GPU TECHNOLOGY CONFERENCE

DirectX 11 Overview

Cem Cebenoyan | 9/20/10

PRESENTED BY NVIDIA.

Outline

- Why DirectX 11?
- Direct Compute
- Tessellation
- Multithreaded Command Buffers
- Dynamic Shader Linking
- New texture compression formats
- Read-only depth, conservative oDepth, …

Outline – Why DirectX 11?

- **Why DirectX 11?**
- Direct Compute
- Tessellation
- Multithreaded Command Buffers
- Dynamic Shader Linking
- New texture compression formats
- Read-only depth, conservative oDepth

DirectX 11 Overview

- Focused on high performance and GPU acceleration
- Direct3D 11 is a strict superset of 10 and 10.1
- Runs on downlevel hardware!
  - Down to Direct3D 9 hardware
  - Can ask for a specific D3D_FEATURE_LEVEL
- Available on Vista and Windows 7

Outline - DirectCompute

- Why DirectX 11?
- **Direct Compute**
- Tessellation
- Multithreaded Command Buffers
- Dynamic Shader Linking
- New texture compression formats
- Read-only depth, conservative oDepth, …

DirectCompute

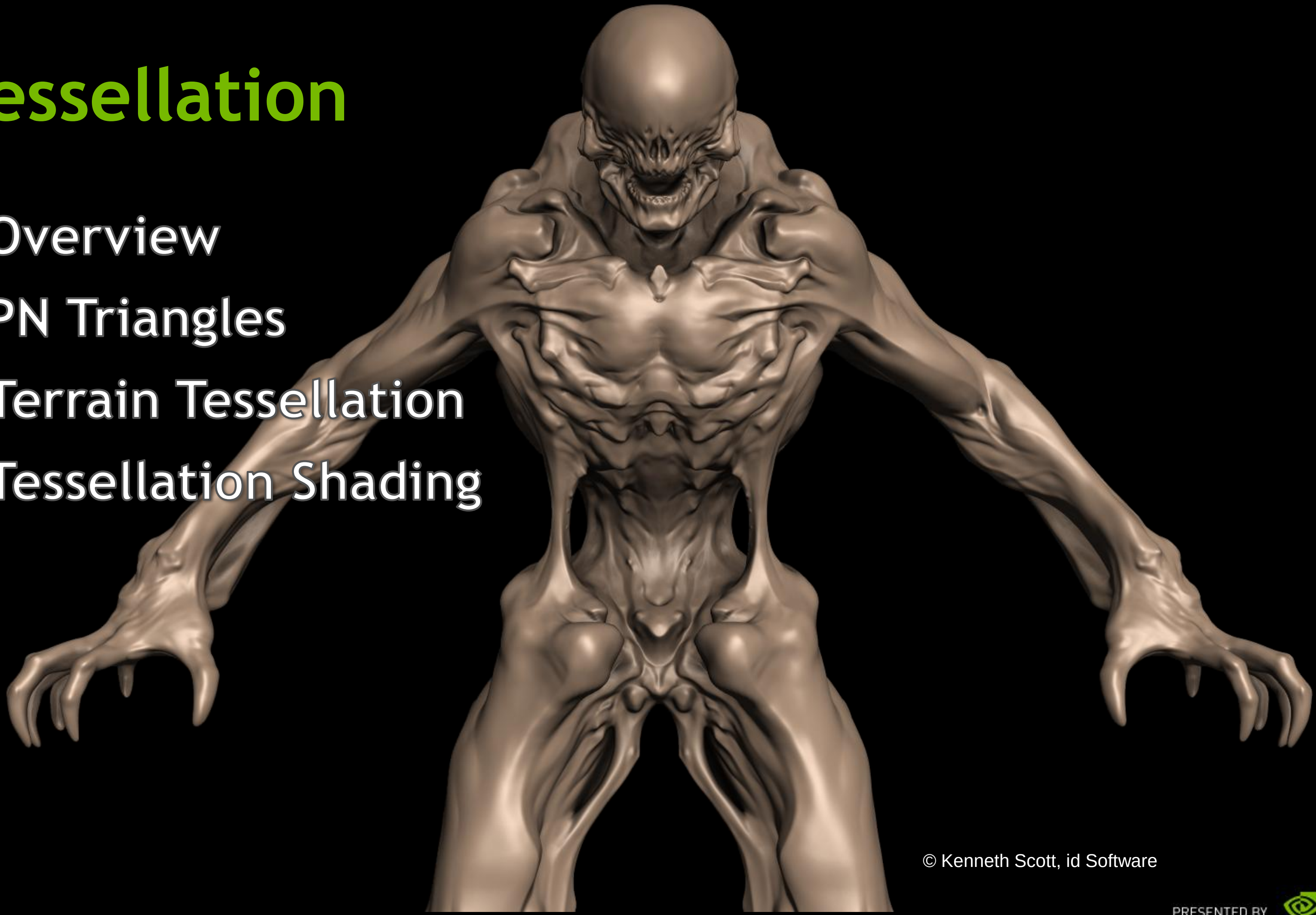
- General purpose programming on CUDA GPUs using compute shaders
- Interoperates with Direct3D
- Uses HLSL
- Not the focus of this talk!

