
// verts X
60, 36, 62, 38, 63, 35, 53, 34, 46, 33, 46, 35, 47, 34, 50, 35, 55, 36, 58, 39, 45, 36, 44, 35,
45, 35, 47, 56, 25, 11, 22, 24, 26, 24, 20, 21, 25, 21, 21, 7, 8, 1, 0, 0, 2, 10, 12, 19,
19, 20, 18, 25, 29, 25, 22, 26, 41, 40, 41, 45, 52, 50, 46, 49, 43, 50, 50, 51, 48, 40, 39, 6,
4, 0, 27, 22, 23, 23, 22, 20, 33, 36, 44, 44, 44, 46, 47, 46, 51, 48, 50, 48, 48, 46, 39, 39,
33, 33, 27, 28, 20, 23, 19, 23, 20, 21, 35, 20, 23, 33, 43, 46, 45, 54, 52, 40, 33, 26, 15, 15,
25, 23, 34, 33, 48, 46, 44, 44, 40, 33, 28, 24, 25, 23, 0, 0, 1, 7, 1, 6, 38, 29, 60, 61,
57, 63, 44, 54, 37, 47, 16, 24, 6, 10, 1, 0, 0, 0, 14, 13, 25, 25, 38, 34, 33, 30, 22, 23,
13, 14, 4, 4, 2, 2, 0, 0, 0, 16, 28, 50, 41, 25, 15, 4, 2, 0, 0, 0, 11, 18, 26, 30,
30, 56, 39, 49, 26, 42, 16, 8, 0, 15, 0, 0, 0, 8, 14, 20, 22, 38, 27, 41, 31, 29, 24, 15,
13, 0, 0, 0, 22, 41, 46, 41, 18, 5, 0, 32, 37, 33, 27, 33, 33, 33, 36, 33, 30, 32, 32, 29,
33, 36, 0, 0, 0, 0, 0, 0, 33, 32,
// verts Y
191, 190, 175, 176, 157, 155, 127, 129, 123, 124, 116, 115, 107, 105, 104, 97, 94, 93, 77, 76, 43, 42, 28, 32,
23, 14, 13, 0, 14, 0, 23, 28, 43, 74, 93, 101, 106, 113, 121, 123, 146, 172, 188, 177, 165, 143, 122, 111,
113, 103, 94, 73, 42, 29, 26, 20, 20, 26, 28, 41, 73, 94, 106, 115, 113, 127, 149, 165, 170, 188, 173, 188,
180, 188, 11, 14, 18, 17, 21, 24, 26, 30, 22, 24, 19, 21, 11, 14, 9, 11, 2, 4, 1, 4, 0, 1,
0, 1, 0, 1, 1, 4, 3, 5, 7, 11, 12, 16, 19, 20, 19, 16, 7, 0, 0, 0, 0, 0, 0, 0,
7, 4, 3, 7, 4, 10, 12, 17, 21, 22, 21, 17, 12, 10, 177, 178, 175, 165, 172, 166, 186, 172, 190, 168,
191, 169, 199, 169, 188, 170, 187, 171, 186, 168, 191, 196, 227, 236, 227, 236, 227, 232, 224, 230, 224, 229, 220, 229,
221, 230, 221, 224, 221, 221, 221, 221, 202, 202, 201, 200, 192, 186, 214, 219, 220, 220, 205, 191, 205, 194, 207, 188,
212, 201, 216, 206, 217, 208, 221, 199, 230, 196, 173, 237, 219, 236, 219, 237, 218, 238, 217, 238, 218, 238, 220, 239,
224, 239, 223, 253, 255, 255, 254, 253, 252, 251, 250, 14, 13, 12, 13, 0, 0, 7, 3, 3, 3, 3, 0, 0,
0, 0, 219, 214, 184, 224, 233, 5, 0,

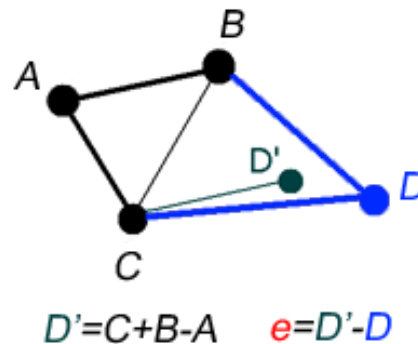
```


codification.method 3

- The idea is to reduce the range and variance of the coordinates.
