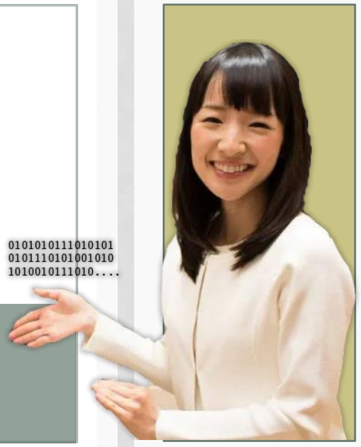
FROM BITSTREAMS TO BUNNIES:

HOW TO GO MARIE KONDO ON THE VRAM

BY BAKTASH ABDOLLAH-SHAMSHIR-SAZ

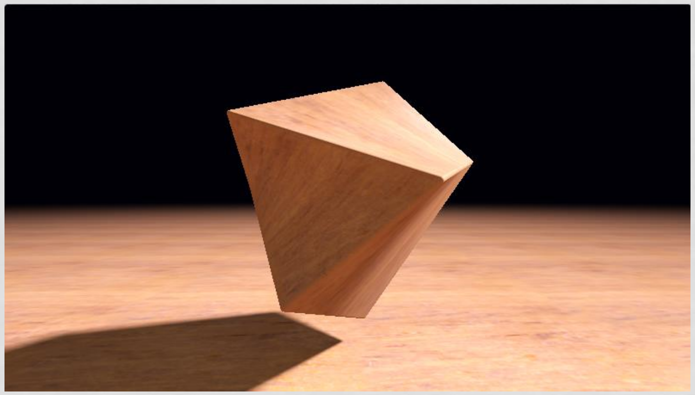


THE MOST IMPORTANT RULE

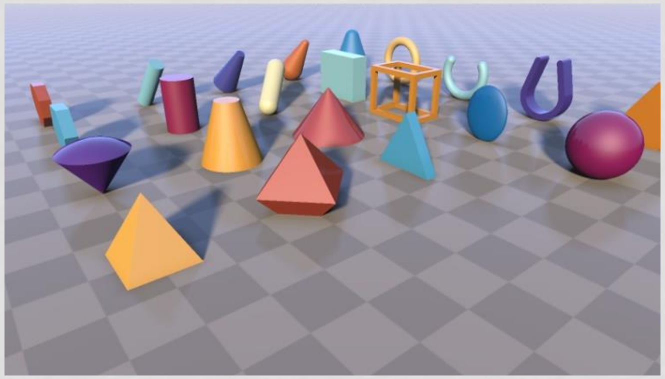
- Marie Kondo is well known for this most simple rule: does it spark joy?
  - This actually applies to memory optimization on the GPU.
- How to read this if you're designing data streams:
  - Do you need this (bit) to achieve the desired effect (joy)?
  - Or could this be inferred from the surrounding environment?
  - Memory reads are expensive.
  - So let's only keep things that spark joy. We can infer the rest.

CHALLENGE #1

- ShaderToy is a great platform for visualizing analytical expressions (implicit surfaces, analytical bilinear patches etc.)



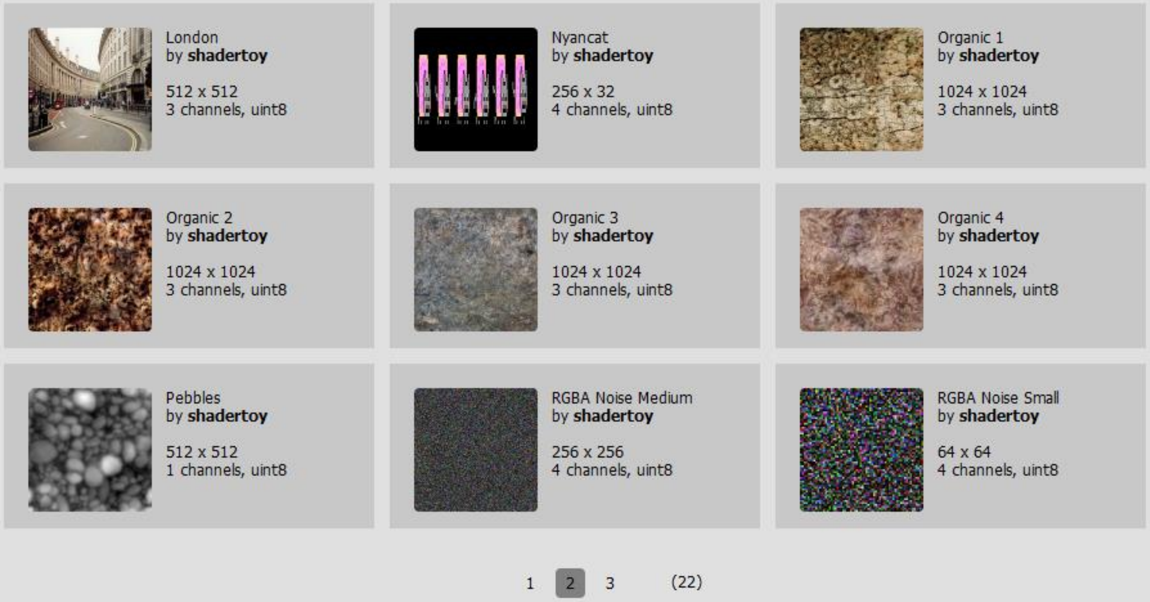
https://www.shadertoy.com/view/3tjczm
by Inigo Quilez



https://www.shadertoy.com/view/Xds3zN
by Inigo Quilez

CHALLENGE #1

- However, you cannot bring in external images 😔
- You're limited to a pre-selected group of images



London
by **shadertoy**
512 x 512
3 channels, uint8

Nyancat
by **shadertoy**
256 x 32
4 channels, uint8

Organic 1
by **shadertoy**
1024 x 1024
3 channels, uint8

Organic 2
by **shadertoy**
1024 x 1024
3 channels, uint8

Organic 3
by **shadertoy**
1024 x 1024
3 channels, uint8

Organic 4
by **shadertoy**
1024 x 1024
3 channels, uint8

Pebbles
by **shadertoy**
512 x 512
1 channels, uint8

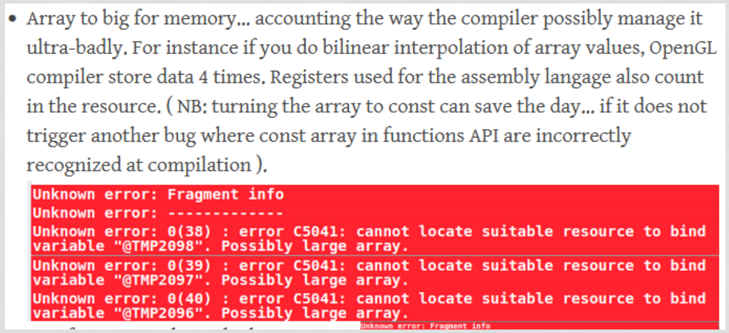
RGBA Noise Medium
by **shadertoy**
256 x 256
4 channels, uint8

RGBA Noise Small
by **shadertoy**
64 x 64
4 channels, uint8

1 2 3 (22)

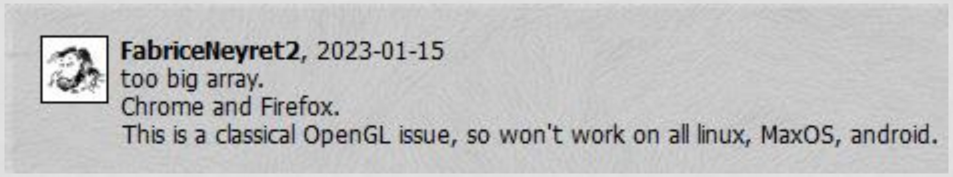
CHALLENGE #1

- If you try to bring in contraband data via large arrays:
  - Generally, you should be able to go as large as 4096 array elements.
  - However, some compilers – specifically nVidia OpenGL backends for ANGLE on Linux/Android/macOS – will explode as they erroneously allot 4x the amount of memory/registers necessary for accessing said data.
  - Thus, reducing your cross-platform capacity to 1024 array elements.



- Array to big for memory... accounting the way the compiler possibly manage it ultra-badly. For instance if you do bilinear interpolation of array values, OpenGL compiler store data 4 times. Registers used for the assembly langage also count in the resource. (NB: turning the array to const can save the day... if it does not trigger another bug where const array in functions API are incorrectly recognized at compilation).

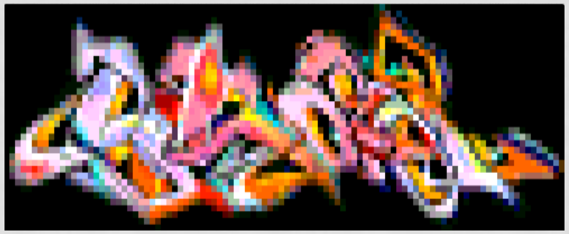
```
Unknown error: Fragment info
Unknown error: ------------
Unknown error: 0(38) : error C5041: cannot locate suitable resource to bind
variable "@TMP2098". Possibly large array.
Unknown error: 0(39) : error C5041: cannot locate suitable resource to bind
variable "@TMP2097". Possibly large array.
Unknown error: 0(40) : error C5041: cannot locate suitable resource to bind
variable "@TMP2096". Possibly large array.
Unknown error: Fragment info
```



FabriceNeyret2, 2023-01-15
too big array.
Chrome and Firefox.
This is a classical OpenGL issue, so won't work on all linux, MaxOS, android.

EXAMPLE #1

- Say we want to encode the following RGBA8 image:
- Do we need all 32 bits to represents this?
  - Could we just keep RGB8 and color key the rest with black?
  - Could we go further and crunch down RGB8 to R2G4B2 and spend 1 byte per pixel along with the color key?
- Yes and yes!
  - And the color representation won't be nearly as bad as it will be uniformly applied



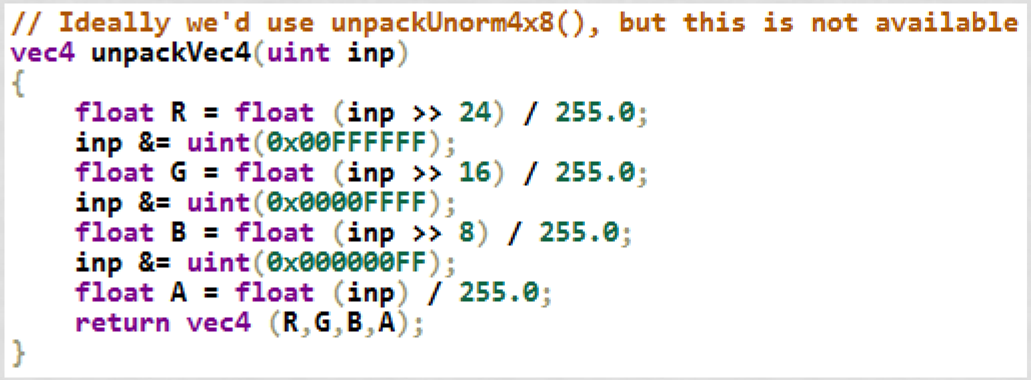
EXAMPLE #1

(We get the image we want in-shader working for all platforms!)

https://www.shadertoy.com/view/tltGWf

(By yours truly)