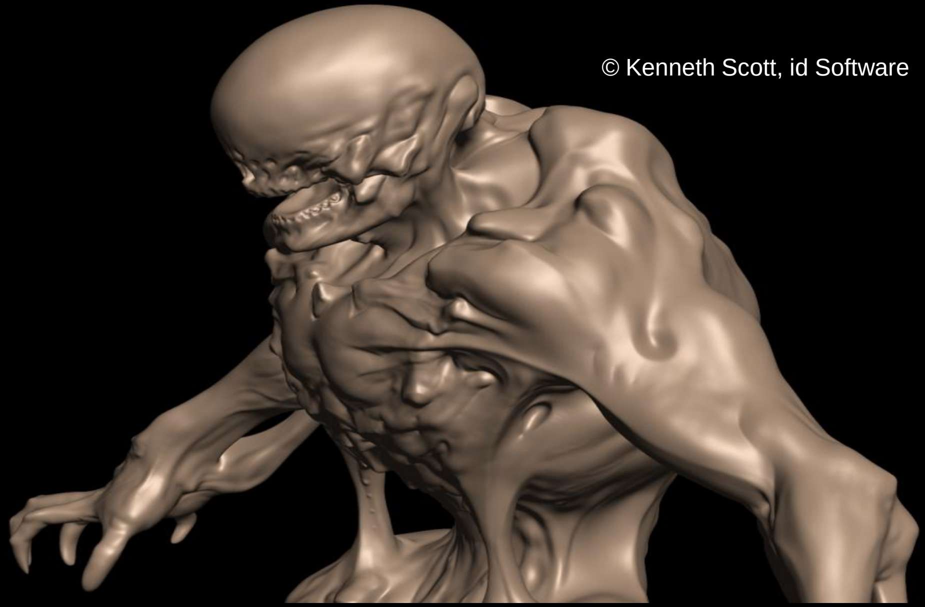
Outline - Tessellation

- Why DirectX 11?
- Direct Compute
- **Tessellation**
- Multithreaded Command Buffers
- Dynamic Shader Linking
- New texture compression formats
- Read-only depth, conservative oDepth, …

Tessellation

- Overview
- PN Triangles
- Terrain Tessellation
- Tessellation Shading

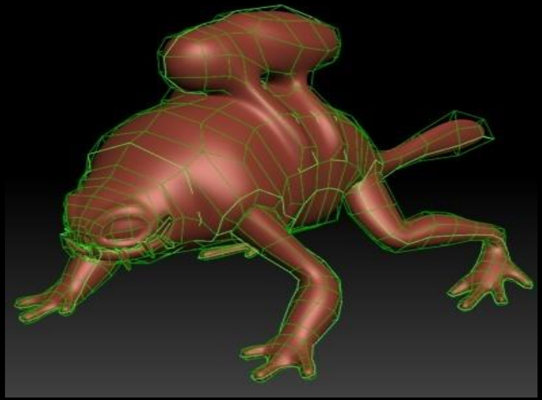




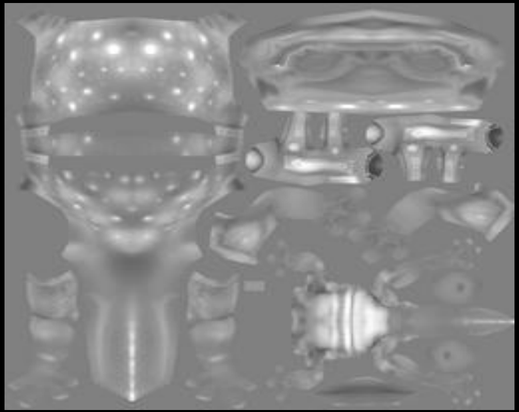
Motivation - Compression

- Save memory and bandwidth
  - Important bottlenecks to rendering highly detailed surfaces