 - the smaller the range, the less bits needed to represent it.
 - the smaller the variance, the better the entropy code will do.
 - the worst thing is when the distribution is uniform, white noise.
- A proven efficient way to do so is to try to predict what incoming data will be based on already encoded data, and store only the error (coder and decoder have to use the same predictor of course).
 - used in PNG images, voice in mobile phones, ...
- Two options:
 - use geometric prediction (basically, predicting data -where vertices will be- based on the fact we know we are encoding vertices of a 3d mesh)
 - use generic data prediction.

codification.method 2.geometry

- Geometric prediction:
 - delta coding can be seen as geometric prediction too: we assume next vertex to be next to the last one... what is obviously wrong, but at least we ensure our prediction is not too bad...
 - except when we change from one strip to another.
 - parallelogram rule: assume last three vertices and next one will form a perfect rombe. Predict where this new vertex will be to form such a shape:



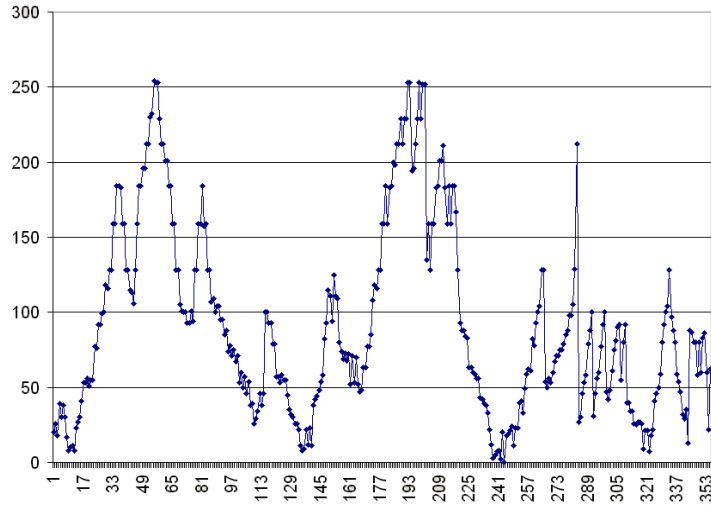
- Normally very good estimator for high polycount meshes. However not that good with low poly meshes...

codification.method 2.geometry

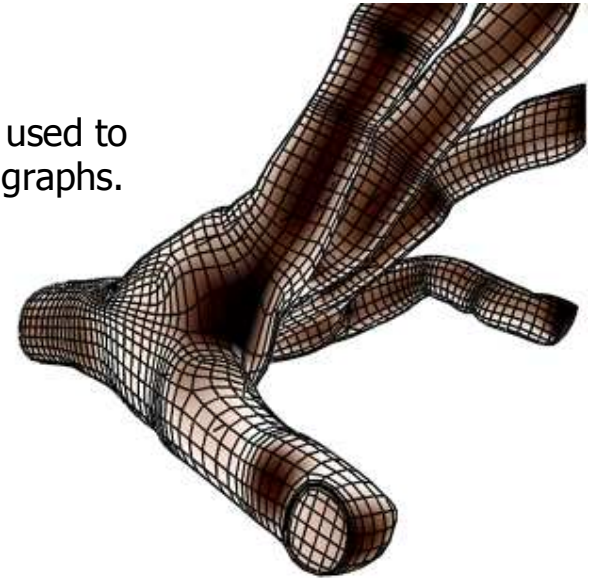
- Generic data prediction:
 - if we assume data changes slowly, we can generalize delta coding:
 - $e[n] = x[n] - x[n-1]$
 - LPC, more general linear: combination of the last N encoded data samples
 - **$e[n] = a_0x[n] + a_1x[n-1] + a_2x[n-2] + a_3x[n-3] + \dots$**
 - (note, parallelogram rule can be seen as such a generalization too)
 - the trick is to find the coefficients a_n that minimize the variance of $e[n]$ (or in other terms, the energy of the signal $e[n]$)
 - to do so you attend telecom courses and dig into covariance matrix stuff, or just use the Yule-Walker equations ☺
 - this method is used by your mobile phone thousands of times per second (and Candytron ;))
 - store the coefficients a_n next to your vertex data.