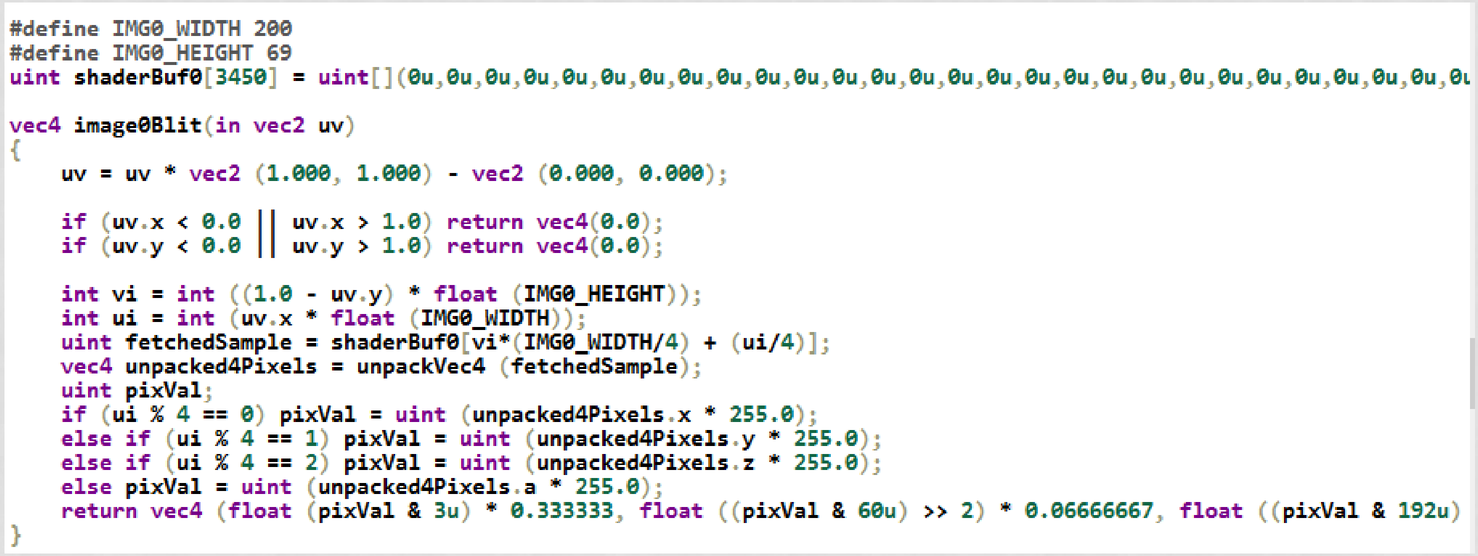
Necessary util (will not exist everywhere):



```
// Ideally we'd use unpackUnorm4x8(), but this is not available
vec4 unpackVec4(uint inp)
{
    float R = float (inp >> 24) / 255.0;
    inp &= uint(0x00FFFFFF);
    float G = float (inp >> 16) / 255.0;
    inp &= uint(0x0000FFFF);
    float B = float (inp >> 8) / 255.0;
    inp &= uint(0x000000FF);
    float A = float (inp) / 255.0;
    return vec4 (R,G,B,A);
}
```

EXAMPLE #1

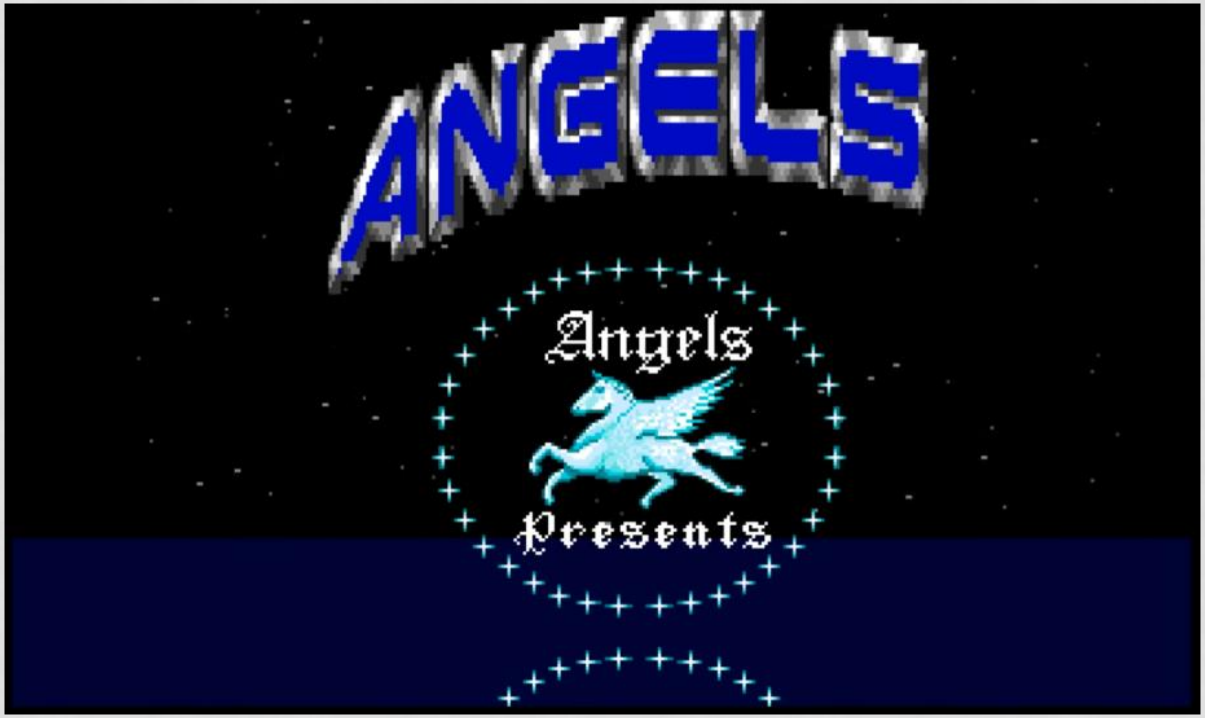
- How to use?
  - Read entire unsigned int containing the byte we're interested in
  - Expand the byte into 3 component color



```
#define IMG0_WIDTH 200
#define IMG0_HEIGHT 69
uint shaderBuf0[3450] = uint[](0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0

vec4 image0Blit(in vec2 uv)
{
    uv = uv * vec2 (1.000, 1.000) - vec2 (0.000, 0.000);

    if (uv.x < 0.0 || uv.x > 1.0) return vec4(0.0);
    if (uv.y < 0.0 || uv.y > 1.0) return vec4(0.0);

    int vi = int ((1.0 - uv.y) * float (IMG0_HEIGHT));
    int ui = int (uv.x * float (IMG0_WIDTH));
    uint fetchedSample = shaderBuf0[vi*(IMG0_WIDTH/4) + (ui/4)];
    vec4 unpacked4Pixels = unpackVec4 (fetchedSample);
    uint pixVal;
    if (ui % 4 == 0) pixVal = uint (unpacked4Pixels.x * 255.0);
    else if (ui % 4 == 1) pixVal = uint (unpacked4Pixels.y * 255.0);
    else if (ui % 4 == 2) pixVal = uint (unpacked4Pixels.z * 255.0);
    else pixVal = uint (unpacked4Pixels.a * 255.0);
    return vec4 (float (pixVal & 3u) * 0.333333, float ((pixVal & 60u) >> 2) * 0.06666667, float ((pixVal & 192u)
}
```

EXAMPLE #2

- Replicating old school Angels cracktro (Shadow of the Beast II on the Amiga)

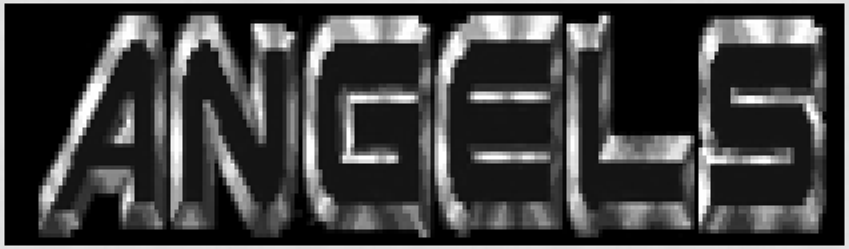


EXAMPLE #2

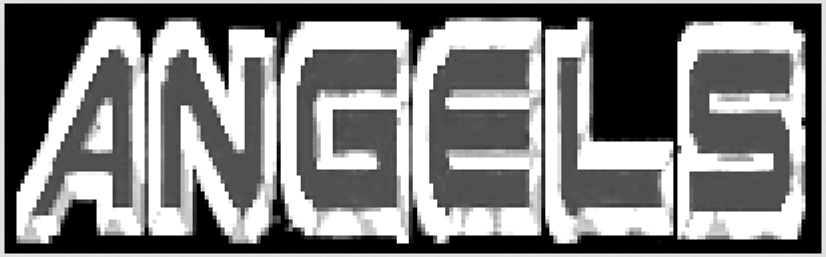
- If we quantize the colors to R2G4B2 we lose the fidelity on the grayscale metallic texture
- Can we keep that and make an exception about the blue?
- Thus staying in 1Bpp *and* maintain fidelity?

EXAMPLE #2

- Encode 1Bpp grayscale but keep the blue part a constant low luminance

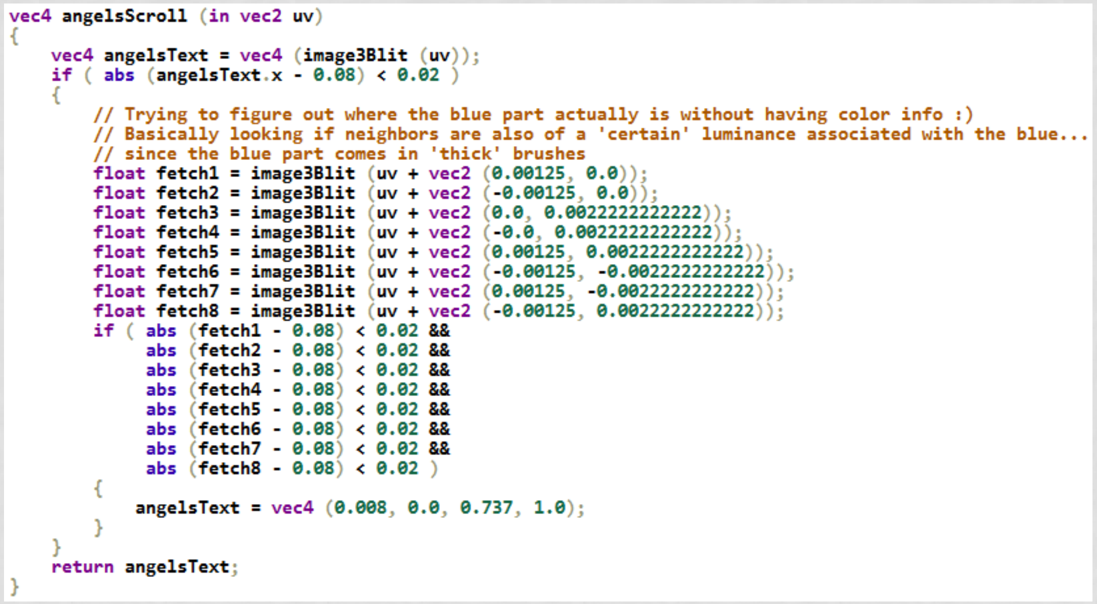


- Luminance scaled x4



EXAMPLE #2

- If we check center luminance…
- … and it’s the (low) magic number…
- … and all neighbors also have the magic number…
- … we can infer that it’s blue!



```
vec4 angelsScroll (in vec2 uv)
{
    vec4 angelsText = vec4 (image3Blit (uv));
    if ( abs (angelsText.x - 0.08) < 0.02 )
    {
        // Trying to figure out where the blue part actually is without having color info :)
        // Basically looking if neighbors are also of a 'certain' luminance associated with the blue...
        // since the blue part comes in 'thick' brushes
        float fetch1 = image3Blit (uv + vec2 (0.00125, 0.0));
        float fetch2 = image3Blit (uv + vec2 (-0.00125, 0.0));
        float fetch3 = image3Blit (uv + vec2 (0.0, 0.0022222222222));
        float fetch4 = image3Blit (uv + vec2 (-0.0, 0.0022222222222));
        float fetch5 = image3Blit (uv + vec2 (0.00125, 0.0022222222222));
        float fetch6 = image3Blit (uv + vec2 (-0.00125, -0.0022222222222));
        float fetch7 = image3Blit (uv + vec2 (0.00125, -0.0022222222222));
        float fetch8 = image3Blit (uv + vec2 (-0.00125, 0.0022222222222));
        if ( abs (fetch1 - 0.08) < 0.02 &&
             abs (fetch2 - 0.08) < 0.02 &&
             abs (fetch3 - 0.08) < 0.02 &&
             abs (fetch4 - 0.08) < 0.02 &&
             abs (fetch5 - 0.08) < 0.02 &&
             abs (fetch6 - 0.08) < 0.02 &&
             abs (fetch7 - 0.08) < 0.02 &&
             abs (fetch8 - 0.08) < 0.02 )
        {
            angelsText = vec4 (0.008, 0.0, 0.737, 1.0);
        }
    }
    return angelsText;
}
```