=



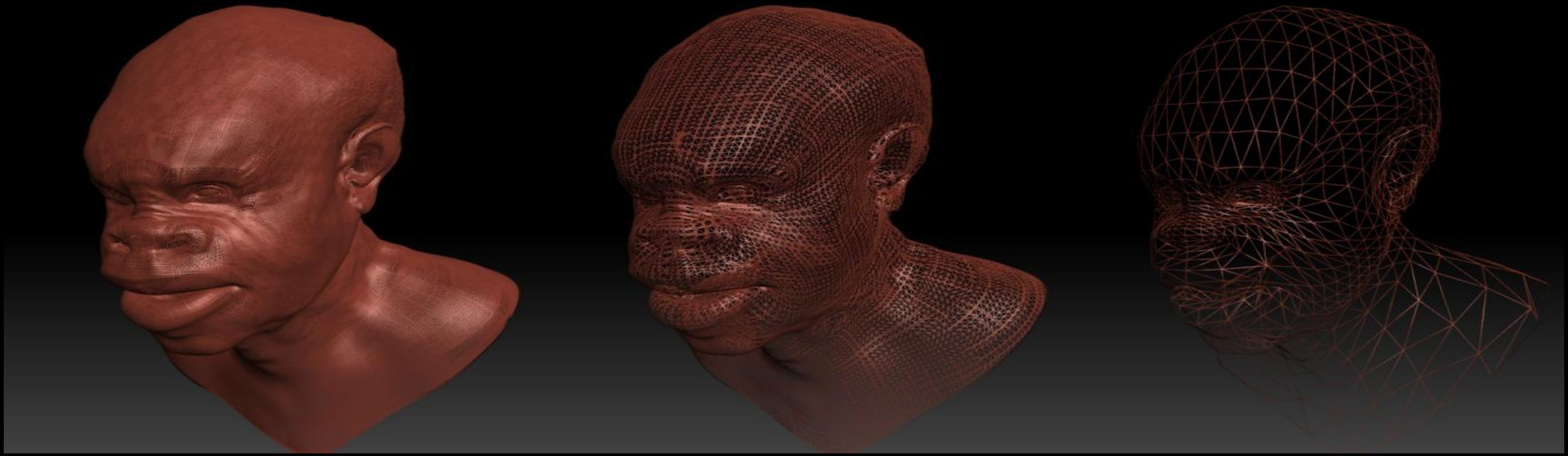
+



	Level 8	Level 16	Level 32	Level 64
Regular Triangle Mesh	16MB	59MB	236MB	943MB
D3D11 compact representation	1.9MB	7.5MB	30MB	118MB

