

Tricks and techniques for rgba's past and future intros

Iñigo 'iq' Quilez / rgba



index

- **1 General intro system**

- The tools
- The pipeline
- The code
- The implementation
- The content
 - Textures
 - Mesh animation

- **2 Interesting techniques used**

- Ambient Occlusion
- Mesh generation
 - Procedural
 - Compression
 - Other tricks

- **3 Techniques for the present/future**

- Mesh compression revisited
- Collashing
- Automatic skinning
- Curves
- Cloth
- Hair

- **4 Tip & tricks**

- Normals for quads
- Normalizing a mesh
- Carmull-Clark subdivision in 50 lines

- **5 Conclussions**

General intro system

• 1 General intro system

- The tools
- The pipeline
- The code
- The implementation
- The content
 - Textures
 - Mesh animation

• 2 Interesting techniques used

- Ambient Occlusion
- Mesh generation
 - Procedural
 - Compression
 - Other tricks

• 3 Techniques for the present/future

- Mesh compression revisited
- Collashing
- Automatic skinning
- Curves
- Cloth
- Hair

• 4 Tip & tricks

- Normals for quads
- Normalizing a mesh
- Carmull-Clark subdivision in 50 lines

• 5 Conclusions

the tools

- Basically, no tools other than:
 - a small 3DSMax exporter to save one object to a text file.
 - and Visual Studio, our big editor.
 - Messenger, Photoshop, VirtualDubMod,... (these are crucial for the pipeline).
- Everything is scripted... in C 😊
 - Textures are created by editing “formulas”. Press F5 for feedback.
 - Shaders are written in the C code also. F5 for feedback.
 - All animations are procedural, and programmed in pure C. F5 again...
 - Scenes are built in C. Press F5 to see what you created.
 - Meshes are generated stacking modifiers in C language. F5 for feedback.
- We do everything with formulas. We suffer the “The Matrix” syndrome.

the tools

- Pros
 - No need to spend time developing tools, and fixing, and customizing, and
 - It's flexible - if you need something or want to try a new idea, just make a small func right there; and if it doesn't work, just remove it. No need to recompile tools or make a new plugins.
 - No need to keep GUI tool up to date with the algorithms/technology.
 - No "After all this work, we should amortize the tool with several (probably similar looking) intros".
 - Artists do not like compilers, so the coder `_is_` needed.
 - Paradise: 100 textures, 70% objects and 60% cameras by coder.
- Cons
 - It can be a pain to get some results (specially for artists)
 - Complex and frustrating to polish the end result.
 - Really slow to develop an intro (not a problem depending the pretensions).

the pipeline

- Main scene setup
 - 1. Coder programatically creates a rough scene (objects, textures)
 - 2. Send screenshot to artist by mail.
 - 3. Artist photoshops on top of it how it should look like.
 - 4. Coder fixes
 - if not happy, goto 1.



Given the deer mesh, the coder creates the rest.



After photoshopping, artis decides some tress would make it "look cool".



So coder makes quickly hacks some code to generate some trees.

the pipeline

- Camera setup
 - 1. Coder creates as many cameras as possible with formulas
 - 75 % in Paradise
 - 40 % in 195/95/256
 - 2. When an artist is _really_ needed, export scene from the intro.exe to .obj.
 - 3. Send by mail, and sleep for 2 days.
 - 4. Artist creates the cameras (more or less blindly if objects move!).
 - 5. Coder exports splines to .txt, and then to .cpp.
 - 6. Coder modifies code to use the new splines.
 - 7. If not happy, manually correct the spline, or procedurally modify it.
 - 8. If still not happy,
 - Make .avi to show the problem.
 - Goto 2

the code

- We need to store data with the compressor in mind (same type data items together whatever the source is).
- But at the same time we need to keep it easy to edit (related data together, whatever the type).
- So, we sacrifice a bit compressibility for a robust and coherent structure.
- Generic code is well separated from intro-specific data/code (C files).
- All modifiers/operators/textures/shaders/types/entities are referenced with #defines (maintained by hand, but an intro doesn't have more than few hundreds anyway) so the final look is like:

```
static uint8 commands[] = {  
    4, C_QLOAD, C_MAPEA,      C_SCALE, C_TRANSLATE,          // 00_0. bicho!  
    5, C_RLOAD, C_DISPLACE, C_SCALE, C_ROTATE, C_TRANSLATE,  // 00_1. col2  
    3, C_QLOAD, C_MAPEA,      C_SCALE,                      // 00_5. petalo  
    2, C_QLOAD, C_GENLOWTEX,                                     // 00_6. glow quad  
    //-----  
    3, C_QLOAD, C_CLONE, C_SCALE, /*C_TRANSLATE,*/          // 01_07. tubos  
    2, C_QLOAD, C_GENLOWTEX,                                   // 01_09. quad glow  
    6, C_QLOAD, C_DISPLACE, C_SCALE, C_ROTATE, C_MAPEA, C_TRANSLATE, // 01_10. reloj  
    5, C_RLOAD, C_DISPLACE, C_SCALE, C_ROTATE, C_MAPEA,      // 01_11. bipite gordo
```

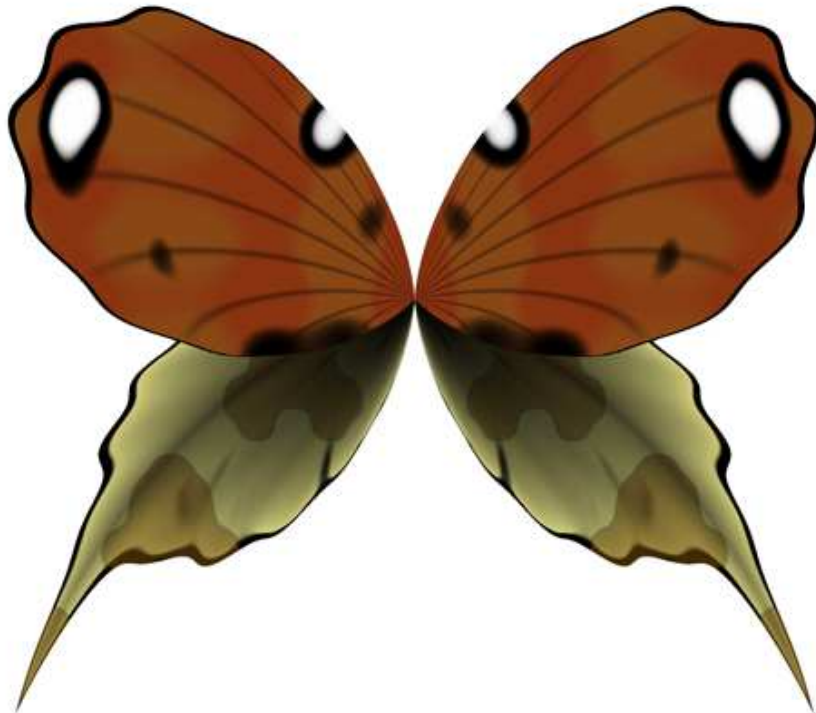

the implementation

- We don't have any material system.
- We don't have "lights" in the scenes.
- We only have shaders.
- One object one shader more or less (easy to find 50 shaders in our intros)
- No single shading model / generic-shader,
 - We square the $N \cdot L$, or cube root the AO, or do $N \cdot (L + (0, .5, 0))$ or even $|N \cdot L|$ if it helps to make the thing look better.
- Each shader customizes the lighting/shading regardless what other shaders do. We just don't care about lighting models.
- However, most shaders use same name for variables, and have the same structure, so they compress incredibly well.
- The only drawback could be amount of state switches, but demos/intros do not have many objects anyway.

the content

content.textures

- Textures are (resolution independent) formulas.
- For each (x,y) get a color (r,g,b,a)
- Quick to build textures; no buttons to press, no boxes to connect/stack. Just type few lines (and optionally press a key to see).



- Useful math functions:
 - To create patterns: `sin()`, `cos()`, `fmod()`, `noise()`
 - To define shapes: `step()`, `sstep()`
 - To tune colors and shapes: `pow()`, `sqrt()`
 - Others: `exp()`, `abs()`

content.textures

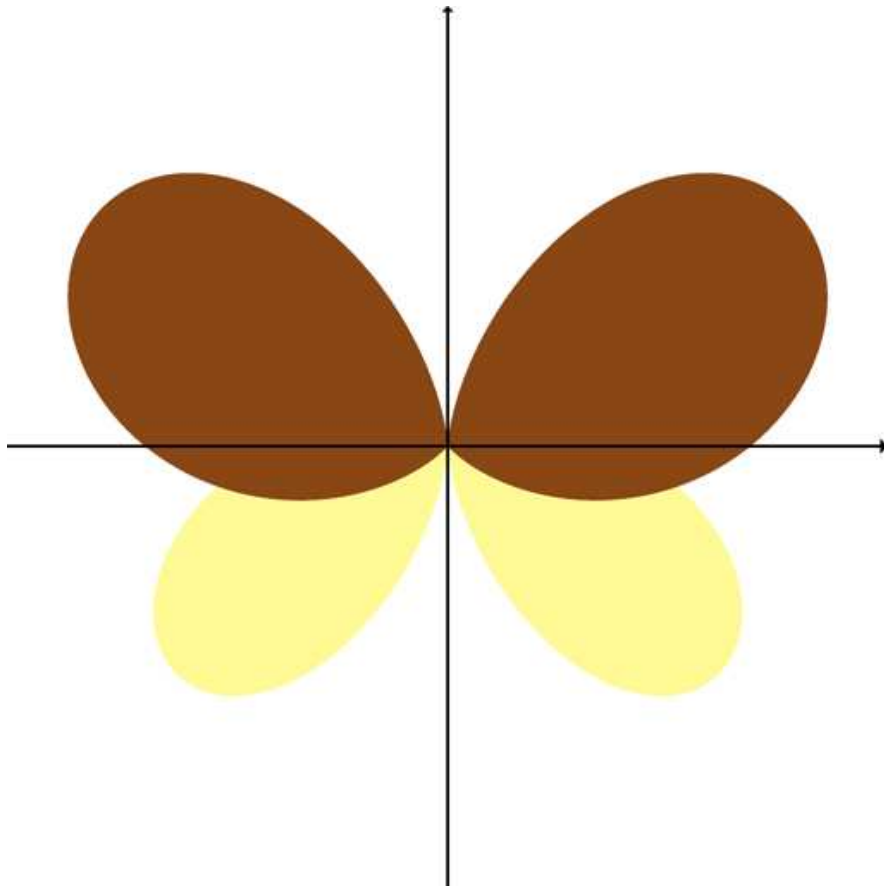
- First thing to do is to detect the main features of we want to mimic. For example:

- Two main lobe shaped wings with small irregularities in the border.
- A “ring” of darker color along this border.
- Radial “dark lines”.
- Symmetric color patterns, often similar to eyes
- Smaller “random” subtle color variations.



content.textures

We start with a coarse approximation



$$\beta = \max\left(\frac{3|\alpha| - \pi}{2}, 0\right)$$

$$\gamma = \min\left(\frac{4|\alpha|}{3}, \pi\right)$$

$$th = \frac{1}{2} \sin^2 \beta$$

$$th' = \frac{1}{2} \sin^2 \gamma$$

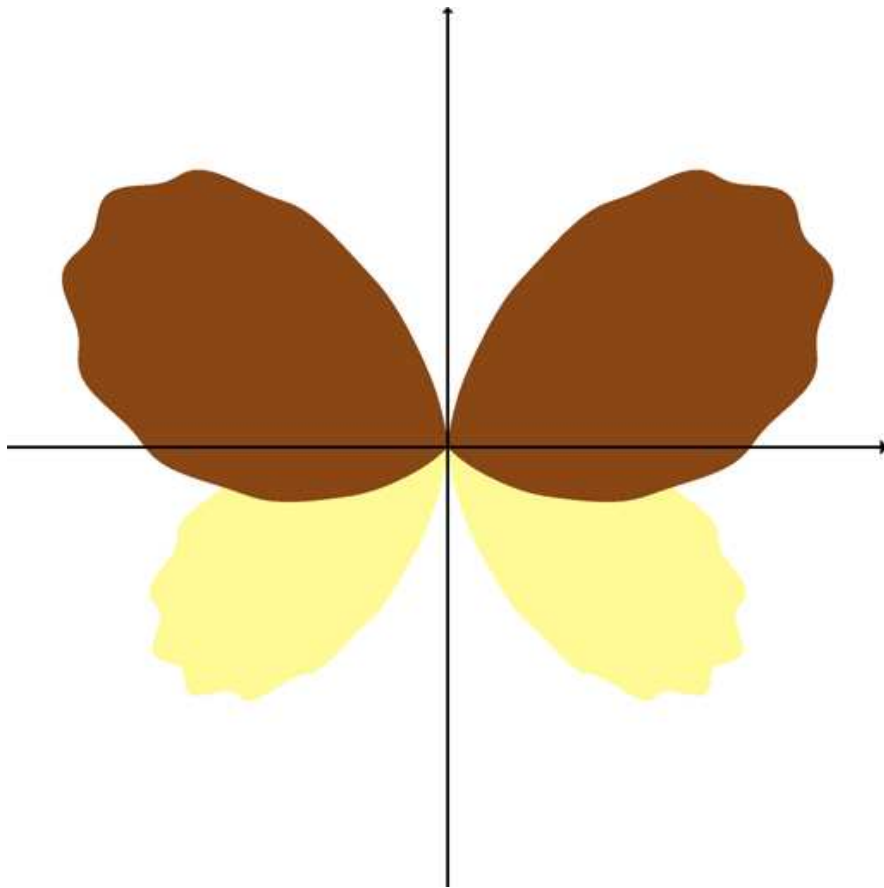
$$c1 = \text{yellow}$$

$$c2 = \text{red}$$

$$\text{color} = \text{lerp}(c1 \cdot \text{step}(th, r), c2, \text{step}(th', r))$$

content.textures

...and then we go for the details



$$\beta = \max\left(\frac{3|\alpha| - \pi}{2}, 0\right)$$

$$\gamma = \min\left(\frac{4|\alpha|}{3}, \pi\right)$$

$$th = \left(\frac{1}{2} + \frac{1}{100} \sin(43\alpha)\right) \sin^2 \beta$$

$$th' = \left(\frac{1}{2} + \frac{1}{100} \sin(24\alpha)\right) \sin^2 \gamma$$

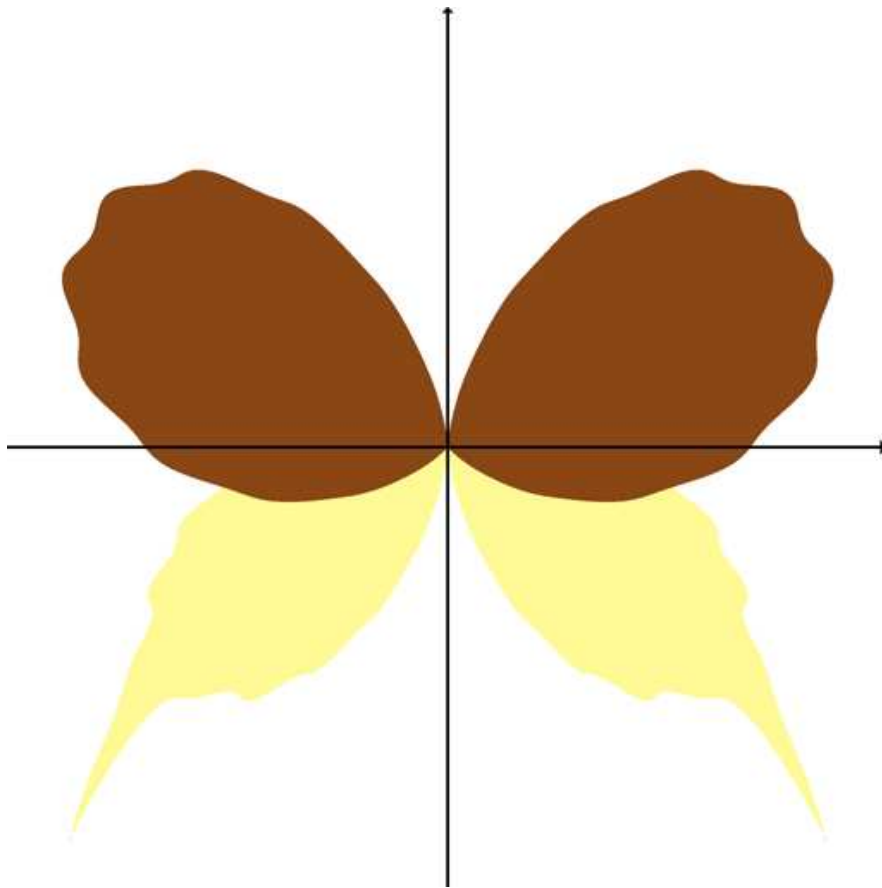
$$c1 = \text{yellow}$$

$$c2 = \text{red}$$

$$\text{color} = \text{lerp}(c1 \cdot \text{step}(th, r), c2, \text{step}(th', r))$$

content.textures

...and then we go for the details



$$\beta = \max\left(\frac{3|\alpha| - \pi}{2}, 0\right)$$

$$\gamma = \min\left(\frac{4|\alpha|}{3}, \pi\right)$$

$$th = \left(\frac{1}{2} + \frac{1}{100} \sin(43\alpha)\right) \sin^2 \beta + \frac{1}{4} e^{-30 \cdot \left|\alpha\right| - \left(\frac{4\pi + 5r}{7}\right)}$$

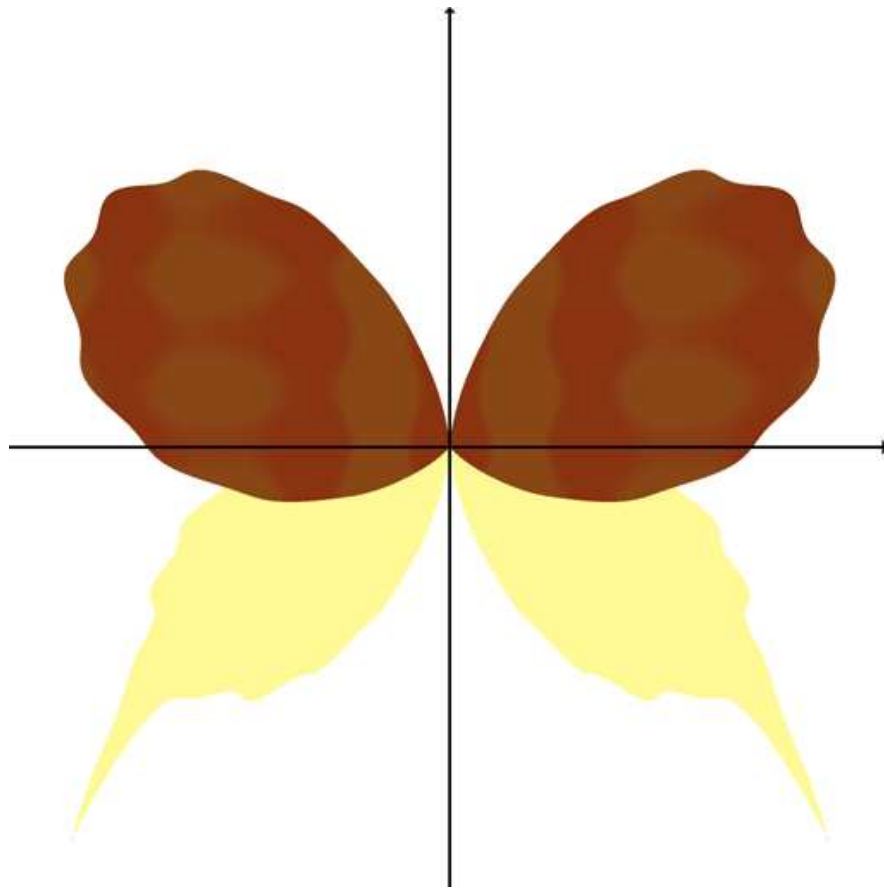
$$th' = \left(\frac{1}{2} + \frac{1}{100} \sin(24\alpha)\right) \sin^2 \gamma$$