EXAMPLE #2

ANGELS
Angels
Presents

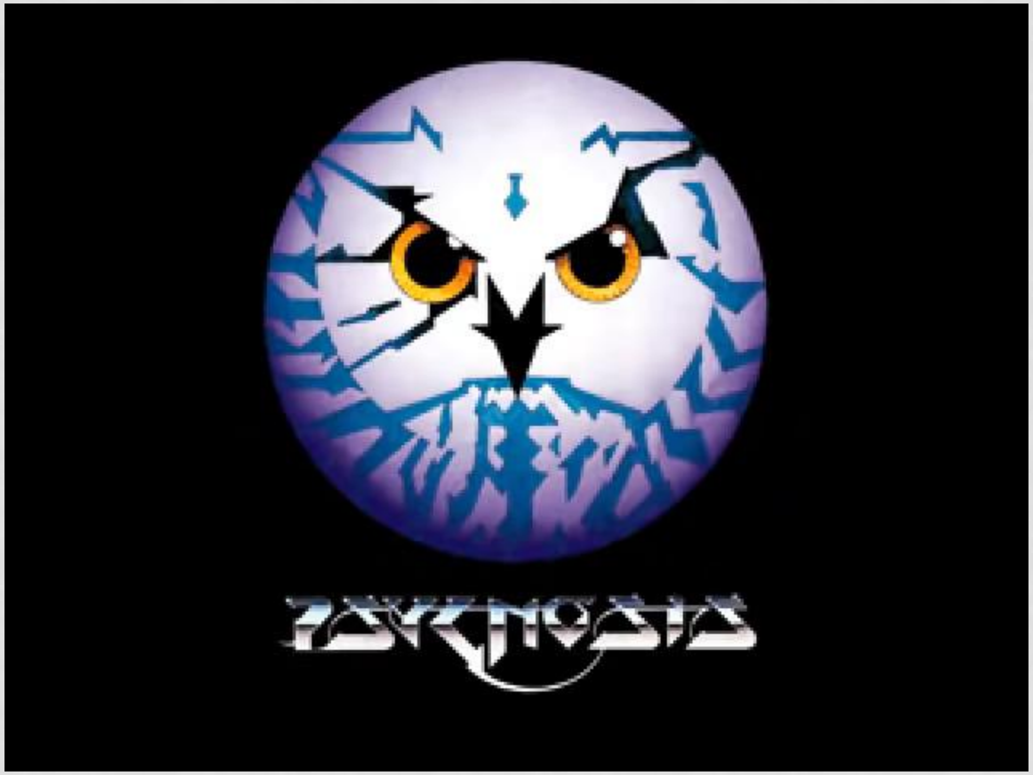
https://www.shadertoy.com/view/WljSR1

(By yours truly)

We have sparked joy!
(by inferring from circumstantial information)

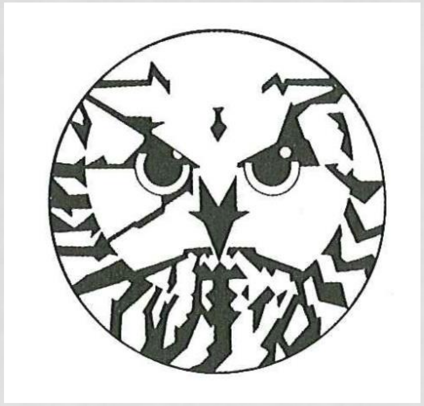
EXAMPLE #3

- Can we replicate the Psygnosis owl? (R.I.P Ian Hetherington)



EXAMPLE #3

- Our options:
  - In-shader SVG renderer:
    - Will be slow
    - Will use a lot of floats and registers
  - Leverage what we have:
    - 1 Byte per pixel
    - Is this really necessary?
    - Can we do better? The owl can be just a black and white stencil:

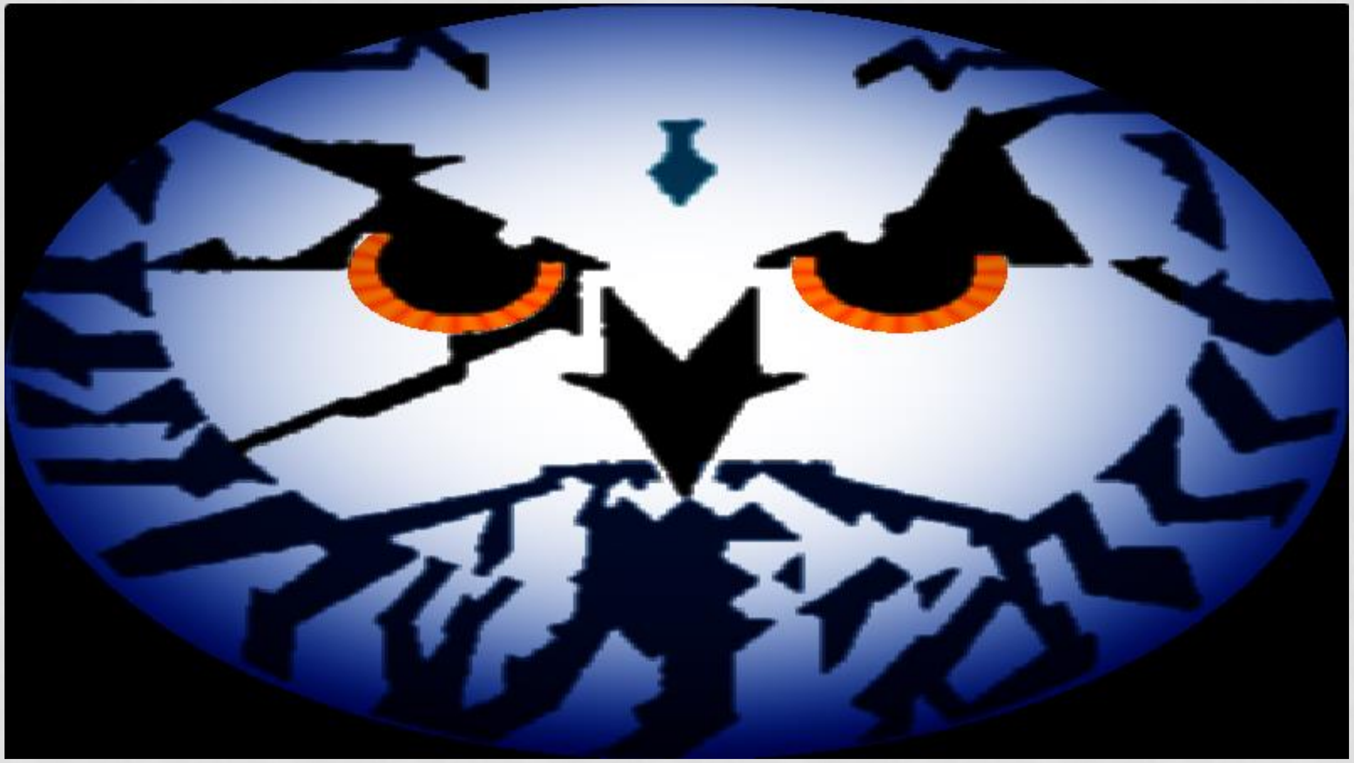


EXAMPLE #3

- We can do literally 1 bit per pixel:
  - And also maintain a rather high resolution

EXAMPLE #3

- Apply 3x3 AA when sampling
  - Add colors and patterns strategically
  - And voila!

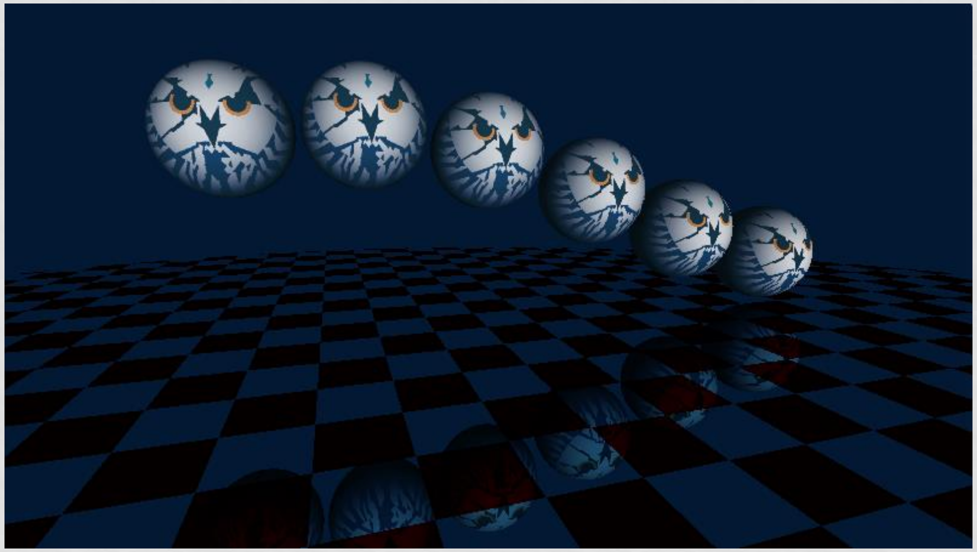


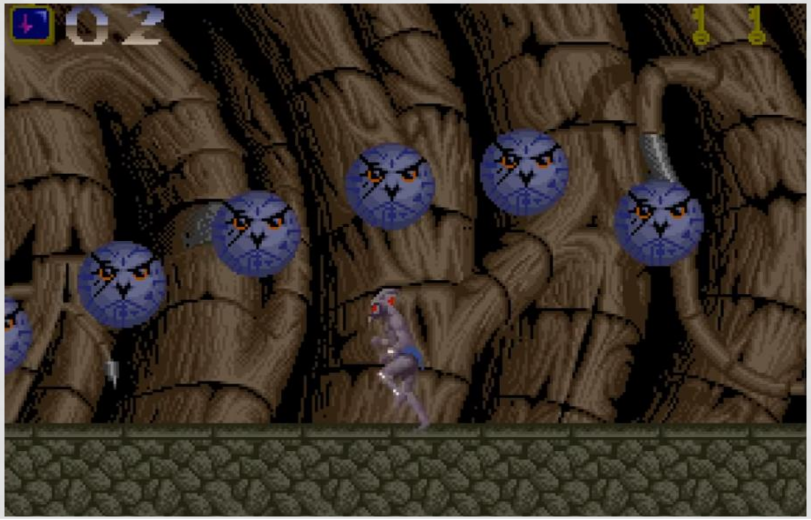
EXAMPLE #3

- End result:

  https://www.shadertoy.com/view/3lBSzK

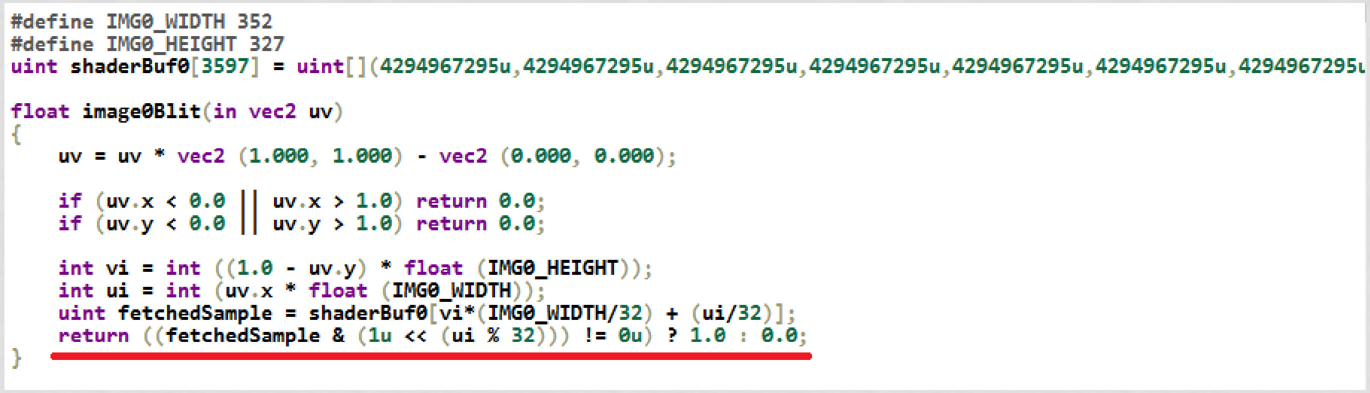
  (By yours truly)
- Homage to this scene from Shadow of the Beast I