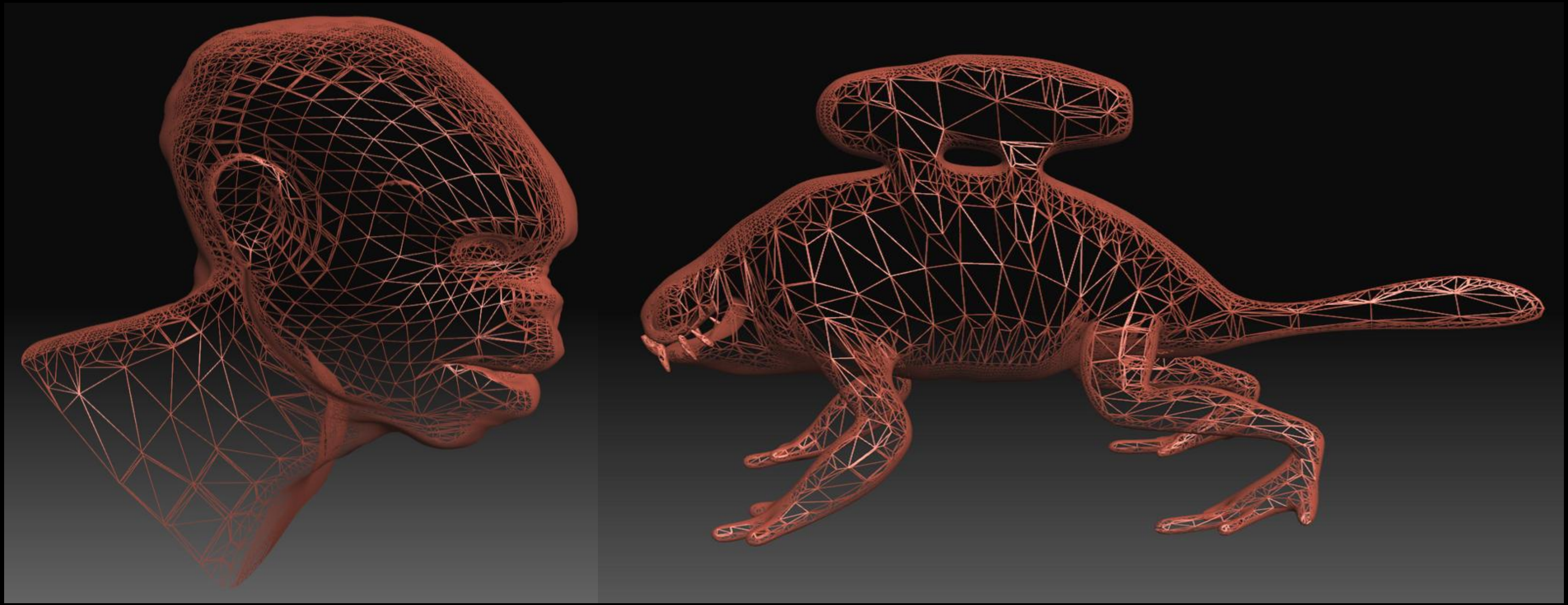
Motivation - Scalability

- Continuous Level of Detail



Motivation - Scalability

- View Dependent Level of Detail



Motivation - Animation & Simulation

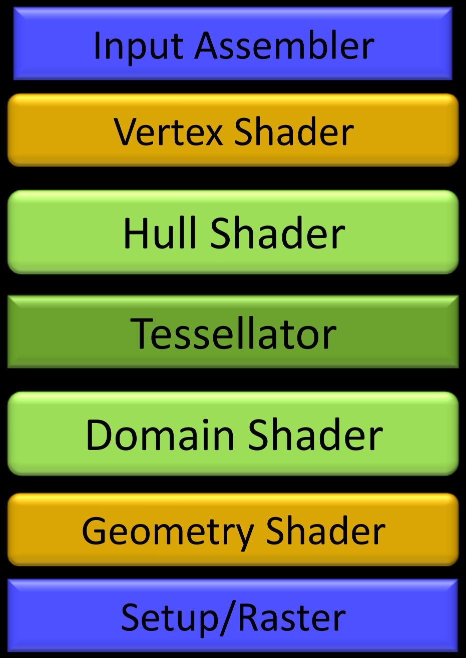
- Perform Expensive Computations at lower frequency:
  - Realistic animation: blend shapes, morph targets, etc.

  - Physics, collision detection, soft body dynamics, etc.

Tessellation Pipeline

- Direct3D11 has support for programmable tessellation

- Two new programmable shader stages:
  - Hull Shader (HS)
  - Domain Shader (DS)
- One fixed function stage:
  - Tessellator (TS)

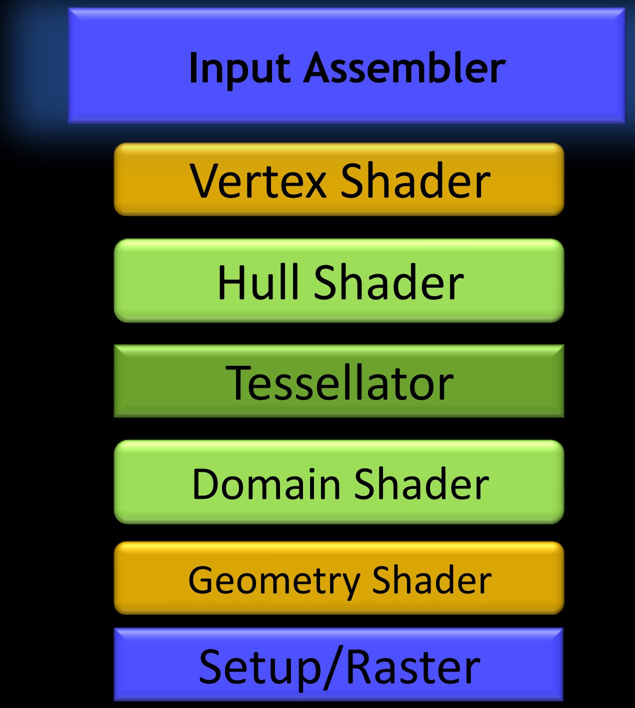


Tessellation Pipeline

- **Hull Shader** transforms basis functions from base mesh to surface patches
- **Tessellator** produces a semi-regular tessellation pattern for each patch
- **Domain Shader** evaluates surface

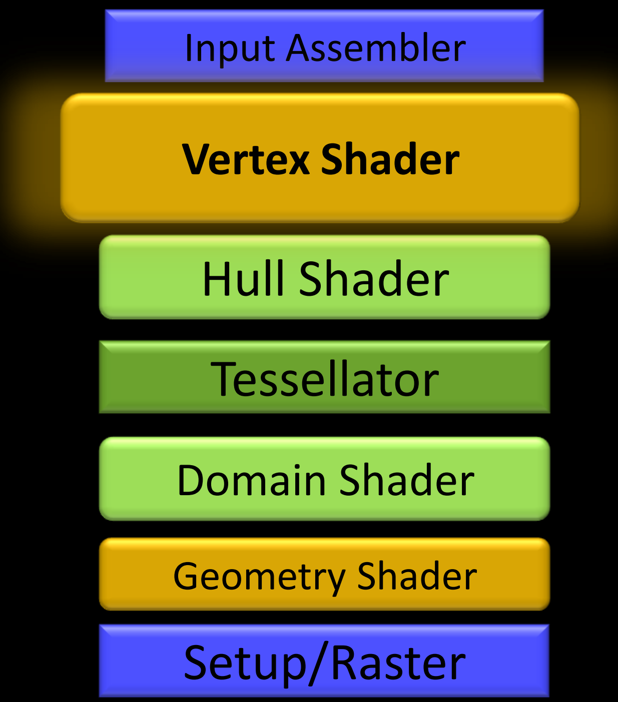
Input Assembler

- New patch primitive type
  - Arbitrary vertex count (up to 32)
  - No implied topology
  - Only supported primitive when tessellation is enabled