$$c1 = \text{yellow}$$

$$c2 = \text{red}$$

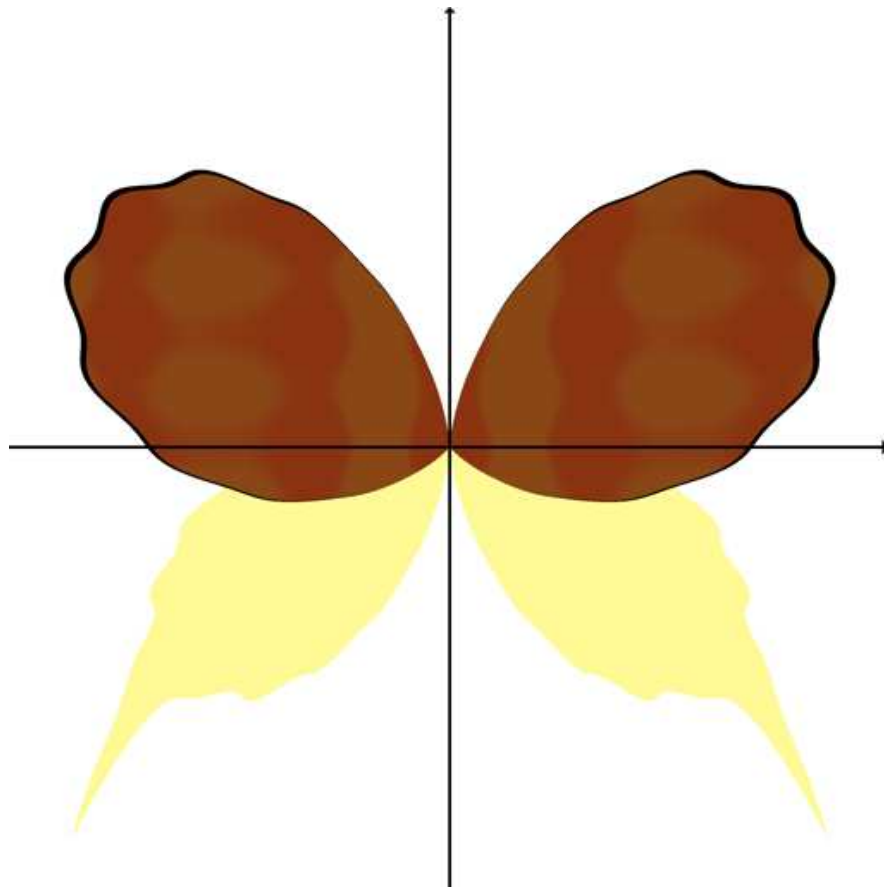
$$\text{color} = \text{lerp}(c1 \cdot \text{step}(th, r), c2, \text{step}(th', r))$$

content.textures



$$p_1 = \sum_{i=1}^N a_i \cdot \cos(k_i \cdot xy)$$
$$p_1 = sstep(-1/2, 1/2, \sum_{i=1}^N a_i \cdot \cos(k_i \cdot xy))$$
$$c2 = lerp(orange, red, p_1)$$

content.textures



$$p_1 = \sum_{i=1}^N a_i \cdot \cos(k_i \cdot xy)$$

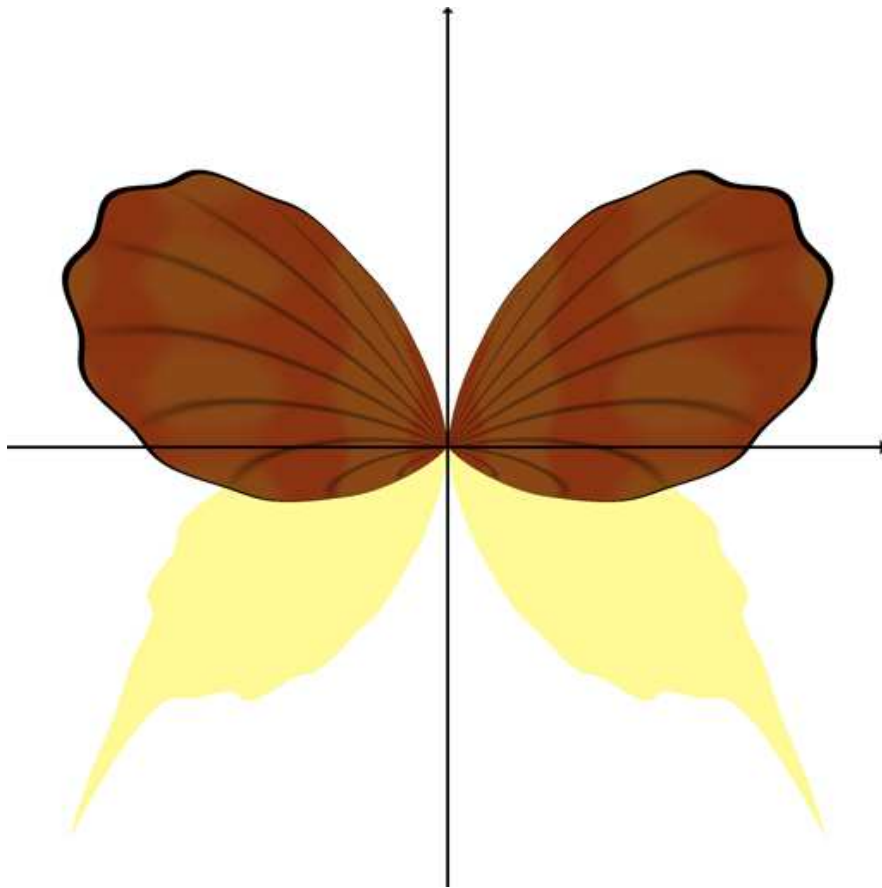
$$p_1 = sstep(-1/2, 1/2, \sum_{i=1}^N a_i \cdot \cos(k_i \cdot xy))$$

$$rn = r / th$$

$$p_2 = \max(0, 1 - 2 \cdot rn^{64})$$

$$c2 = p_2 \cdot lerp(\text{orange}, \text{red}, p_1)$$

content.textures



$$p_1 = \sum_{i=1}^N a_i \cdot \cos(k_i \cdot xy)$$

$$p_1 = sstep(-1/2, 1/2, \sum_{i=1}^N a_i \cdot \cos(k_i \cdot xy))$$

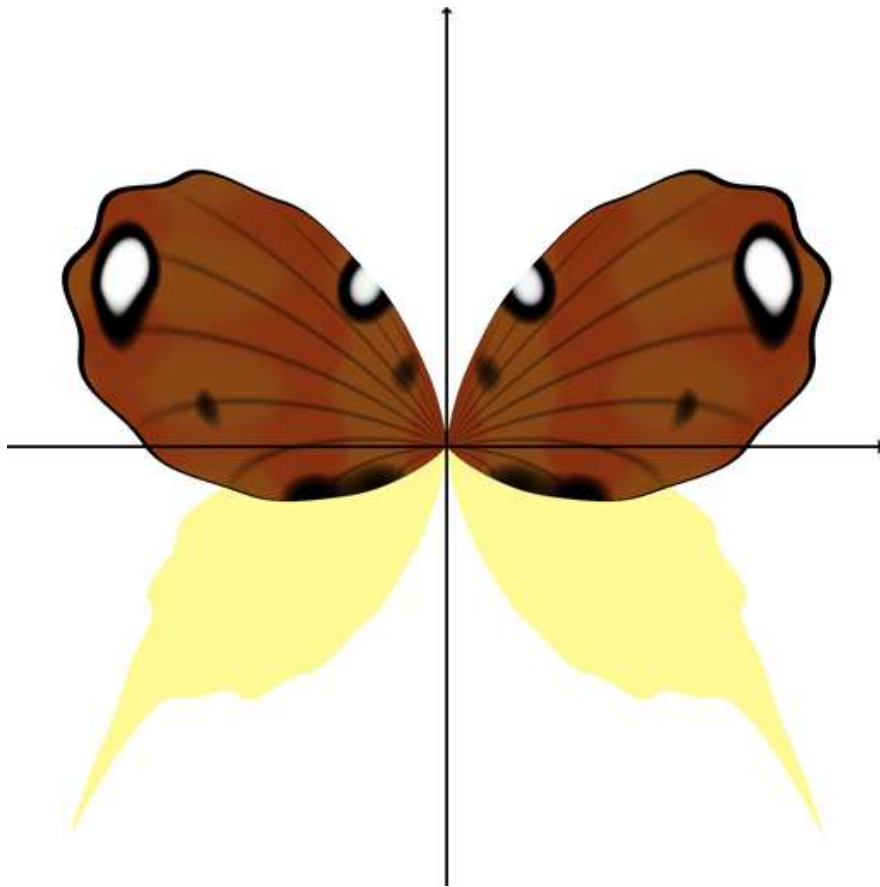
$$rn = r / th'$$

$$p_2 = \max(0, 1 - 2 \cdot rn^{64})$$

$$p_3 = 1 - .4 \left(\frac{1}{2} + \frac{1}{2} \sin(32(\alpha - rn/3)) \right)^{1+40rn}$$

$$c2 = p_3 p_2 \cdot lerp(orange, red, p_1)$$

content.textures



$$p_1 = \sum_{i=1}^N a_i \cdot \cos(k_i \cdot xy)$$

$$p_1 = sstep(-1/2, 1/2, \sum_{i=1}^N a_i \cdot \cos(k_i \cdot xy))$$

$$rn = r/th'$$

$$p_2 = \max(0, 1 - 2 \cdot rn^{64})$$

$$p_3 = 1 - .4 \left(\frac{1}{2} + \frac{1}{2} \sin(32(\alpha - rn/3)) \right)^{1+40rn}$$