codification.method 2.geometry

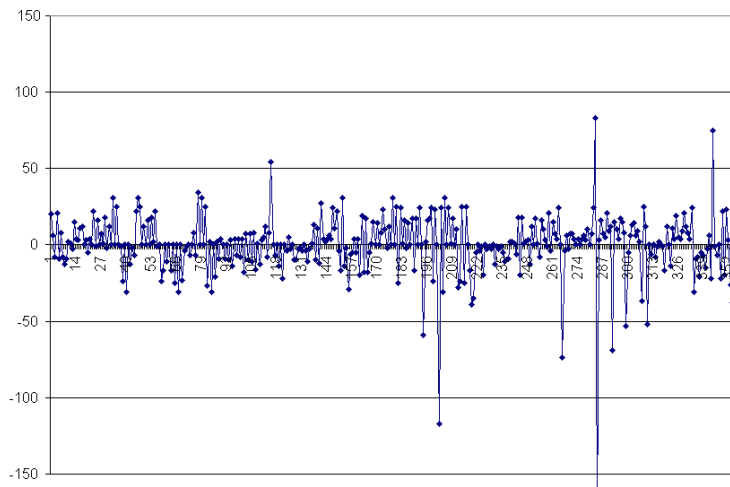
X, high variance



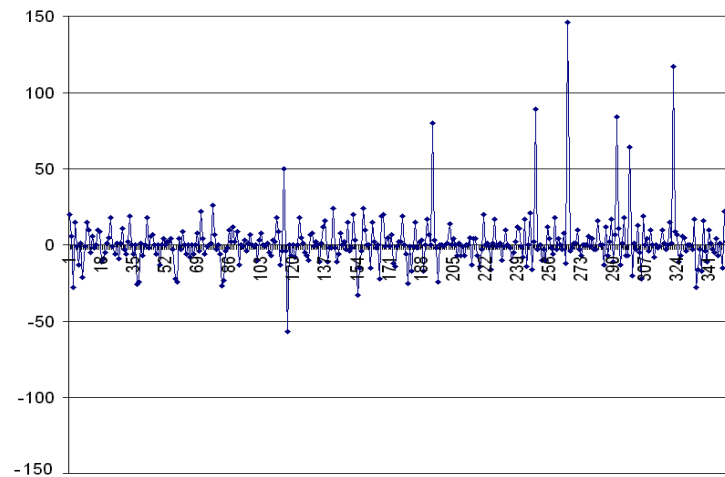
hand mesh used to extract the graphs.
353 verts.



delta X, much lower variance -> less bits!

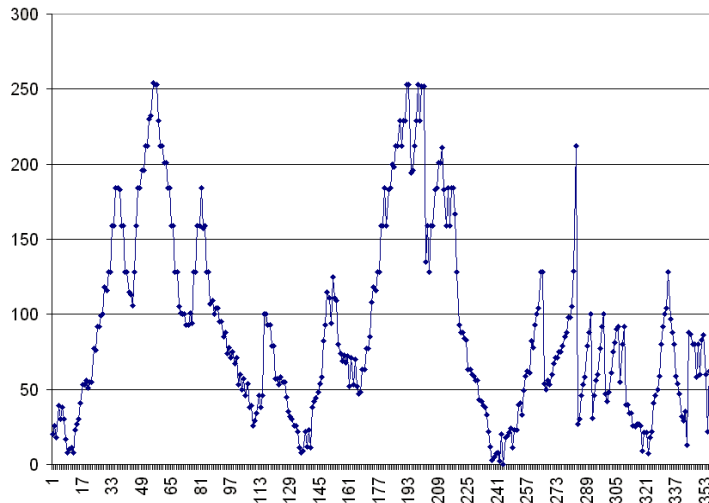


LPC error X, even lower variance!