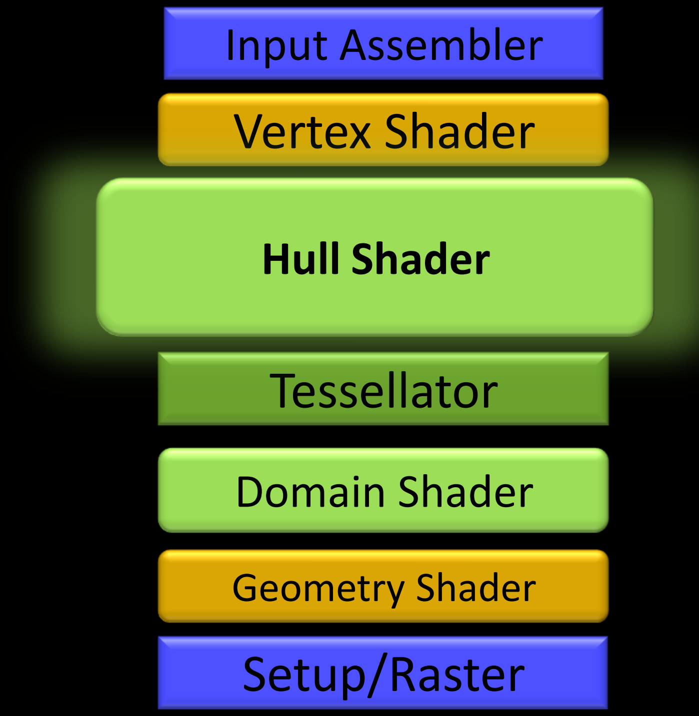
Vertex Shader

- Transforms patch control points
- Usually used for:
  - Animation (skinning, blend shapes)
  - Physics simulation
- Allows more expensive animation at a lower frequency



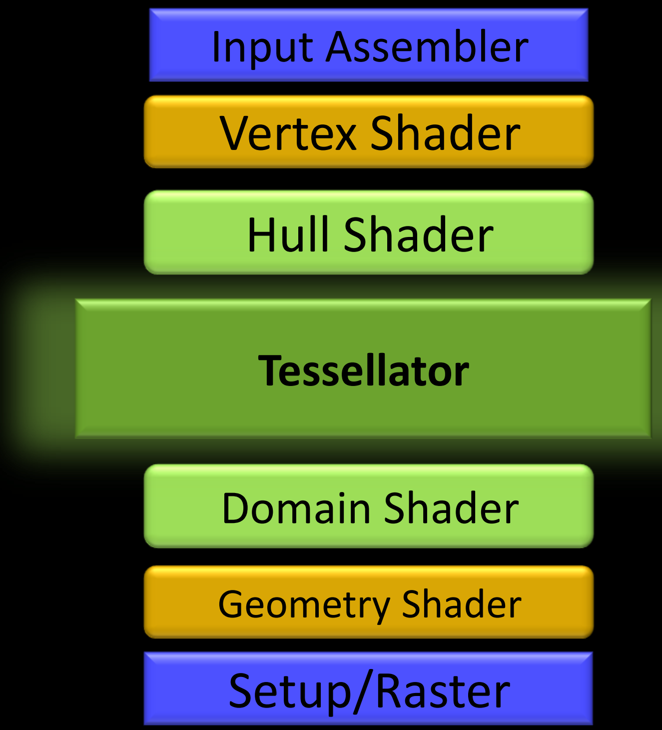
Hull Shader (HS)