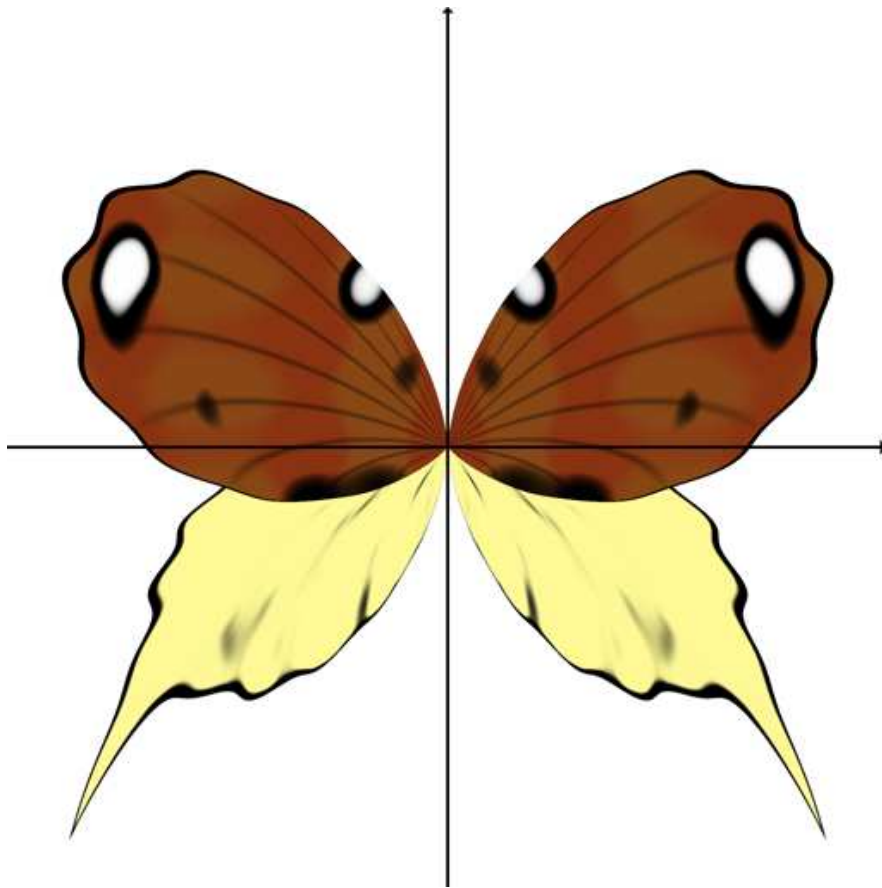
$$tu = \frac{1}{2} noise(6|x|, 6y) + \frac{1}{4} noise(12|x|, 12y)$$

$$p_4 = sstep(0.12, 0.22, tu)$$

$$p_5 = sstep(0.22, 0.30, tu)$$

$$c2 = lerp(white, p_4 p_3 p_2 \cdot lerp(orange, red, p_1), p_5)$$

content.textures



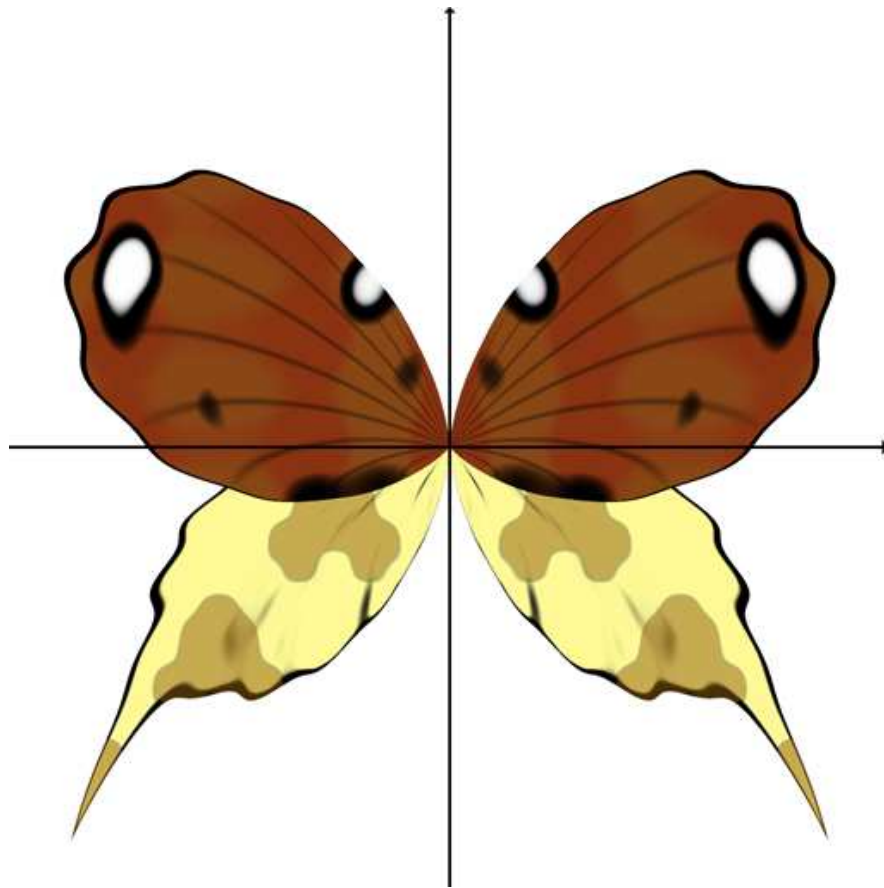
$$uv = \left(5r, \frac{5\beta}{1+r} \right)$$

$$p_1 = \sum_{i=1}^N \frac{1}{2^i} \cdot \text{noise}(2^i \cdot uv)$$

$$p_2 = \max \left(0, 1 - 2 \cdot \left(r / th' \right)^{64} \right)^3$$

$$c1 = \text{yellow} \cdot \text{sstep}(-1/2, 0, p_1) \cdot p_2$$

content.textures



$$uv = \left(5r, \frac{5\beta}{1+r} \right)$$

$$p_1 = \sum_{i=1}^N \frac{1}{2^i} \cdot \text{noise}(2^i \cdot uv)$$

$$p_2 = \max \left(0, 1 - 2 \cdot \left(r / th' \right)^{64} \right)^3$$

$$p_3 = \sum_{i=1}^N a_i \cdot \cos(w_i \cdot uv)$$

$$c1 = \text{lerp}(\text{orange}, \text{yellow} \cdot \text{sstep}(-1/2, 0, p1), p_3) \cdot p_2$$

- Result, a (ok, very simple) texture in just few minutes.

content.mesh animation

- Meshes are animated also with formulas (most compact animation system ever).
- For example, to move the hears of a rhino, for each vertex
 - calc distance d to an predefined (hardcoded) "center" point on the hear.
 - calc a turbulence func $n(t)$
 - if $n(t) > 0.9$ grab the current time $t_o = t$
 - rotate by an $a = \cos(19(t-t_o)) * \exp(-10(t-t_o)) * \text{smoothstep}(0.3, 0.31, 1-d)$
- Pros:
 - 5 minutes to setup.
 - No animator involved (yeah!).
 - No curves to export.
- Cons:
 - To cheap "coder-like" animation...
 - ... or may be not if more than 5 minutes were inverted...

Interesting techniques used

- **1 General intro system**

- The tools
- The pipeline
- The code
- The implementation
- The content
 - Textures
 - Mesh animation

- **2 Interesting techniques used**

- Ambient Occlusion
- Mesh generation
 - Procedural
 - Compression
 - Other tricks

- **3 Techniques for the present/future**

- Mesh compression revisited
- Collashing
- Automatic skinning
- Curves
- Cloth
- Hair

- **4 Tip & tricks**

- Normals for quads
- Normalizing a mesh
- Carmull-Clark subdivision in 50 lines

- **5 Conclusions**

ambient occlusion

- Pros
 - Conceptually simple, yet it improves realism, or artistic choices.
 - Easy to implement (for each vertex: place camera, render, and average).
 - Fast to compute (if VBO/IBO of course).
 - Zero runtime cost.
 - Probably no need to change your vertex format: pick any unused slot, or store it in the normal length (extract it when normalizing it in the shader).



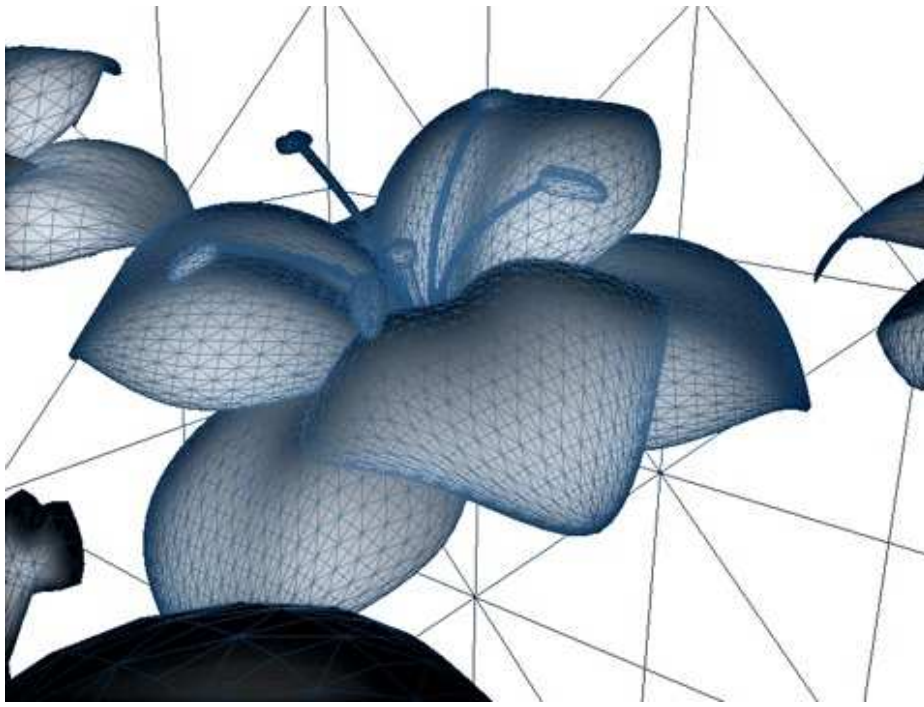
Without ambient occlusion.



Witht ambient occlusion. Note the "soft shadows".

ambient occlusion

- Cons
 - Need for moderate tessellation level (what is not a problem in most intros/demos anyway).
 - Only works for static objects ☹. Hm... Really?



Tessellation should be high enough to capture occlusion variations.

3d meshes

3d meshes.procedural generation

- Only two generators/primitives: revolution and seed mesh. There is no need to have spheres, cylinders or cones as primitives.
- The rest is just operators/modifiers:
 - Subdivide
 - Relax/smooth
 - Symmetrice
 - Extrude
 - Rotate, translate, scale
 - **Displace**
- The most powerful is "displace": why? Because it's all about formulas again!
 - Write a small shader (formulas), and create everything you want;
 - From deserts to machines.
 - For example, the oldest procedural creation object ever: a mountain = one quad + subdivide + displace (formulas - fbm).

3d meshes.procedural generation



Columns are revolution objects. Before displacement.



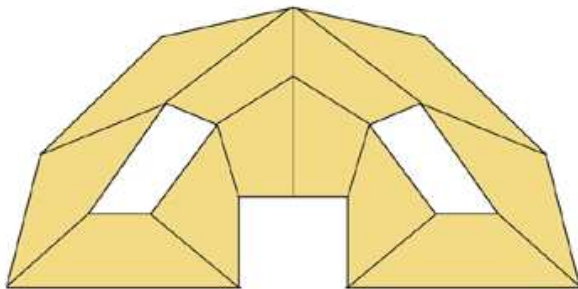
After displacement.