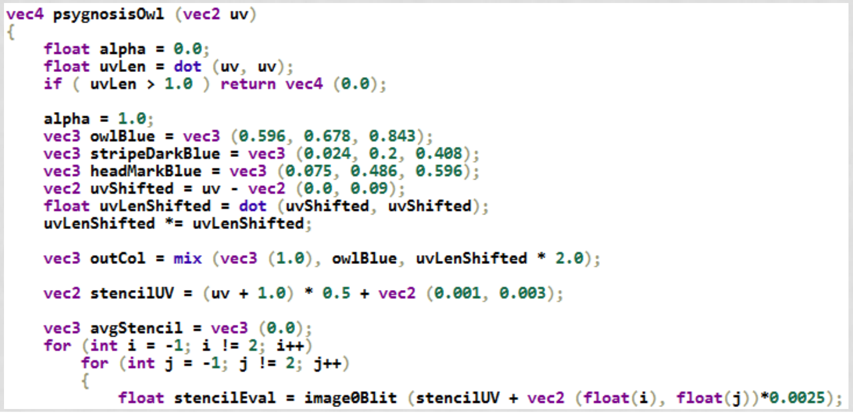
EXAMPLE #3

- Stencil decode is much simpler
  - Just 1 bit we're interested in

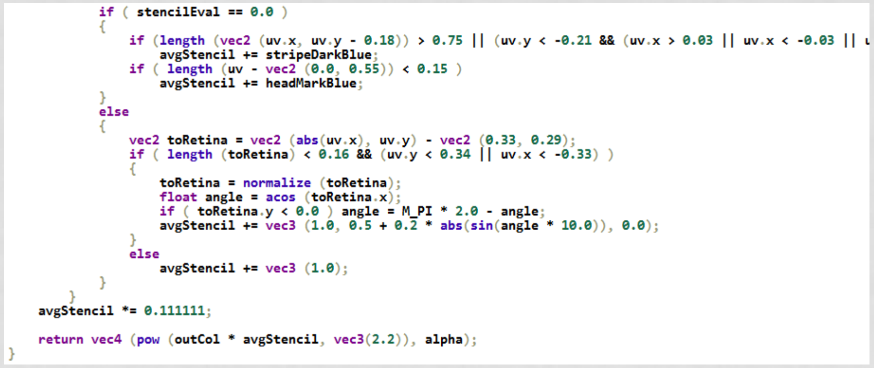


```
#define IMG0_WIDTH 352
#define IMG0_HEIGHT 327
uint shaderBuf0[3597] = uint[](4294967295u,4294967295u,4294967295u,4294967295u,4294967295u,4294967295u,4294967295u

float image0Blit(in vec2 uv)
{
    uv = uv * vec2 (1.000, 1.000) - vec2 (0.000, 0.000);

    if (uv.x < 0.0 || uv.x > 1.0) return 0.0;
    if (uv.y < 0.0 || uv.y > 1.0) return 0.0;

    int vi = int ((1.0 - uv.y) * float (IMG0_HEIGHT));
    int ui = int (uv.x * float (IMG0_WIDTH));
    uint fetchedSample = shaderBuf0[vi*(IMG0_WIDTH/32) + (ui/32)];
    return ((fetchedSample & (1u << (ui % 32))) != 0u) ? 1.0 : 0.0;
}
```

- Image reconstruction is much more involved:



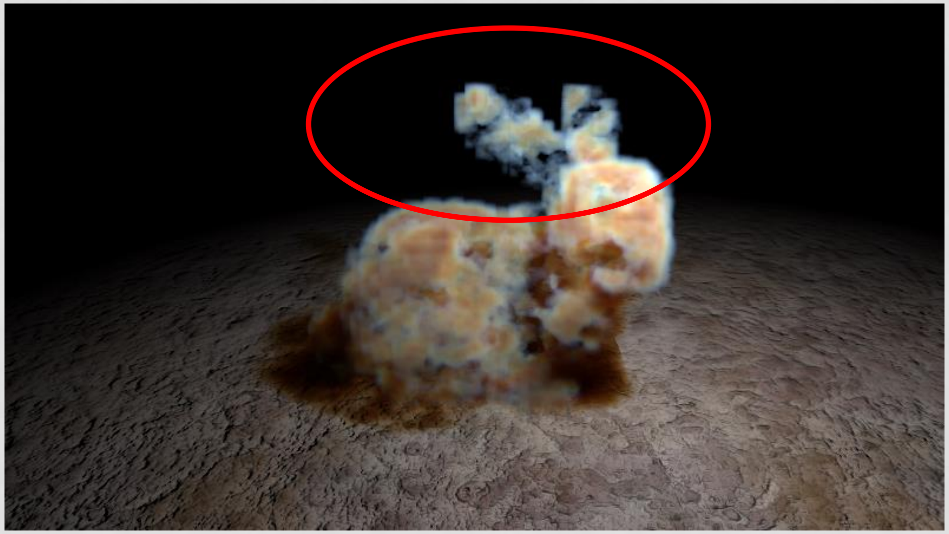
```
vec4 psygnosisOwl (vec2 uv)
{
    float alpha = 0.0;
    float uvLen = dot (uv, uv);
    if ( uvLen > 1.0 ) return vec4 (0.0);

    alpha = 1.0;
    vec3 owlBlue = vec3 (0.596, 0.678, 0.843);
    vec3 stripeDarkBlue = vec3 (0.024, 0.2, 0.408);
    vec3 headMarkBlue = vec3 (0.075, 0.486, 0.596);
    vec2 uvShifted = uv - vec2 (0.0, 0.09);
    float uvLenShifted = dot (uvShifted, uvShifted);
    uvLenShifted *= uvLenShifted;

    vec3 outCol = mix (vec3 (1.0), owlBlue, uvLenShifted * 2.0);

    vec2 stencilUV = (uv + 1.0) * 0.5 + vec2 (0.001, 0.003);

    vec3 avgStencil = vec3 (0.0);
    for (int i = -1; i != 2; i++)
        for (int j = -1; j != 2; j++)
        {
            float stencilEval = image0Blit (stencilUV + vec2 (float(i), float(j))*0.0025);
```



```
            if ( stencilEval == 0.0 )
            {
                if (length (vec2 (uv.x, uv.y - 0.18)) > 0.75 || (uv.y < -0.21 && (uv.x > 0.03 || uv.x < -0.03 || u
                    avgStencil += stripeDarkBlue;
                if ( length (uv - vec2 (0.0, 0.55)) < 0.15 )
                    avgStencil += headMarkBlue;
            }
            else
            {
                vec2 toRetina = vec2 (abs(uv.x), uv.y) - vec2 (0.33, 0.29);
                if ( length (toRetina) < 0.16 && (uv.y < 0.34 || uv.x < -0.33) )
                {
                    toRetina = normalize (toRetina);
                    float angle = acos (toRetina.x);
                    if ( toRetina.y < 0.0 ) angle = M_PI * 2.0 - angle;
                    avgStencil += vec3 (1.0, 0.5 + 0.2 * abs(sin(angle * 10.0)), 0.0);
                }
                else
                    avgStencil += vec3 (1.0);
            }
        }
    avgStencil *= 0.111111;

    return vec4 (pow (outCol * avgStencil, vec3(2.2)), alpha);
}
```

CHALLENGE #2

- What about geometry?
- Example:
  - The Stanford bunny
  - Used by Sebastien Hillaire to demonstrate improved delta-tracking integral (https://www.shadertoy.com/view/MdlyDs)
- Coarse around the ears:
  - Can we do better?



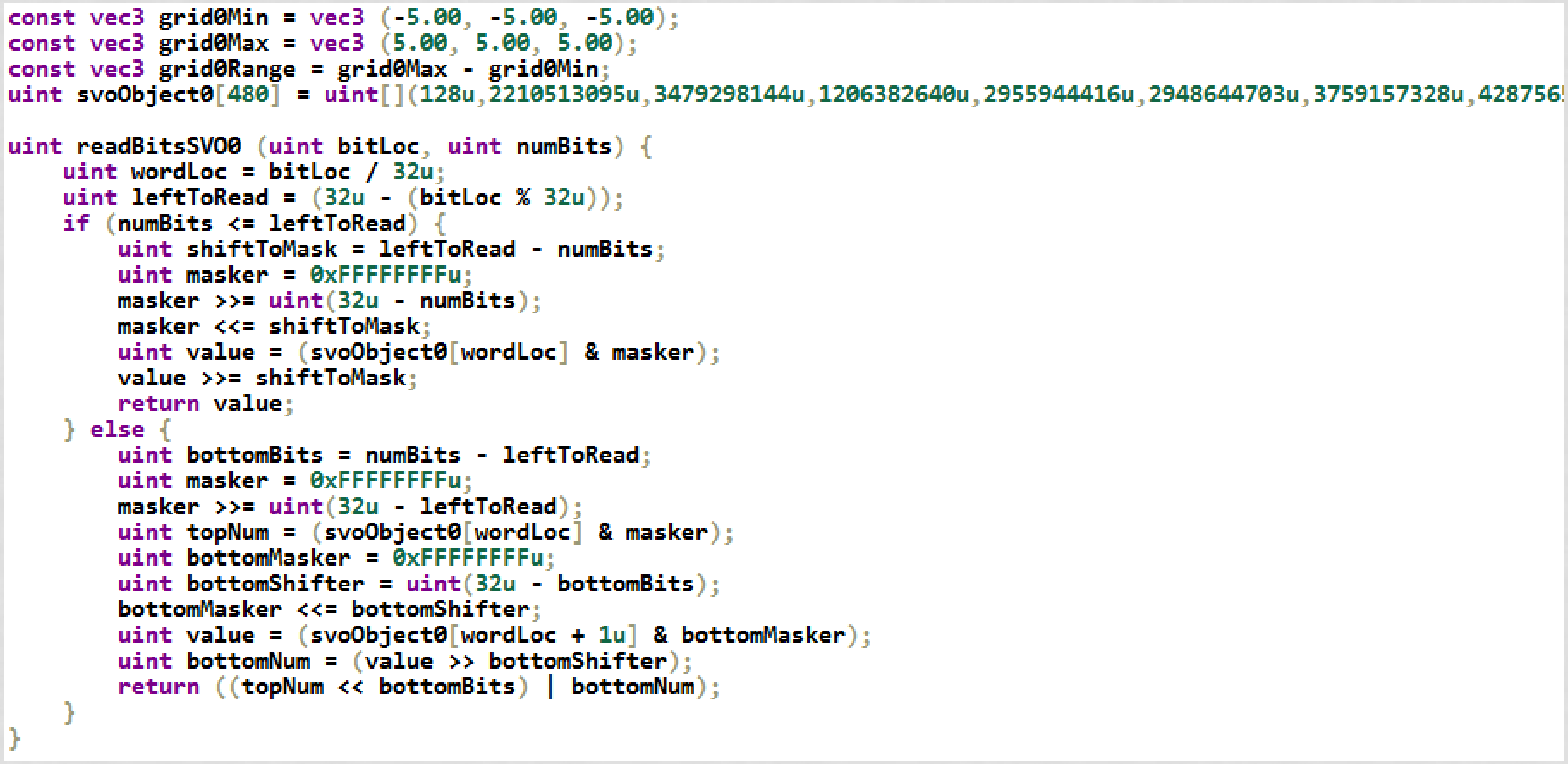
CHALLENGE #2

- Yes, we can!
- Encode entire geometry as a Sparse Voxel Octree
- Waste no bits on empty top and mid-level bricks
- You can trace this, live!
- We packed the bunny and had room for 2 more!
- https://www.shadertoy.com/view/dlBGRc

(By yours truly)