- Transforms control points to a different basis
- Computes tessellation factors

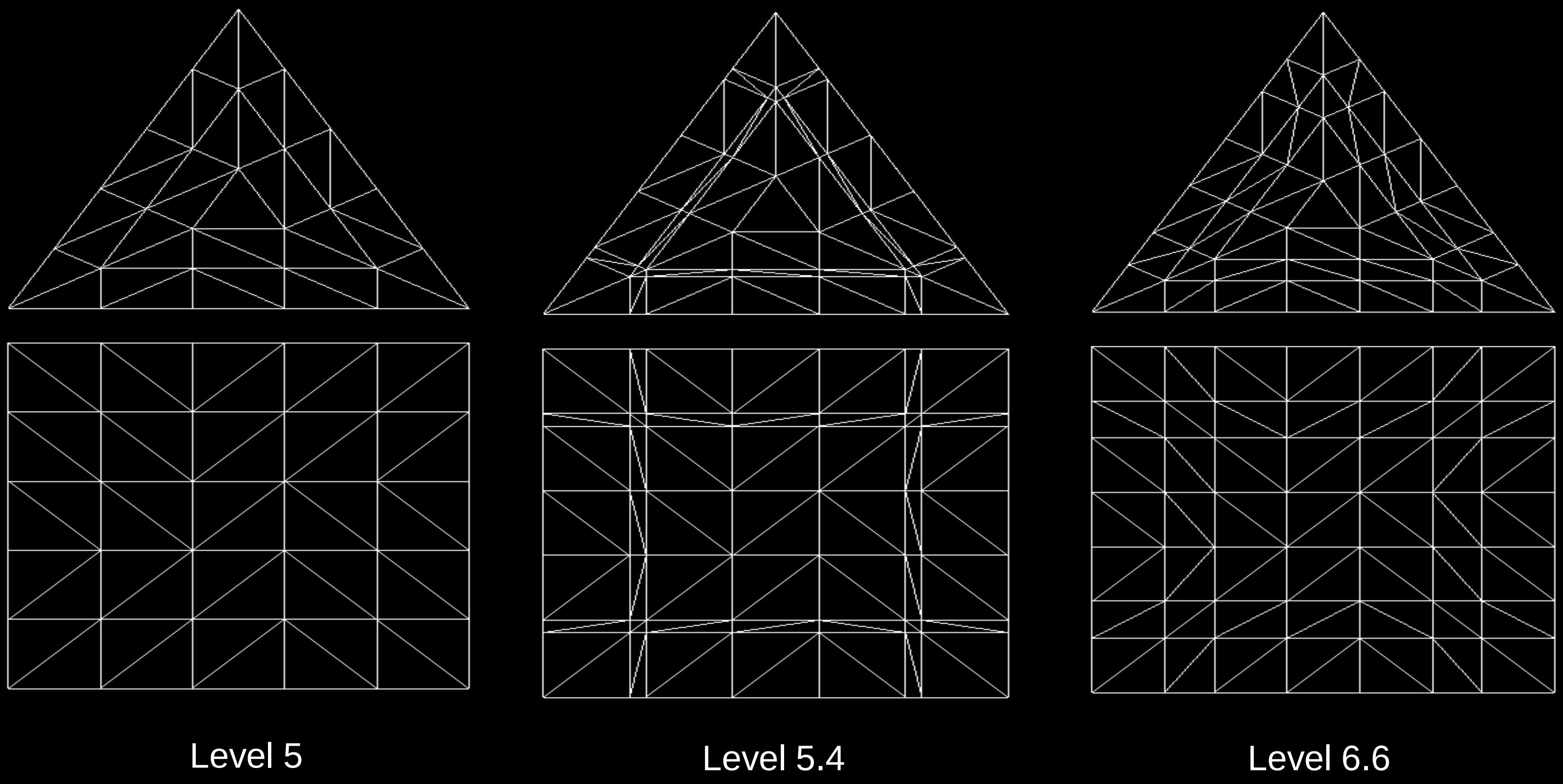


Tessellator (TS)

- Fixed function stage, but configurable
- Fully symmetric
- Domains:
  - Triangle, Quad, Isolines
- Spacing:
  - Discrete, Continuous, Pow2



Tessellator (TS)