The green body piece is made of a dozen polygons. The ripples are procedurally added by a sinus displacement shader.

3d meshes.procedural generation

- Example: seed mesh + symetric + extrude + subdivide + displace.



Seed mesh

Displacement shader for teeth

```
a = atan2f( ver->x, ver->z );
f = sinf( 16.0f*a );
f = .1f*smoothstep( f, -.1f, .1f );
r = ver->x*ver->x + ver->z*ver->z;
if( r>0.9f )
{
    ver->x += f*nor->x;
    ver->z += f*nor->z;
}
```

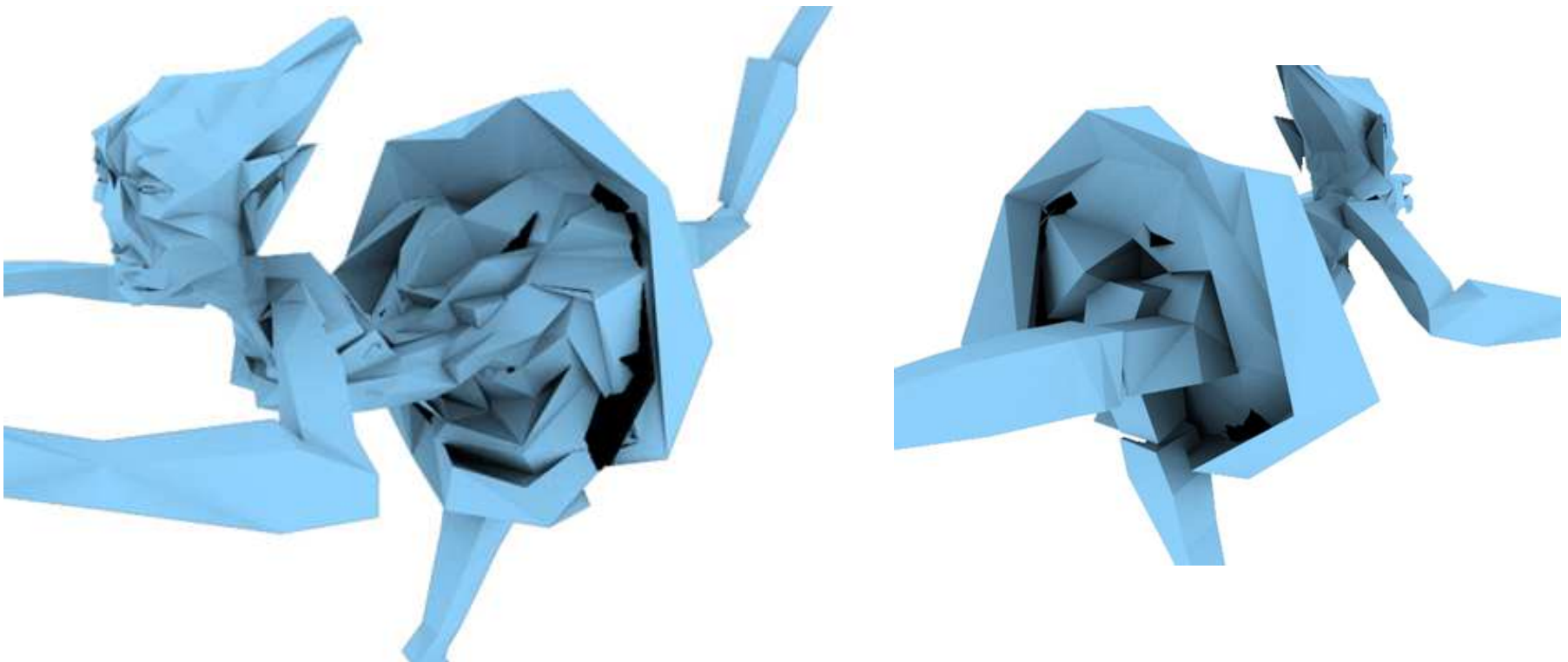


3d meshes.compression

- When proceduralism can not go further, we need something else.
- Once a “seed” mesh is loaded, procedural generation can continue on top of it.
- In *Paradise* each seed mesh was around 200 polys. In *195/95/256* each seed is around 900 polys.
- The BIG problem with storing meshes is the vertex arrays. Index arrays are very compressible.
- In CG research papers you will find 1 bit/face compression rates. Forget it. It only works for dense (high correlated) meshes. Seed meshes are anything but low frequency.

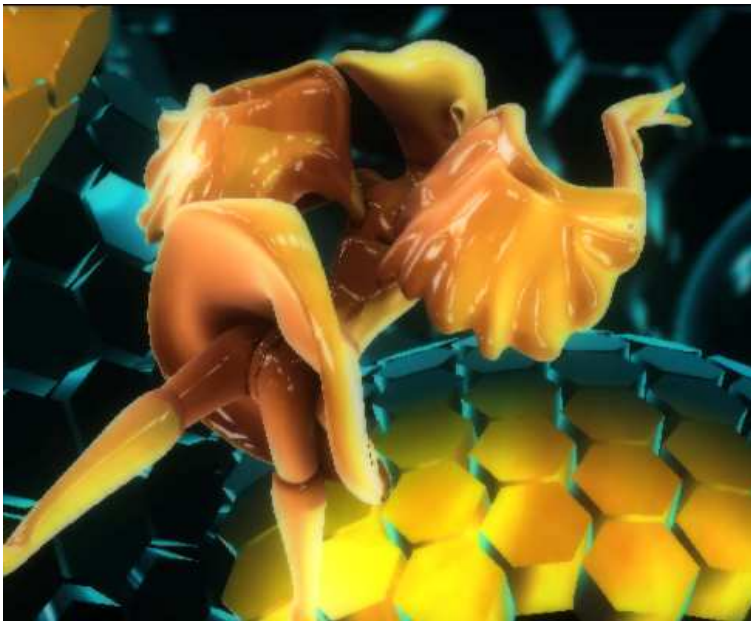
3d meshes.compression

- Seed meshes are much more like a white noise than a smooth thing.
- Polygons traversing a considerable proportion of the object's bounding box are frequent; and this is bad for vertex prediction (read delta coding).
- Clashing triangles, non manifold geometry are also normal.



3d meshes.other tricks

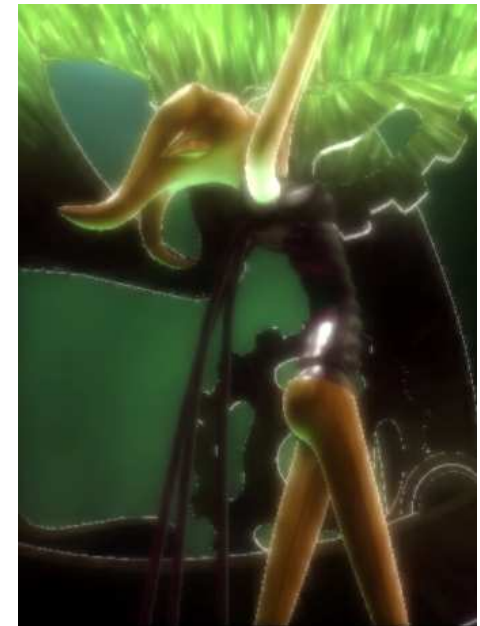
- Apply common sense, simple trick can bring the biggest space savings:



The wings are symmetric.



*The body is symmetric.
The legs are instanced.*



*Both legs are instanced.
The head comes from a symmetric seed mesh, and a displacement shader is used to bend the horns in opposite directions.*

Techniques for the present/future

- **1 General intro system**

- The tools
- The pipeline
- The code
- The implementation
- The content
 - Textures
 - Mesh animation

- **2 Interesting techniques used**

- Ambient Occlusion
- Mesh generation
 - Procedural
 - Compression
 - Other tricks

- **3 Techniques for the present/future**

- Mesh compression revisited
- Collashing
- Automatic skinning
- Curves
- Cloth
- Hair

- **4 Tip & tricks**

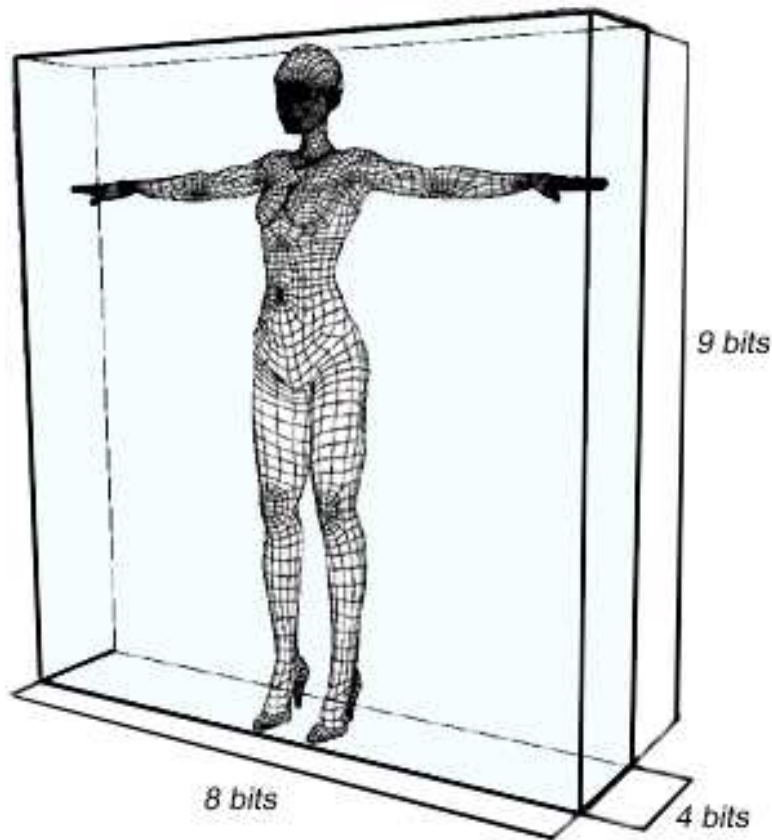
- Normals for quads
- Normalizing a mesh
- Carmull-Clark subdivision in 50 lines

- **5 Conclussions**

mesh compression revisited

quantification

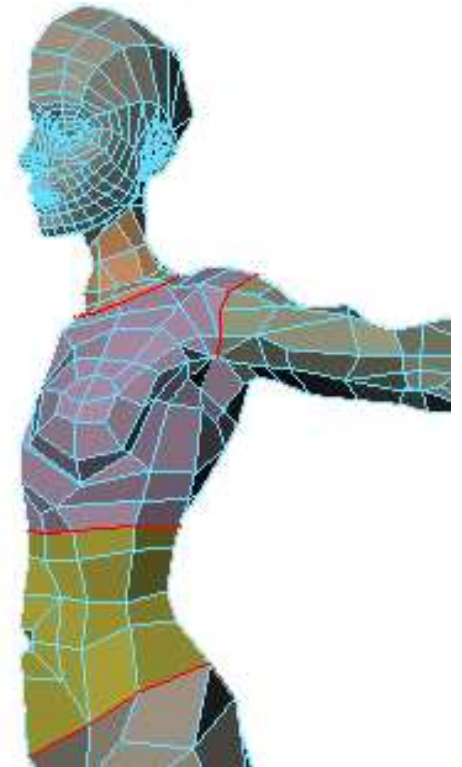
- First step is always quantification.
- Do not waste bits, sample in uniform cubes.