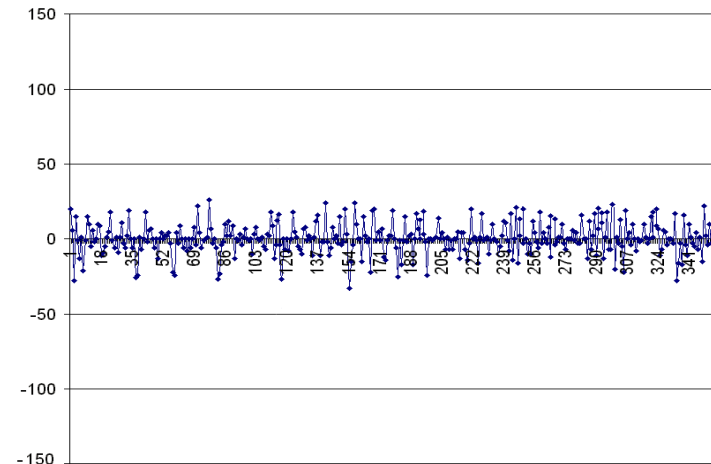


codification.method 2.geometry

- The peaks on the second and third graph are due to the fact that when a new strips starts, it's vertices cannot be well predicted from last strips vertices of course.
- Solution would be to start a new prediction process for each strip, and store the first vertex in a separate array.
 - similar to the DC component on a .JPG image compression (and other DCT based systems – MP3).



Range: 255, compressed to 7.2 bit approx.



Range: 70, compressed to 5 bit approx.

index

- **1 introduction**

- background
- subdivision
- compressing

- **2 quantification**

- principle
- improvements
 - dimensions
 - sampling
- conclusions for 4k

- **3 encoding**

- method 0
- method 1
- method 2
- method 3

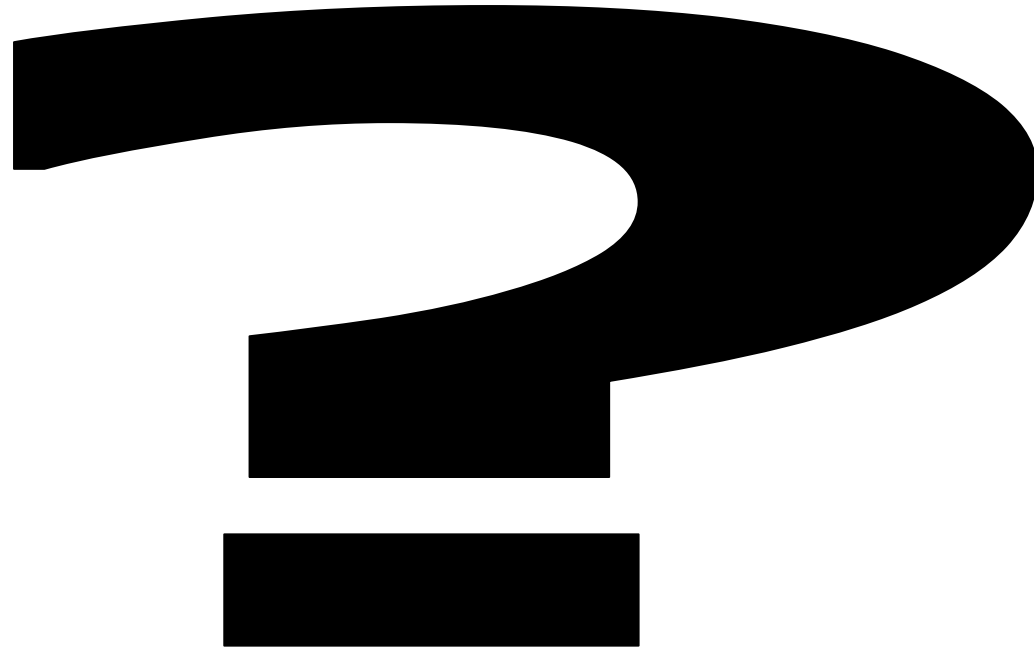
- **4 conclusions**



conclusions

- Some stats:
 - Method 1: *Kinderplomber*: 967 (2.04 bytes/tri) (237v, 236q)
 - Method 2: *Stiletto*: 710 bytes (1.75 bytes/tri) (249v, 203q)
 - Method 3: Hand: 1345 bytes (1.91 bytes/tris) (358v, 353q)
- (note, comparisons are not completely fair, numbers depend a lot on the model)
- method 3 is superior, but it doesn't worth the effort in 4k (you lose in code what you win in data...)
- Should we forget subdivision and directly encode higher polycounts?
 - we skip the subdivision code overhead... (but this will be free in two years because of the HW...)
 - we have more polys, but vertices become more easy to predict...
- In general, do not be too smart when it comes to pack bits.

Questions



(simples, please...)