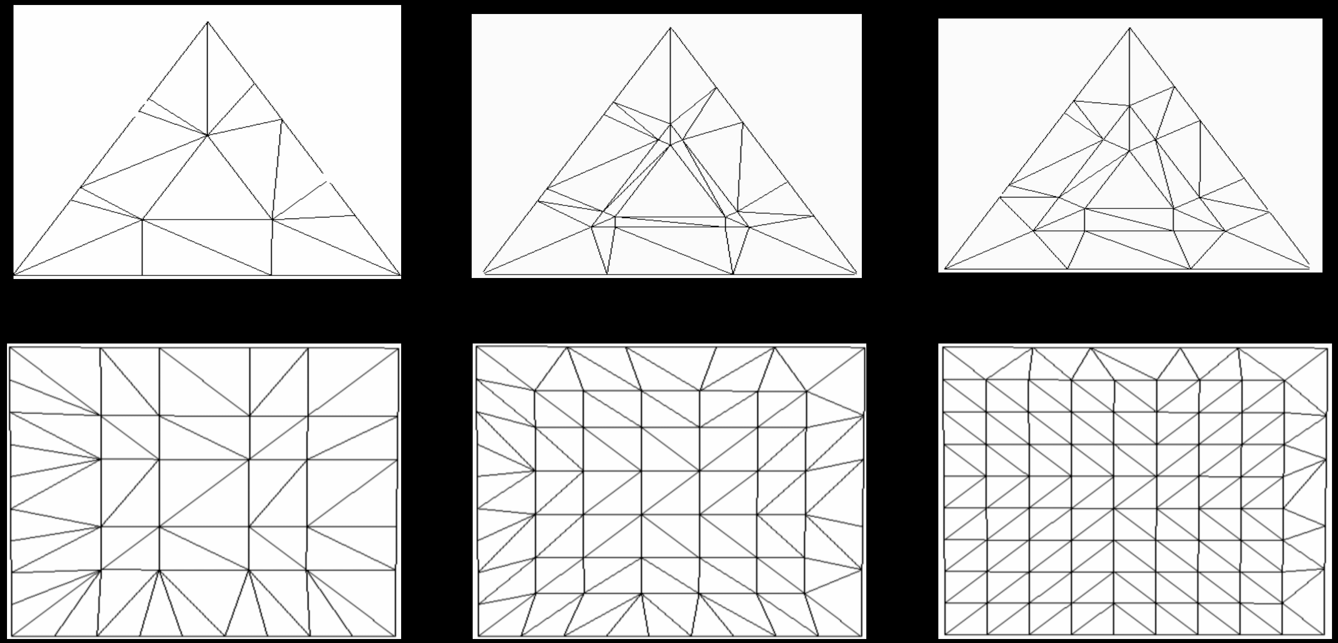


Tessellator (TS)

Left = 3.5
Right = 4.4
Bottom = 3.0

Top,Right = 4.5

Bottom,Left = 9.0



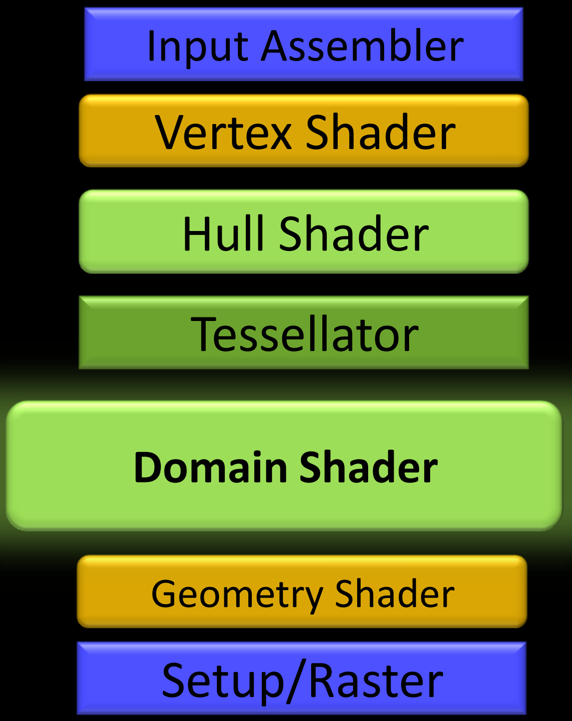
Inside Tess: minimum

Inside Tess: average

Inside Tess: maximum

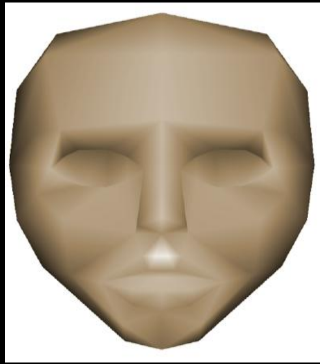
Domain Shader (DS)

- Evaluate surface given parametric UV coordinates
- Interpolate attributes
- Apply displacements



Example - PN Triangles

- Simple tessellation scheme
  - Provides smoother silhouettes and better shading
- Operates directly on triangle meshes with per vertex Positions and Normals
  - Easily integrated into existing rendering pipelines



Input Triangles

Output
Curved PN triangles

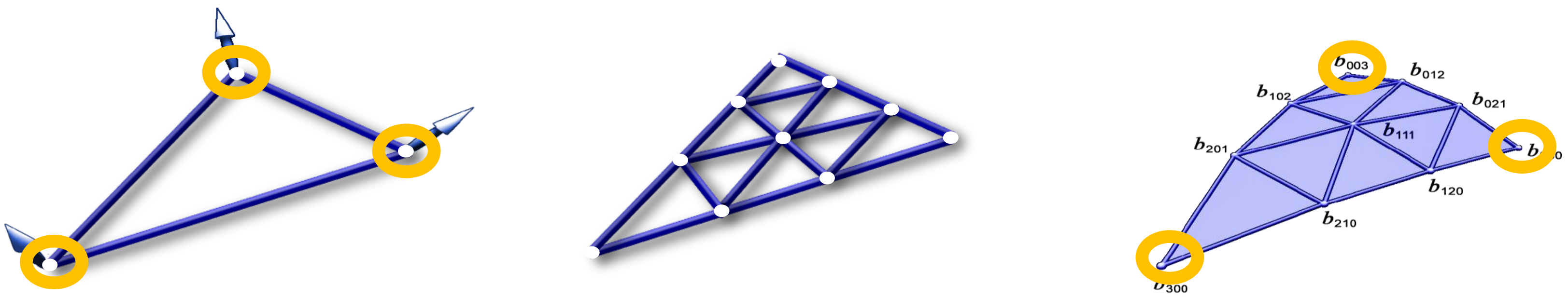
Vlachos et al, http://ati.amd.com/developer/curvedpntriangles.pdf