CHALLENGE #2

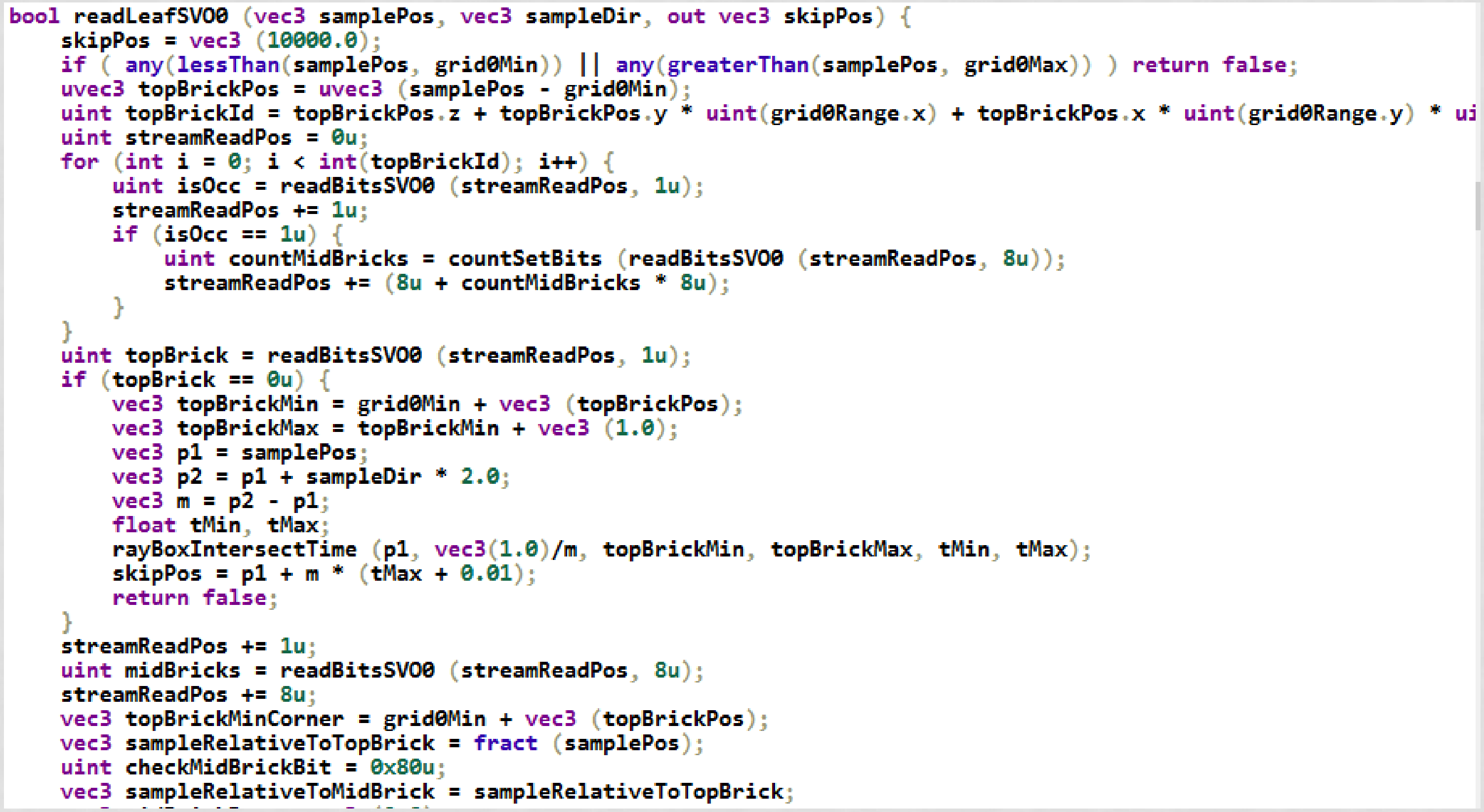
- For this we actually need a bitstream reader:



```
const vec3 grid0Min = vec3 (-5.00, -5.00, -5.00);
const vec3 grid0Max = vec3 (5.00, 5.00, 5.00);
const vec3 grid0Range = grid0Max - grid0Min;
uint svoObject0[480] = uint[](128u,2210513095u,3479298144u,1206382640u,2955944416u,2948644703u,3759157328u,428756

uint readBitsSVO0 (uint bitLoc, uint numBits) {
    uint wordLoc = bitLoc / 32u;
    uint leftToRead = (32u - (bitLoc % 32u));
    if (numBits <= leftToRead) {
        uint shiftToMask = leftToRead - numBits;
        uint masker = 0xFFFFFFFFu;
        masker >>= uint(32u - numBits);
        masker <<= shiftToMask;
        uint value = (svoObject0[wordLoc] & masker);
        value >>= shiftToMask;
        return value;
    } else {
        uint bottomBits = numBits - leftToRead;
        uint masker = 0xFFFFFFFFu;
        masker >>= uint(32u - leftToRead);
        uint topNum = (svoObject0[wordLoc] & masker);
        uint bottomMasker = 0xFFFFFFFFu;
        uint bottomShifter = uint(32u - bottomBits);
        bottomMasker <<= bottomShifter;
        uint value = (svoObject0[wordLoc + 1u] & bottomMasker);
        uint bottomNum = (value >> bottomShifter);
        return ((topNum << bottomBits) | bottomNum);
    }
}
```

CHALLENGE #2

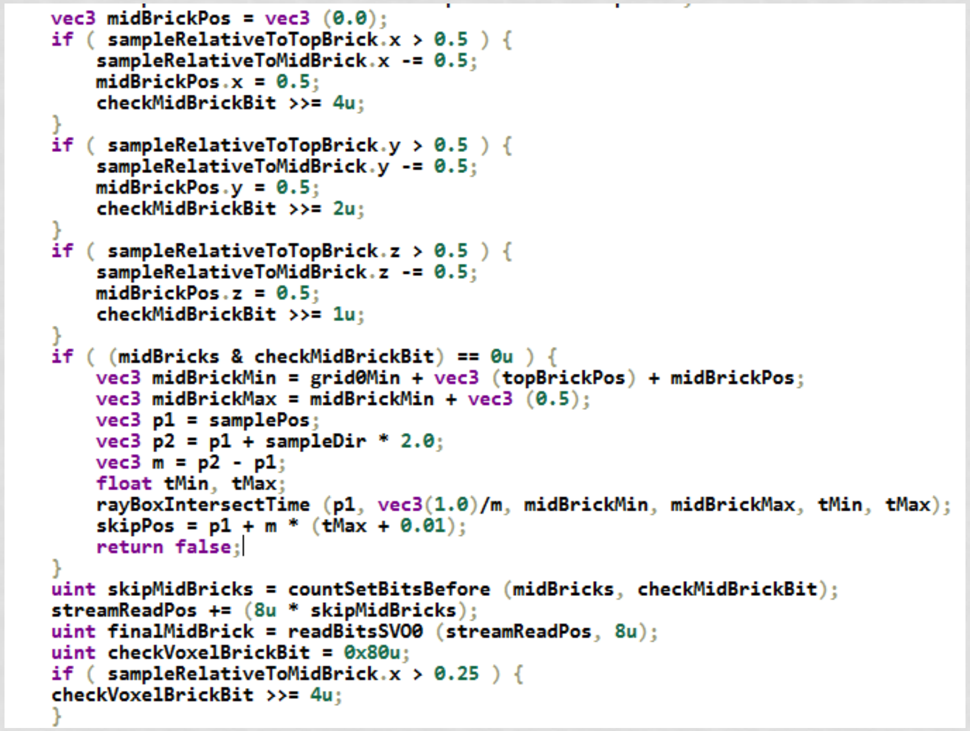
- Live trace via hole-skipping ray-box (slab) intersection tests:



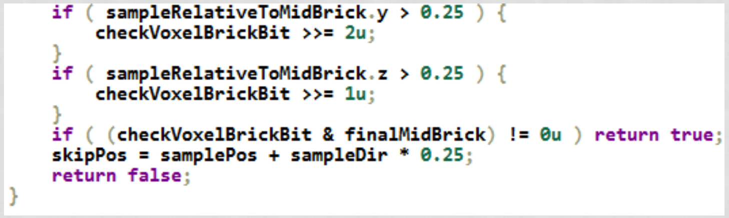
```
bool readLeafSVO0 (vec3 samplePos, vec3 sampleDir, out vec3 skipPos) {
    skipPos = vec3 (10000.0);
    if ( any(lessThan(samplePos, grid0Min)) || any(greaterThan(samplePos, grid0Max)) ) return false;
    uvec3 topBrickPos = uvec3 (samplePos - grid0Min);
    uint topBrickId = topBrickPos.z + topBrickPos.y * uint(grid0Range.x) + topBrickPos.x * uint(grid0Range.y) * ui
    uint streamReadPos = 0u;
    for (int i = 0; i < int(topBrickId); i++) {
        uint isOcc = readBitsSVO0 (streamReadPos, 1u);
        streamReadPos += 1u;
        if (isOcc == 1u) {
            uint countMidBricks = countSetBits (readBitsSVO0 (streamReadPos, 8u));
            streamReadPos += (8u + countMidBricks * 8u);
        }
    }
    uint topBrick = readBitsSVO0 (streamReadPos, 1u);
    if (topBrick == 0u) {
        vec3 topBrickMin = grid0Min + vec3 (topBrickPos);
        vec3 topBrickMax = topBrickMin + vec3 (1.0);
        vec3 p1 = samplePos;
        vec3 p2 = p1 + sampleDir * 2.0;
        vec3 m = p2 - p1;
        float tMin, tMax;
        rayBoxIntersectTime (p1, vec3(1.0)/m, topBrickMin, topBrickMax, tMin, tMax);
        skipPos = p1 + m * (tMax + 0.01);
        return false;
    }
    streamReadPos += 1u;
    uint midBricks = readBitsSVO0 (streamReadPos, 8u);
    streamReadPos += 8u;
    vec3 topBrickMinCorner = grid0Min + vec3 (topBrickPos);
    vec3 sampleRelativeToTopBrick = fract (samplePos);
    uint checkMidBrickBit = 0x80u;
    vec3 sampleRelativeToMidBrick = sampleRelativeToTopBrick;
```

CHALLENGE #2

- Read octree nodes only if they’re occupied (i.e. encountered a set bit).
  - Otherwise, skip the size of the level you’re at (top or mid-level brick):



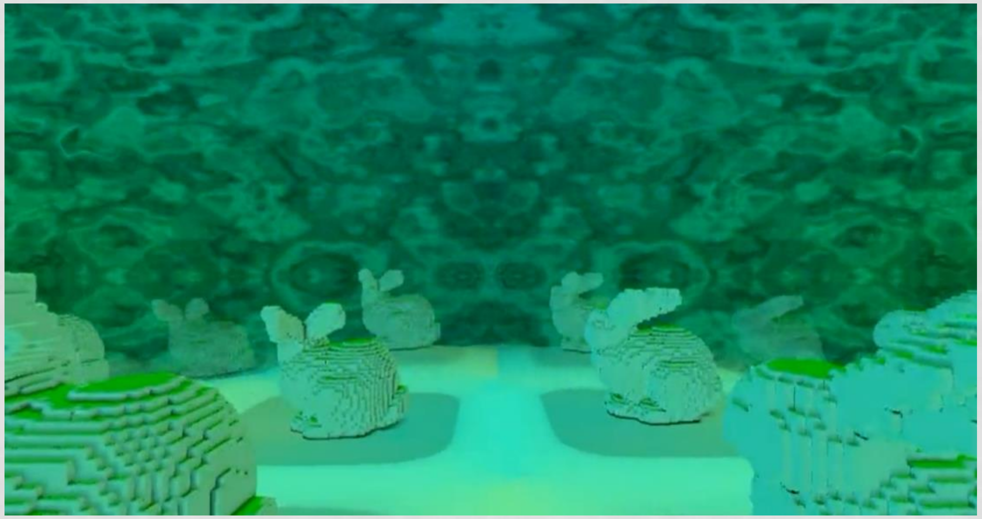
```
vec3 midBrickPos = vec3 (0.0);
if ( sampleRelativeToTopBrick.x > 0.5 ) {
    sampleRelativeToMidBrick.x -= 0.5;
    midBrickPos.x = 0.5;
    checkMidBrickBit >>= 4u;
}
if ( sampleRelativeToTopBrick.y > 0.5 ) {
    sampleRelativeToMidBrick.y -= 0.5;
    midBrickPos.y = 0.5;
    checkMidBrickBit >>= 2u;
}
if ( sampleRelativeToTopBrick.z > 0.5 ) {
    sampleRelativeToMidBrick.z -= 0.5;
    midBrickPos.z = 0.5;
    checkMidBrickBit >>= 1u;
}
if ( (midBricks & checkMidBrickBit) == 0u ) {
    vec3 midBrickMin = grid0Min + vec3 (topBrickPos) + midBrickPos;
    vec3 midBrickMax = midBrickMin + vec3 (0.5);
    vec3 p1 = samplePos;
    vec3 p2 = p1 + sampleDir * 2.0;
    vec3 m = p2 - p1;
    float tMin, tMax;
    rayBoxIntersectTime (p1, vec3(1.0)/m, midBrickMin, midBrickMax, tMin, tMax);
    skipPos = p1 + m * (tMax + 0.01);
    return false;
}
uint skipMidBricks = countSetBitsBefore (midBricks, checkMidBrickBit);
streamReadPos += (8u * skipMidBricks);
uint finalMidBrick = readBitsSVO0 (streamReadPos, 8u);
uint checkVoxelBrickBit = 0x80u;
if ( sampleRelativeToMidBrick.x > 0.25 ) {
checkVoxelBrickBit >>= 4u;
}
```



```
    if ( sampleRelativeToMidBrick.y > 0.25 ) {
        checkVoxelBrickBit >>= 2u;
    }
    if ( sampleRelativeToMidBrick.z > 0.25 ) {
        checkVoxelBrickBit >>= 1u;
    }
    if ( (checkVoxelBrickBit & finalMidBrick) != 0u ) return true;
    skipPos = samplePos + sampleDir * 0.25;
    return false;
}
```