Make the number of bits for each dimension proportional to the \log_2 of the corresponding bounding box side length.

- Or even better, adapt your sampling rate to the density of vertices: split your mesh and quantify each part separately.

Each mesh part (defined by a different color) is coded separately. Thus, the head uses more bits than the rest of the body that has less details and is less important.

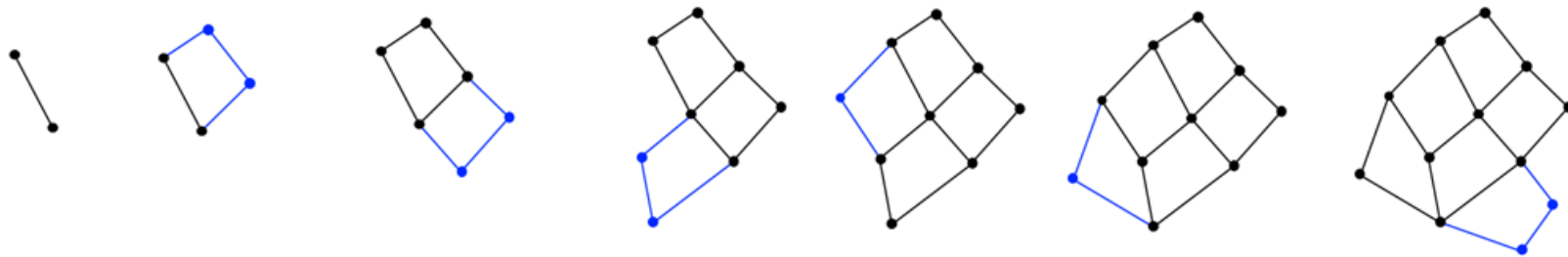


simple codification

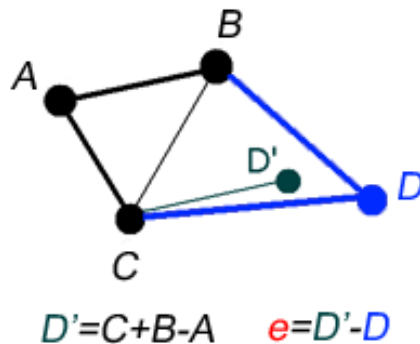
- Vertex arrays:
 - Quantify.
 - Store with delta coding as vertices come from the exporter.
 - Deinterlace the dx , dy , dz , du , dv ... components for better compression.
- Index arrays:
 - Store as delta indices, and deinterlace in high and low bytes (high bytes will be zero most of the time, even if $\#polygons > 256$). Or nibbles!
 - To allow triangles/quads without extra bit, store first triangles, then quads.
- Works more or less fine (used in *195/95/256*).
- Not optimal, we relay on the exporter for mesh traversal.
 - Deltas could probably be smaller.
 - Index arrays have redundant information; each edge is defined twice!

improved codification

- Let's traverse the mesh recursively, so we can more coherently encode geometry and topology.



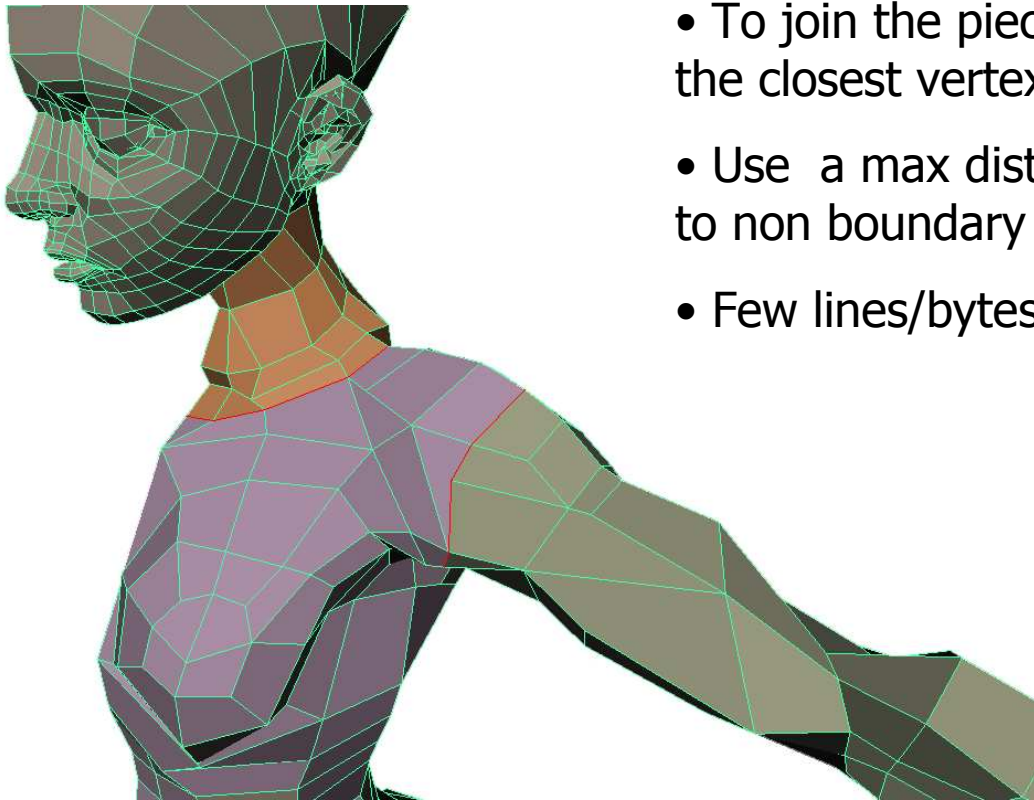
- On the process, try predicting where vertices will be; don't just delta encode.



*Simple delta encoding
would store $D - C$, that
normally is quite larger
than $D - D'$.*

collashing

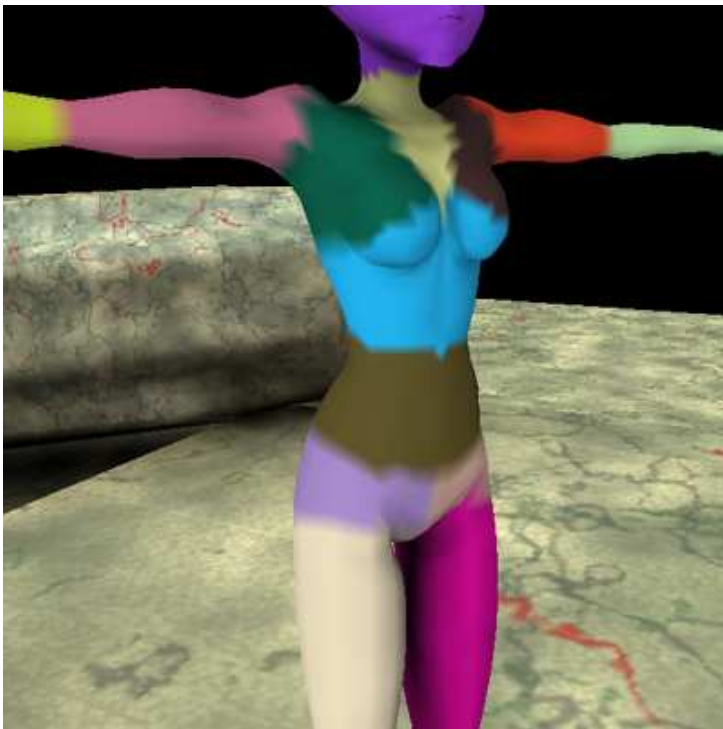
- Divide your (seed) mesh in pieces.
- Allows collashing: library of noses, hears, mouths,... (and all blended variations!).
- Side effect: better quantification quality control, and fast collision detection.
- Very powerfull in combination with "delta meshes".



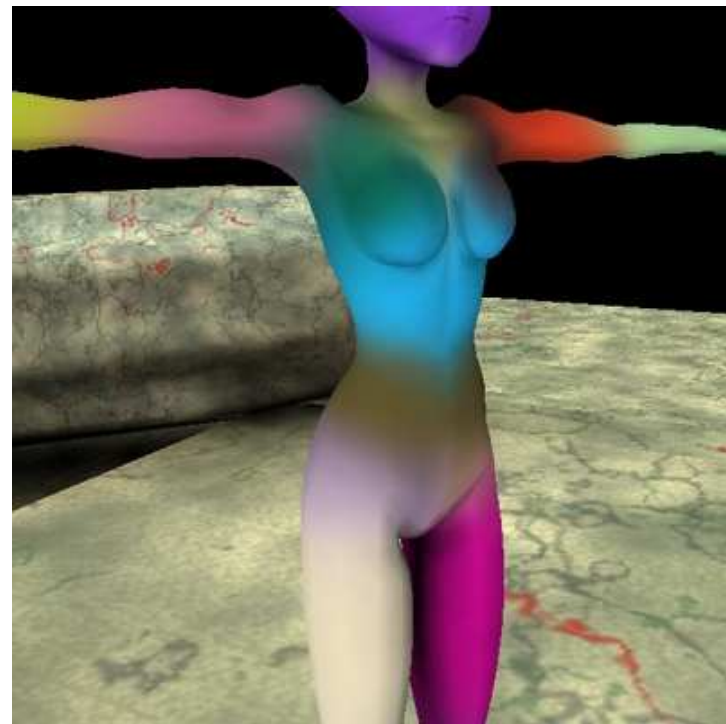
- To join the pieces, for vertex in piece A search the closest vertex in piece B.
- Use a max distance threshold to avoid connecting to non boundary vertices.
- Few lines/bytes of code.

automatic skinning

- Input=just a skeleton; Output=vertex weights.
- For each vertex, find the closest n bones (point to segment distance).
- Assign weights according to the distances.
- Blur the weights (in mesh space), and normalize.