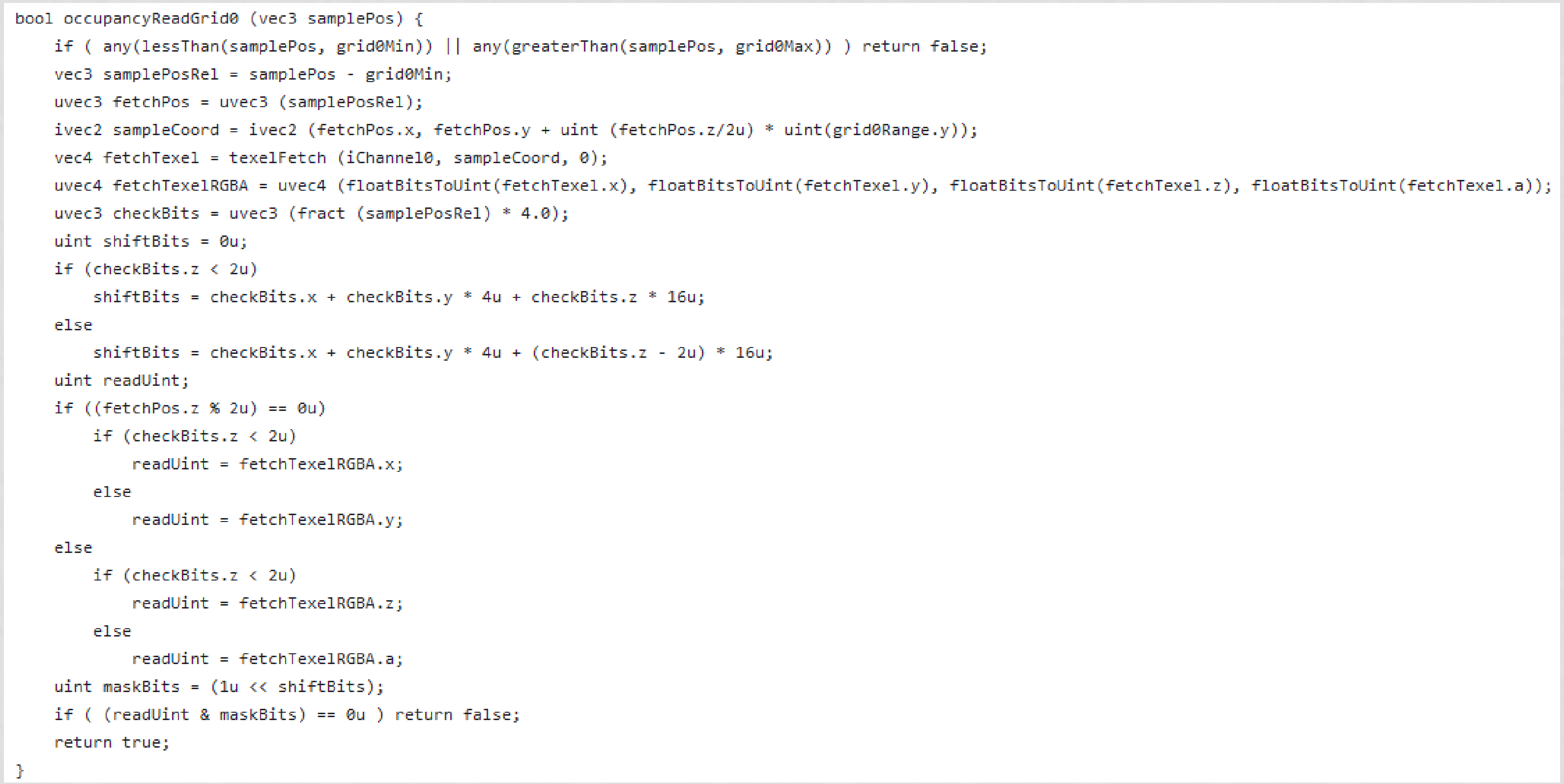
CHALLENGE #2

- Does this work at scale?
  - Turns out: no (lol!)
- First attempt by yours truly to combine with SDFs
  - Too slow
  - Pros:
    - Smooth corners
  - Cons:
    - Too many reads (3x3x3 fetches to construct a local rounded box SDF)
    - All value in hole-skipping gone



CHALLENGE #2

- Code available on Shadertoy-utils (by yours truly):
  - https://github.com/toomuchvoltage/shadertoy-utils
  - Below code executed 3x3x3 times! (Oof…)



```
bool occupancyReadGrid0 (vec3 samplePos) {
    if ( any(lessThan(samplePos, grid0Min)) || any(greaterThan(samplePos, grid0Max)) ) return false;
    vec3 samplePosRel = samplePos - grid0Min;
    uvec3 fetchPos = uvec3 (samplePosRel);
    ivec2 sampleCoord = ivec2 (fetchPos.x, fetchPos.y + uint (fetchPos.z/2u) * uint(grid0Range.y));
    vec4 fetchTexel = texelFetch (iChannel0, sampleCoord, 0);
    uvec4 fetchTexelRGBA = uvec4 (floatBitsToUint(fetchTexel.x), floatBitsToUint(fetchTexel.y), floatBitsToUint(fetchTexel.z), floatBitsToUint(fetchTexel.a));
    uvec3 checkBits = uvec3 (fract (samplePosRel) * 4.0);
    uint shiftBits = 0u;
    if (checkBits.z < 2u)
        shiftBits = checkBits.x + checkBits.y * 4u + checkBits.z * 16u;
    else
        shiftBits = checkBits.x + checkBits.y * 4u + (checkBits.z - 2u) * 16u;
    uint readUint;
    if ((fetchPos.z % 2u) == 0u)
        if (checkBits.z < 2u)
            readUint = fetchTexelRGBA.x;
        else
            readUint = fetchTexelRGBA.y;
    else
        if (checkBits.z < 2u)
            readUint = fetchTexelRGBA.z;
        else
            readUint = fetchTexelRGBA.a;
    uint maskBits = (1u << shiftBits);
    if ( (readUint & maskBits) == 0u ) return false;
    return true;
}
```

CHALLENGE #2

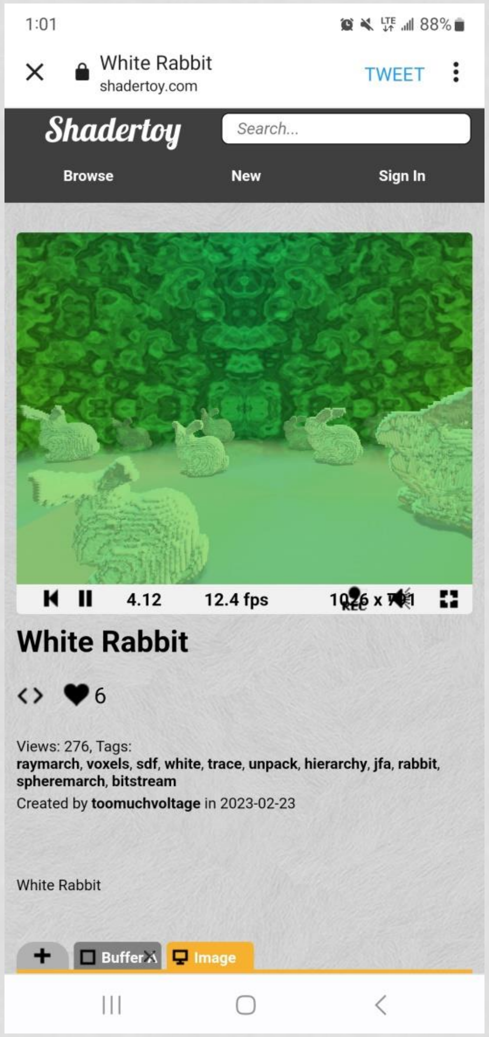
- Second attempt:
  - Encode the entire bunny as a distance field
- Three step approach:

1. Expand SVO into tiles in a sub-region of a floating point target
2. Generate distance field via JFA
   - Keep going until offsetPower is -1.0
3. Compact SDF output to use as little memory as possible
   - 40x40x40 bunny only needs a (40, 40*3) RGBA32F sub-image

CHALLENGE #2

- Bingo!
  - Even runs on my phone: S22 Ultra
- https://www.shadertoy.com/view/cs3GRH

(By yours truly)



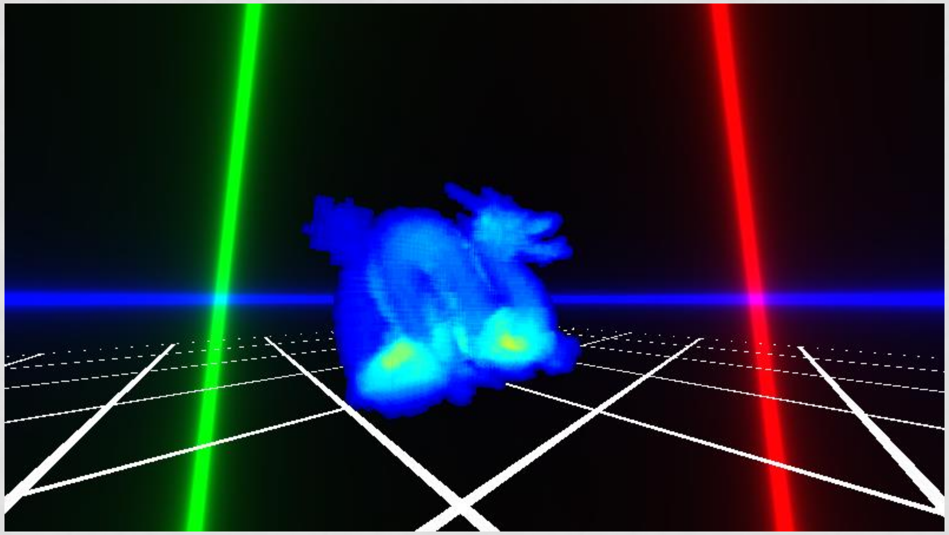
Views: 276, Tags: raymarch, voxels, sdf, white, trace, unpack, hierarchy, jfa, rabbit, spheremarch, bitstream

Created by toomuchvoltage in 2023-02-23

White Rabbit

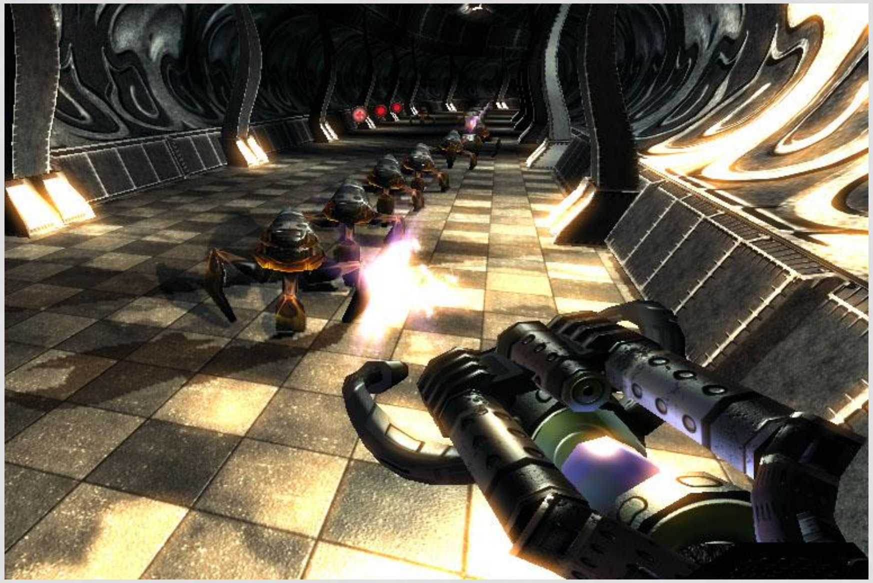
CHALLENGE #2

- Honorable mention:
- RLE encoded Stanford dragon by Anton Schreiner
  - https://www.shadertoy.com/view/tlSSWD
  - There is no option to live-trace here though! (would be too slow)
  - Expanded version would not hole-skip either.



CHALLENGE #2

- Have we seen this sort of runtime expansion before?
- Yes: ***.kkrieger by .theprodukkt***
- Entire video game in <100KB
- All geometry is CSG
- All textures are encoded as successive brush strokes
- Expanded into VRAM at runtime



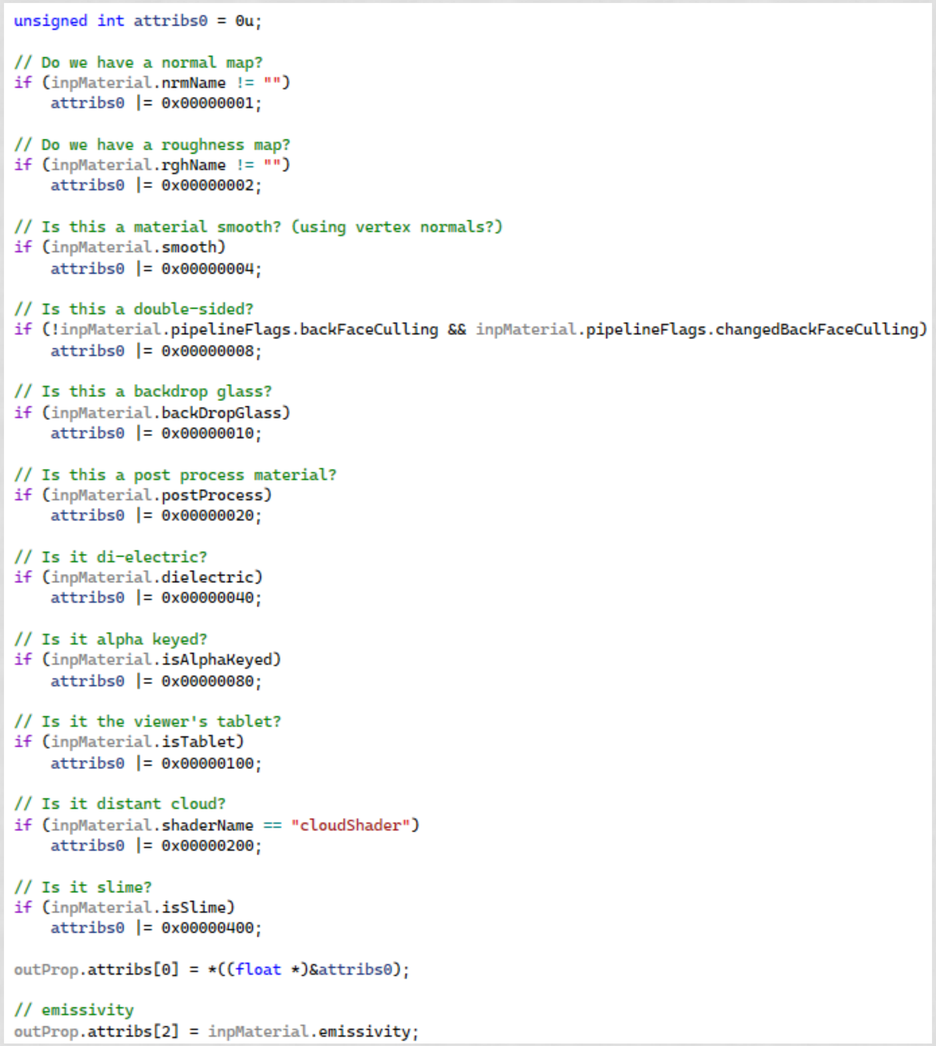
CHALLENGE #2

- **NOTE:** if you want to ship materials with the bunny, encode as swatch bits following the leaf brick bits
- We can access them as we encounter intersections
- Another rule by Marie Kondo:
  - Store items based on frequency of use!
  - In GPU optimization this is spatial locality for spatially coherent access
  - Results in less cache thrashing

10 Amazing Tips from Tidying Up with Marie Kondo
#8 Store Items Based on Frequency of Use
msmojo

CHALLENGE #3

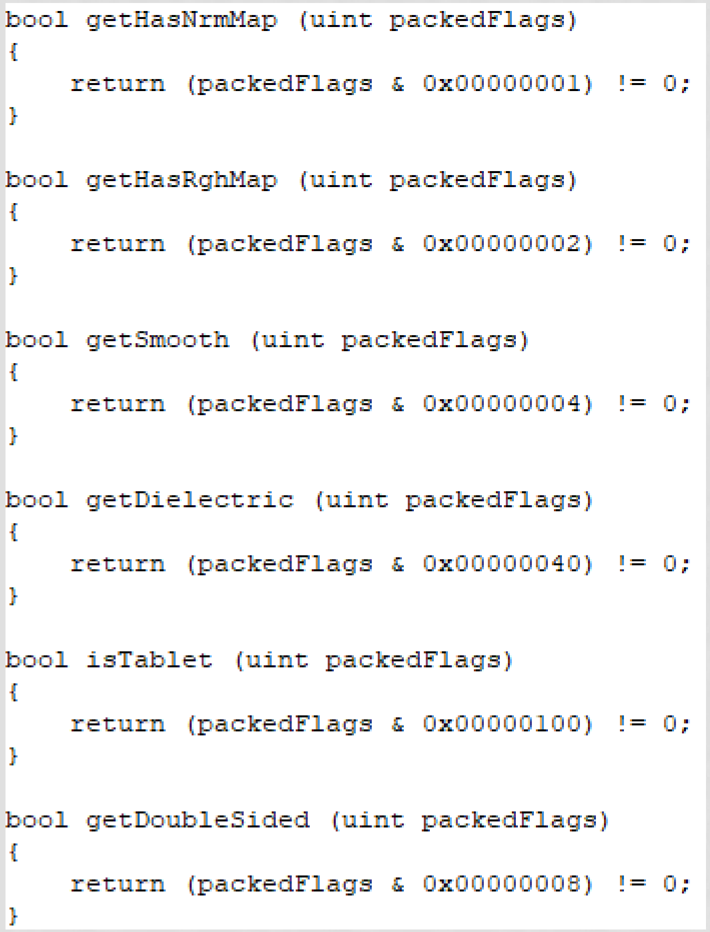
- What about games? Can this help our title?
- Yes!
  - Encode instance properties in your instance property buffers (UAVs/SSBOs) as bits in a bitfield



```
unsigned int attribs0 = 0u;

// Do we have a normal map?
if (inpMaterial.nrmName != "")
    attribs0 |= 0x00000001;

// Do we have a roughness map?
if (inpMaterial.rghName != "")
    attribs0 |= 0x00000002;

// Is this a material smooth? (using vertex normals?)
if (inpMaterial.smooth)
    attribs0 |= 0x00000004;

// Is this a double-sided?
if (!inpMaterial.pipelineFlags.backFaceCulling && inpMaterial.pipelineFlags.changedBackFaceCulling)
    attribs0 |= 0x00000008;

// Is this a backdrop glass?
if (inpMaterial.backDropGlass)
    attribs0 |= 0x00000010;

// Is this a post process material?
if (inpMaterial.postProcess)
    attribs0 |= 0x00000020;

// Is it di-electric?
if (inpMaterial.dielectric)
    attribs0 |= 0x00000040;

// Is it alpha keyed?
if (inpMaterial.isAlphaKeyed)
    attribs0 |= 0x00000080;

// Is it the viewer's tablet?
if (inpMaterial.isTablet)
    attribs0 |= 0x00000100;

// Is it distant cloud?
if (inpMaterial.shaderName == "cloudShader")
    attribs0 |= 0x00000200;

// Is it slime?
if (inpMaterial.isSlime)
    attribs0 |= 0x00000400;

outProp.attribs[0] = *((float *)&attribs0);

// emissivity
outProp.attribs[2] = inpMaterial.emissivity;
```

CHALLENGE #3

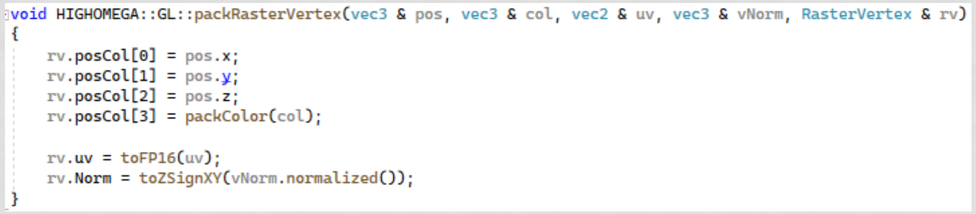
- Decode inside shader



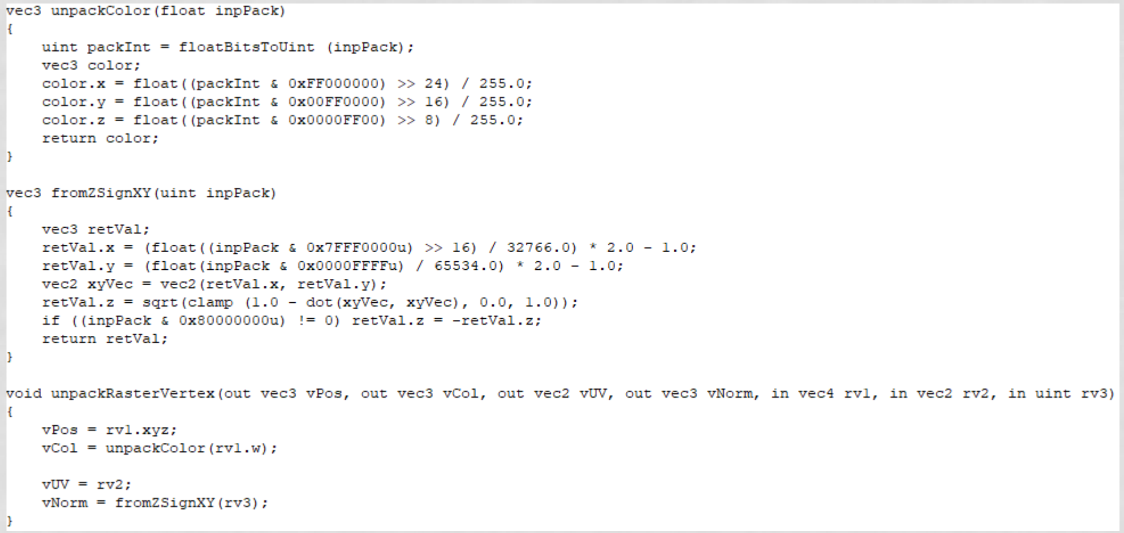
```
bool getHasNrmMap (uint packedFlags)
{
    return (packedFlags & 0x00000001) != 0;
}

bool getHasRghMap (uint packedFlags)
{
    return (packedFlags & 0x00000002) != 0;
}

bool getSmooth (uint packedFlags)
{
    return (packedFlags & 0x00000004) != 0;
}

bool getDielectric (uint packedFlags)
{
    return (packedFlags & 0x00000040) != 0;
}

bool isTablet (uint packedFlags)
{
    return (packedFlags & 0x00000100) != 0;
}

bool getDoubleSided (uint packedFlags)
{
    return (packedFlags & 0x00000008) != 0;
}
```

CHALLENGE #3

- What else?
  - We can pack and unpack data into vertices so as to push more geometry
  - We can store normal as sign of Z, X and Y and infer via sqrt



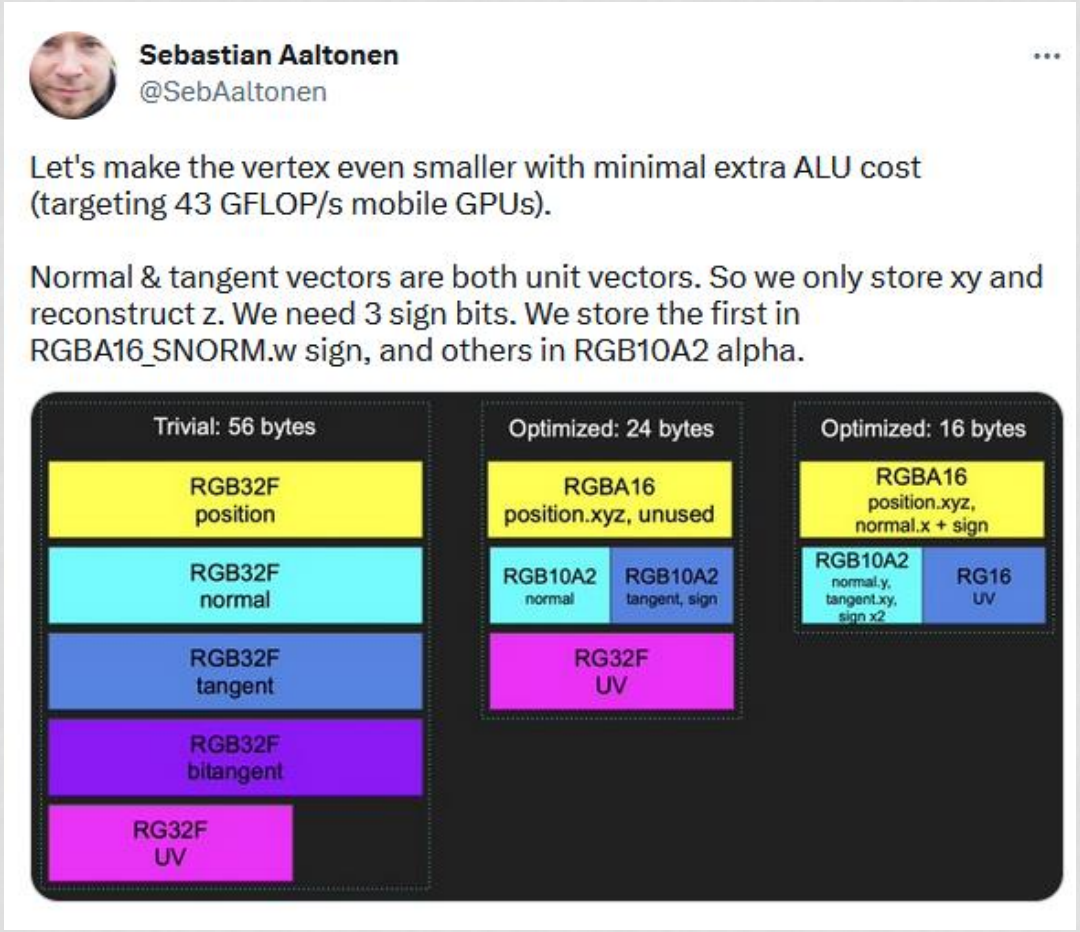
```
void HIGHOMEGA::GL::packRasterVertex(vec3 & pos, vec3 & col, vec2 & uv, vec3 & vNorm, RasterVertex & rv)
{
    rv.posCol[0] = pos.x;
    rv.posCol[1] = pos.y;
    rv.posCol[2] = pos.z;
    rv.posCol[3] = packColor(col);

    rv.uv = toFP16(uv);
    rv.Norm = toZSignXY(vNorm.normalized());
}
```



```
vec3 unpackColor(float inpPack)
{
    uint packInt = floatBitsToUint (inpPack);
    vec3 color;
    color.x = float((packInt & 0xFF000000) >> 24) / 255.0;
    color.y = float((packInt & 0x00FF0000) >> 16) / 255.0;
    color.z = float((packInt & 0x0000FF00) >> 8) / 255.0;
    return color;
}

vec3 fromZSignXY(uint inpPack)
{
    vec3 retVal;
    retVal.x = (float((inpPack & 0x7FFF0000u) >> 16) / 32766.0) * 2.0 - 1.0;
    retVal.y = (float(inpPack & 0x0000FFFFu) / 65534.0) * 2.0 - 1.0;
    vec2 xyVec = vec2(retVal.x, retVal.y);
    retVal.z = sqrt(clamp (1.0 - dot(xyVec, xyVec), 0.0, 1.0));
    if ((inpPack & 0x80000000u) != 0) retVal.z = -retVal.z;
    return retVal;
}

void unpackRasterVertex(out vec3 vPos, out vec3 vCol, out vec2 vUV, out vec3 vNorm, in vec4 rv1, in vec2 rv2, in uint rv3)
{
    vPos = rv1.xyz;
    vCol = unpackColor(rv1.w);

    vUV = rv2;
    vNorm = fromZSignXY(rv3);
}
```