The closest bone identified by each color.



The closest bone, after blurring.

automatic skinning



Deformed mesh without blurring.



With blurring.

animation curves

- For skeletons, only a quaternion per bone is needed.
- Sample it at 30 Hz.
- We can represent each quaternion with 3x10 bits [0..1023], for example
$$(a_x, a_y, a_z, w) \rightarrow (\alpha, \beta, w)$$
- So, 30 bones, one minute of animation (@ 30 Hz) is 198 kb!
 - BUT, if max speed is say 720 degrees/seconds, then max abs delta is 68!
 - or if max speed 540 deg/sec, and 9 bits, then max abs delta is 25.
 - And the average will be _much_ smaller.
 - Specially for axis information.
 - So, as usual, delta encode the signal...
 - ...or better, use linear prediction and encode the error.
 - Need to find the best coefficients, quite like in speech synthesis.
 - And other usual techniques (deinterlacing high and low order bytes/nibbles)

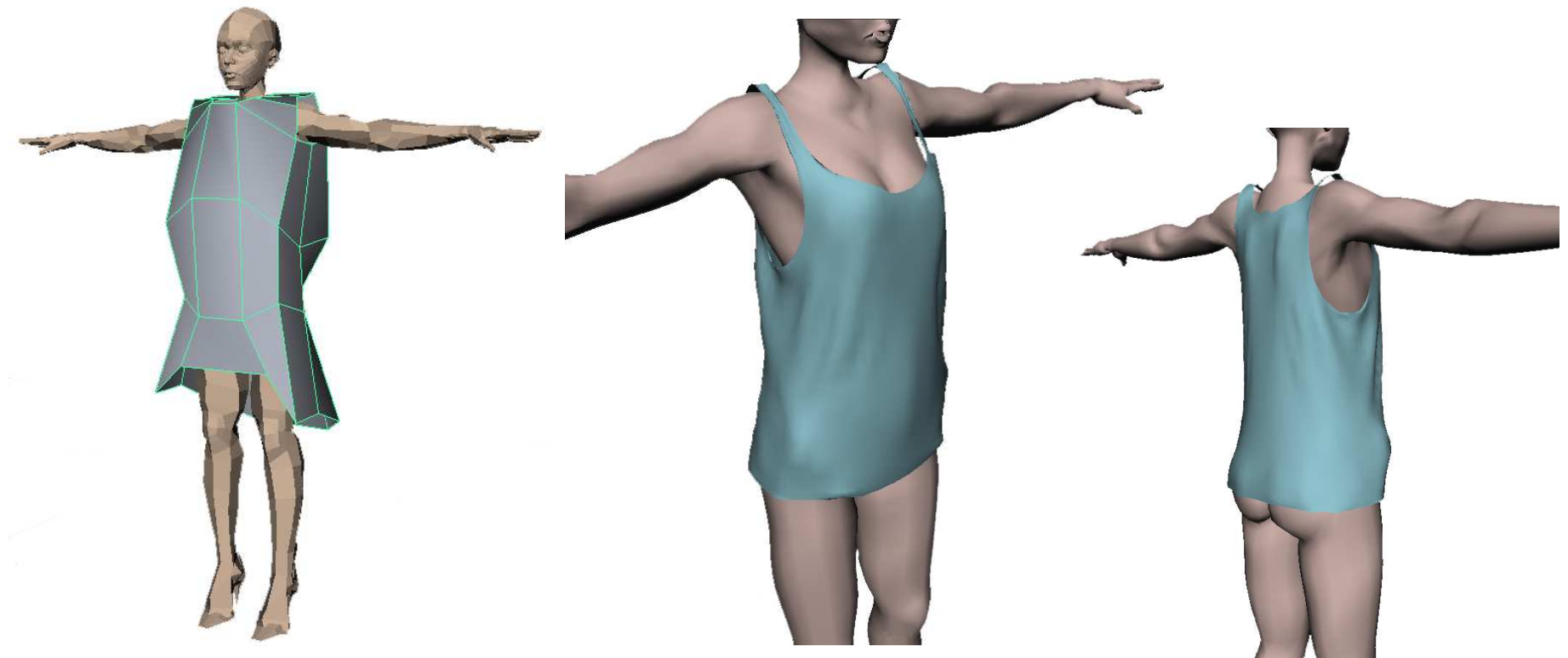
cloth

- We have basically not seen cloth yet in CG movies.
 - And when it is used in non pure CG movies, it's always in short shots.
 - And the cloth is dark, to hide self intersections and other artifacts.
- All this means it's not an easy task.
 - Will the democoders take the challenge?



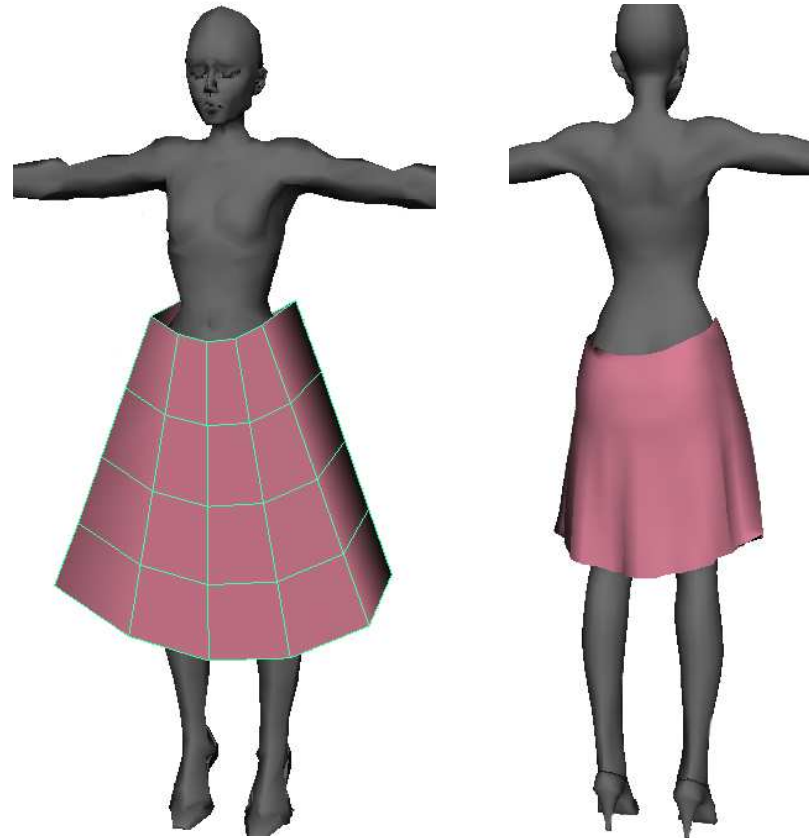
cloth, a first approach for 64 kb

- Classic (and simple!) mass-spring model.
- Can be coded in one hour. Fine tuned in two.
- Simple cloth seed mesh.
- Subdivide, and simulate: gravity and friction, that's all...



cloth, a first approach for 64 kb

- Well, of course we need collisions with the body (ray-mesh).
 - Use the body seed mesh as collision proxy (we have it for free!)
 - Collashing of meshes allows here to discard complete pieces of the mesh that can not intereseect.
 - Unsubdivided body proxy speeds up collision, and to avoids the small penetrations of the cloth mesh.
- Forget about self-collision of the cloth.
 - Really.
 - Just try to hide them the best you can: use dark cloths.



cloth, a first approach for 64 kb

- And what about realtime animation of the cloth on the characters?
- We need realtime collision detection then...
- For a demo, we could afford colliding the complete mesh (with a kd-tree).
- In intros, just collide a set of approximating spheres (very fast!).
 - For each skeleton join, store a radius (orange balls).
 - Interpolate radius along bones and generate spheres on the fly (blue).



One additional byte per bone allows for realtime collision detection.

hair

- Again, hair is almost inexistant in CG movies, even if it's easier to create.
- Only "Violete" in "The Incredibles" has complete hair simulation.
 - Other characters on the same movie have a "block" of hair.
 - "Princess Fiona" in "Shrek" wears a ponytail all the time...
 - And the rest are just metallic, bold, or fury at most.



Tips & tricks

- **1 General intro system**

- The tools
- The pipeline
- The code
- The implementation
- The content
 - Textures
 - Mesh animation

- **2 Interesting techniques used**

- Ambient Occlusion
- Mesh generation
 - Procedural
 - Compression
 - Other tricks

- **3 Techniques for the present/future**

- Mesh compression revisited
- Collashing
- Automatic skinning
- Curves
- Cloth
- Hair

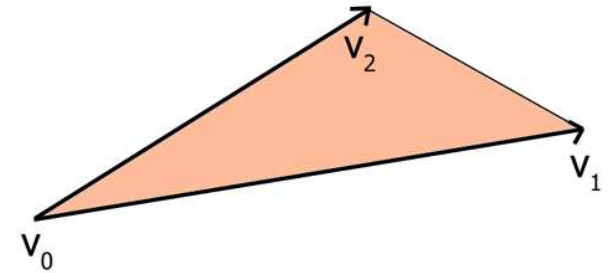
- **4 Tip & tricks**

- Normals for quads
- Normalizing a mesh
- Carmull-Clark subdivision in 50 lines

- **5 Conclusions**

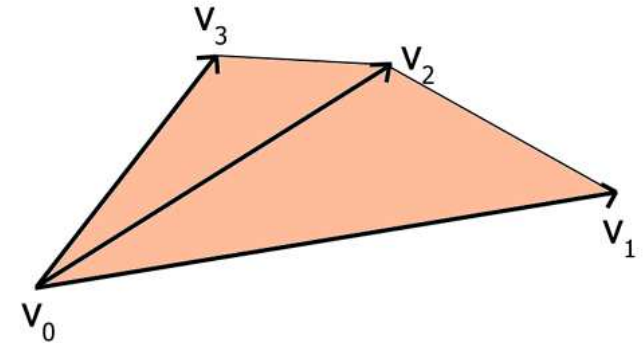
calculating normals for quads, the easy way

$$n = (v_1 - v_0) \wedge (v_2 - v_0) = v_0 \wedge v_1 + v_1 \wedge v_2 + v_2 \wedge v_0$$