CHALLENGE #3

- Try hard enough and you should hit 16 bytes per vertex! ;)
  - https://twitter.com/SebAaltonen/status/1515735247928930311



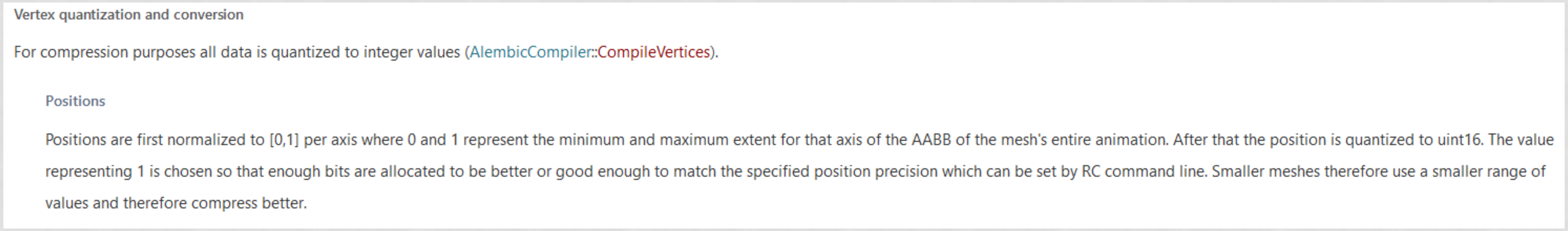
Sebastian Aaltonen
@SebAaltonen

Let's make the vertex even smaller with minimal extra ALU cost (targeting 43 GFLOP/s mobile GPUs).

Normal & tangent vectors are both unit vectors. So we only store xy and reconstruct z. We need 3 sign bits. We store the first in RGBA16_SNORM.w sign, and others in RGB10A2 alpha.

CHALLENGE #3

- Vertex positions in Ryse: Son of Rome were compressed to represent a fraction of the mesh AABB:



Vertex quantization and conversion

For compression purposes all data is quantized to integer values (AlembicCompiler::CompileVertices).

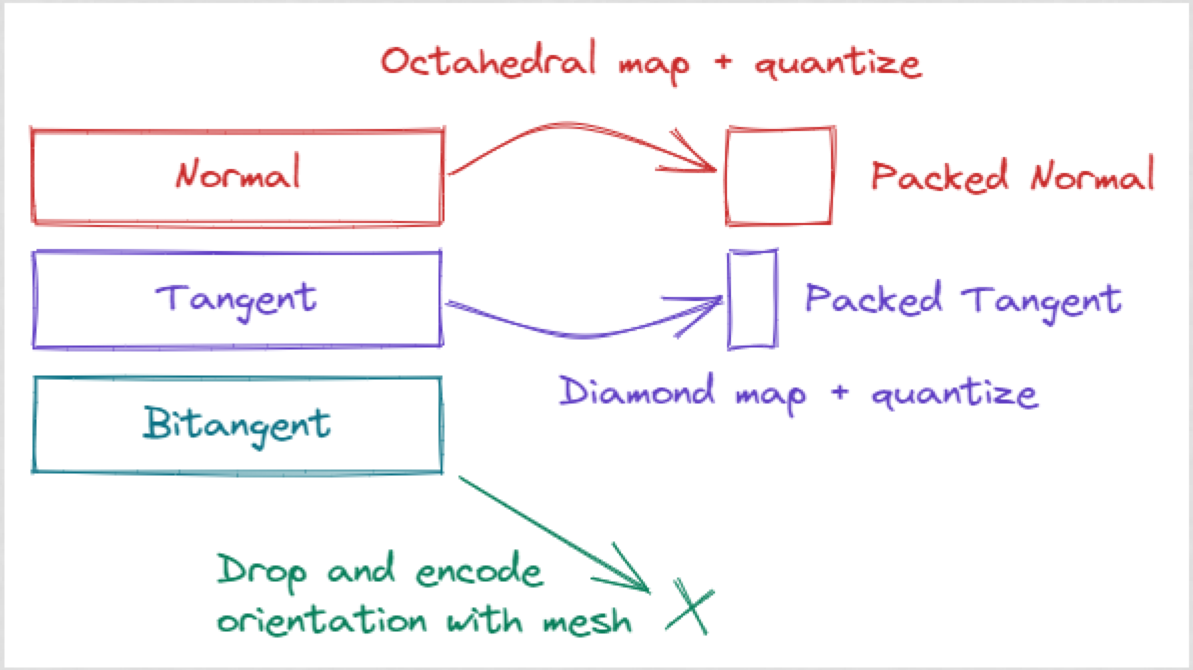
Positions

Positions are first normalized to [0,1] per axis where 0 and 1 represent the minimum and maximum extent for that axis of the AABB of the mesh's entire animation. After that the position is quantized to uint16. The value representing 1 is chosen so that enough bits are allocated to be better or good enough to match the specified position precision which can be set by RC command line. Smaller meshes therefore use a smaller range of values and therefore compress better.

- Presentation missing from the web
- But instructions on how to do this in CryEngine is available here: https://docs.cryengine.com/display/CEMANUAL/Geom+Cache+Technical+Overview
- Entire tangent space was also encoded as a quaternion with some additional info. See q-tangent: https://dl.acm.org/doi/abs/10.1145/2037826.2037841

CHALLENGE #3

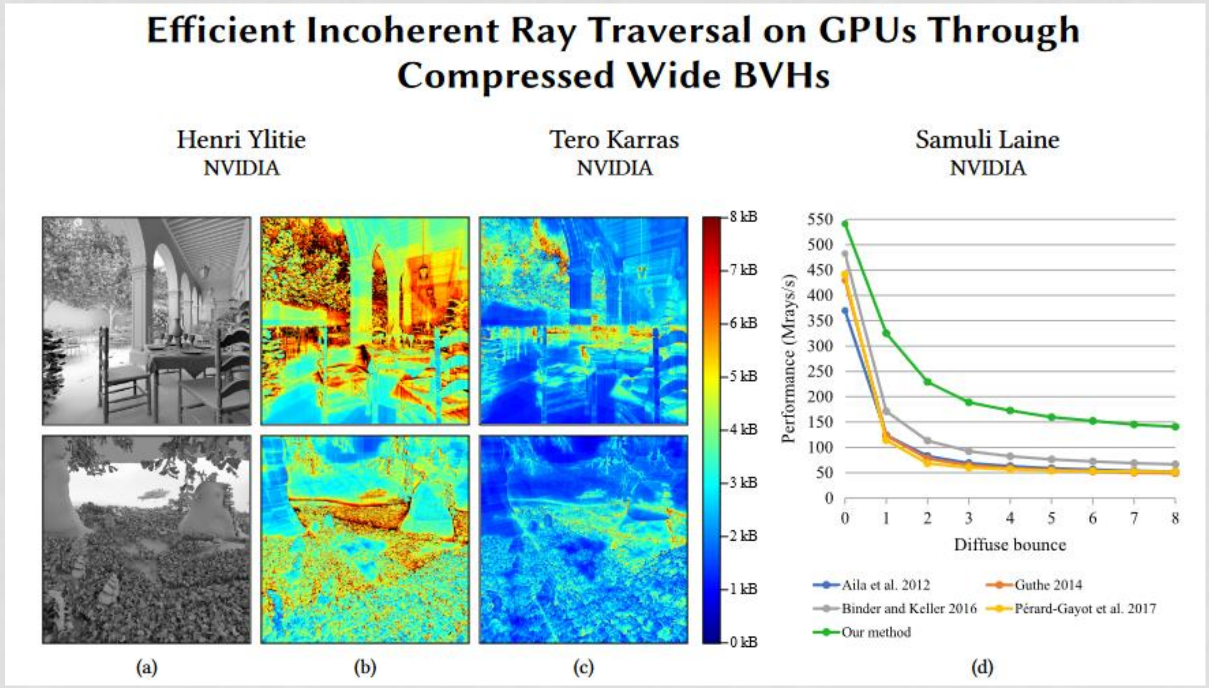
- Even more compact tangent space representation:
  - https://www.jeremyong.com/graphics/2023/01/09/tangent-spaces-and-diamond-encoding/



CHALLENGE #3

- This is efficient in path-tracing too!
  - Ylitie2017 not only encodes vertex positions as fractions of leaf AABBs, but makes internal node AABBs fractions of *each other*:

https://research.nvidia.com/sites/default/files/publications/ylitie2017hpg-paper.pdf



Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs

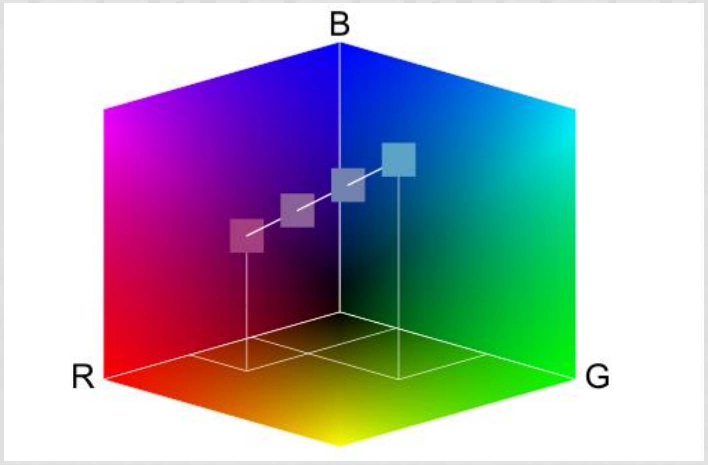
Henri Ylitie, NVIDIA
Tero Karras, NVIDIA
Samuli Laine, NVIDIA

CHALLENGE #3

- Every leaf node in Teardown uses an 8-bit index to look into a color palette:
- Full tech talk here:
  - https://www.youtube.com/watch?v=0VzE8ROwC58
- Many *many* ways to spark joy!

CHALLENGE #3

- Even the hardware does this *for you*!
- BC1-7 block compression is all about storing color endpoints and flattening colors as 1Bpp fractions on the line that forms
- https://www.reedbeta.com/blog/understanding-bcn-texture-compression-formats/



THAT IS ALL!

Thank you for listening!

Feel free to reach out:

baktash@toomuchvoltage.com

@toomuchvoltage