- So, what if we extrapolate this to more than three vertices?

$$n = v_0 \wedge v_1 + v_1 \wedge v_2 + v_2 \wedge v_3 + v_3 \wedge v_0 = n_{012} + n_{023}$$



- We get the average normal of the two triangles.
- This is good enough for (4k) intros.
- And we don't need to calculate edge vectors.

```
for( i=0; i<4; i++ )  
    nor += v[i] ^ v[(i+1)&3];
```


normalizing a mesh

```
// index array has all quads
// normals are assumed to be memset'ed to zero
void calcNormals( Vertex *va, const int *ia, int nv, int nf )
{
    for( int i=0; i<nf; i++, ia+=4 )
        for( int j=0; j<4; j++ )
        {
            tmp = va[ ia[(j+1)&3] ].pos ^ va[ ia[j] ].pos;
            for( int k=0; k<4; k++ )
                va[ ia[k] ].nor += tmp;
        }
}
```

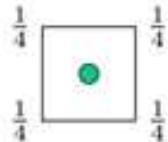
- No need to normalize face normals, we allow bigger faces to influence more on the vertex normal.
- Actually, no need to calculate face normals at all.
- No need to divide the vertex normal by the number of adjacent faces.
- No need to normalize, since in the pixel shader we are normalizing anyway! If there is no pixel shader (!?), then add just two lines...

```
for( int i=0; i<nv; i++ )
    va[i].nor.normalize();
```

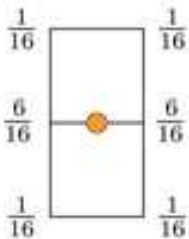
- ...or if you are 4k intro coder, use GL_NORMALIZE.

catmull Clark subdivision, in 50 lines

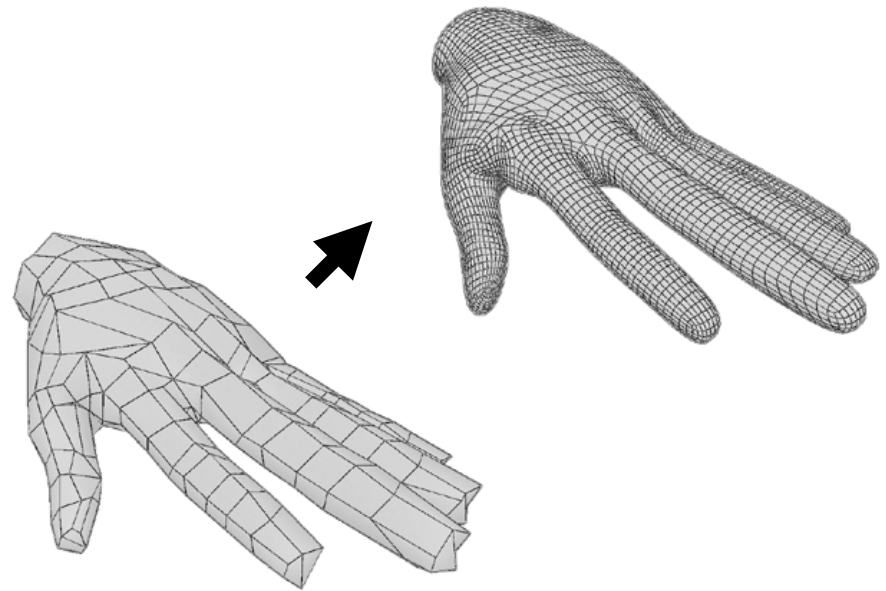
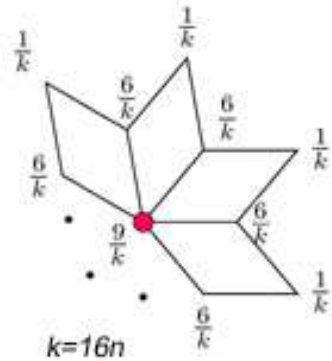
Face Points



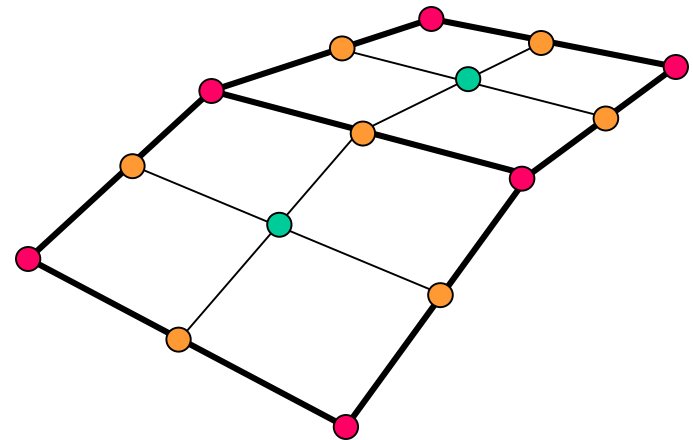
Edge Points



Vertex Points



```
static int getEdgeID( int a, int b, int *edges, int *nedges )
{
    int n = (a>b)?(a<<16)|b:(b<<16)|a;
    int i; for( i=0; edges[2*i+0]!=n && i<nedges; i++ );
    edges[2*i+0] = n;
    edges[2*i+1]++;
    (*nedges)+= (i==nedges);
    return i;
}
```



catmull Clark subdivision, in 50 lines

```

const static float vpw[4] = { 9.0f, 3.0f, 1.0f, 3.0f };
const static float epw[4] = { 3.0f, 3.0f, 1.0f, 1.0f };
void catmulclark( QMESH *dst, const QMESH *src )
{
    int i, j, k;
    int eid[4], abcd[4];
    int edges[MAXEDGES], face_valences[MAXVERTS];
    int nedges = 0;
    int off_vp = src->nq;
    int off_ep = src->nq + src->nv;

    memset( edges, 0, MAXEDGES*4 );           // reset edges
    memset( face_valences, 0, MAXVERTS*4 );   // reset valences
    for( j=0; j<src->nq; j++ )
    {
        for( k=0; k<4; k++ )
        {
            for( i=0; i<4; i++ ) abcd[i] = src->quads[j].ve[(i+k)&3]; // get the 4 vertices
            eid[k] = getEdgeID( abcd[0], abcd[1], edges, &nedges ); // create edges
            face_valences[abcd[0]]++; // update face-valence

            dst->va[j] += 0.25f * src->va[abcd[0]]; // increment face point
            for( i=0; i<4; i++ )
            {
                dst->va[off_vp + abcd[i]] += vpw[i] * src->va[abcd[i]]; // incremente vertex point
                dst->va[off_ep + eid[k]] += epw[i] * src->va[abcd[i]]; // increment edge point
            }
        }

        for( k=0; k<4; k++ ) // make child faces
        {
            dst->quads[4*j+k].ve[0] = j;
            dst->quads[4*j+k].ve[1] = off_ep + eid[(3+k)&3];
            dst->quads[4*j+k].ve[3] = off_ep + eid[(0+k)&3];
            dst->quads[4*j+k].ve[2] = off_vp + src->quads[j].v[k];
        }
    }

    for( j=0; j<src->nv; j++ ) dst->va[off_vp + j] *= 0.0625f/(float)(face_valences[j]);
    for( j=0; j<nedges; j++ ) dst->va[off_ep + j] *= 0.1250f/(float)(edges[j].valence);

    dst->nv = off_ep + nedges;
    dst->nq = src->nq * 4;
}

```

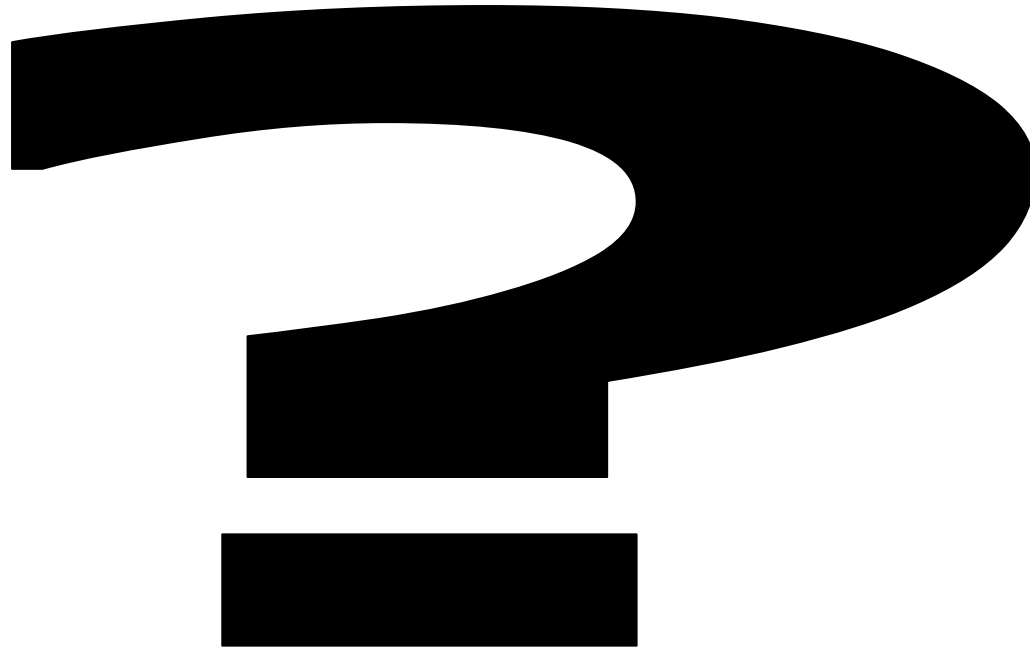
The future

- In recent future we will probably see (I would like to see)
 - Rigid collision detections and physics
 - Global illumination (AO, SH, Radiosity Maps, ...)
 - Cloth
 - Hair
 - Skinned meshes

Conclusions

- There is few thing you could do in a demo and not in an intro (other than speech or mzk).
- And no real restriction to continue improving techniques and quality of intros during 5 more years.
- But now that intros are at the same visual level than demos (and at higher technical level), they can **suffer** the same effect as demos from coder's point of view: the need of an artist to create the content (on top of *art*) to feed the code.
- And this means lot of work for artists. At least, quite more than before.
- And, this only my opinion, in demoscene both art and code quality are beyond content quality in most cases.

Questions



(simples, please...)