PN Triangles - Positions

- 1- Replace input triangle with a bezier patch
  - Use Hull Shader
- 2- Triangulated bezier patch into a specified number of sub triangles
  - Use Tessellator and Domain Shader
  - Number of Sub triangles specified by Hull Shader

PN Triangles - Position Control Points

Computing Position Control Points



Exterior control point positions:

same as input vertex positions

$$b_{300} = P_1$$
$$b_{030} = P_2$$
$$b_{003} = P_3$$

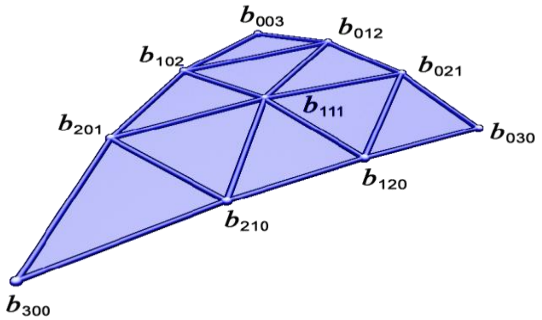
Interior control point positions:

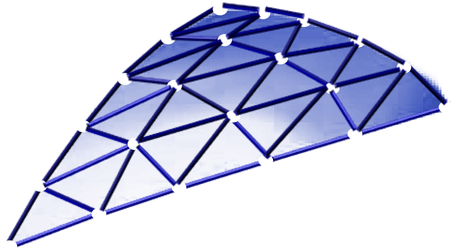
Weighted combinations of input positions and normals

$$w_{ij} = \left(P_j - P_i\right) \bullet N_i$$
$$b_{210} = \left(2P_1 + P_2 - w_{12}N_1\right) / 3$$
$$b_{120} = \left(2P_2 + P_1 - w_{21}N_2\right) / 3$$

PN Triangles - Final Positions

Evaluating tessellated positions from control points

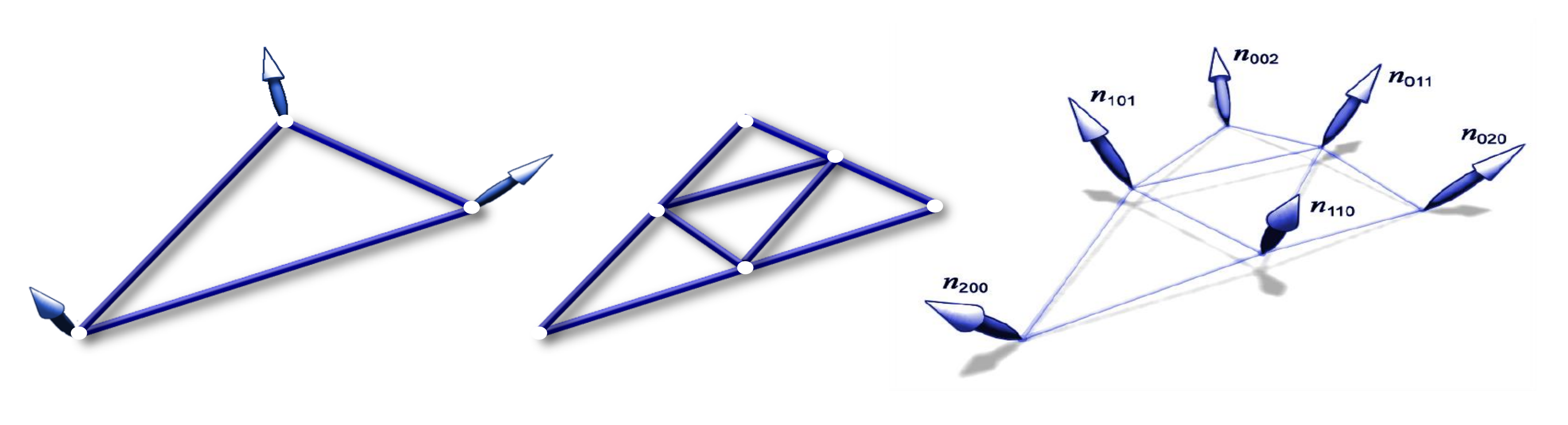




$$w = 1 - u - v \qquad u, v, w \geq 0$$

$$\begin{aligned} b(u,v) \quad = \quad & b_{300}w^3 && + b_{030}u^3 && + b_{003}v^3 \\ & + b_{210}3w^2u && + b_{120}3wu^2 && + b_{201}3w^2v \\ & + b_{021}3u^2v && + b_{102}3wv^2 && + b_{012}3uv^2 \\ & + b_{111}6wuv \end{aligned}$$

PN Triangles - Normals

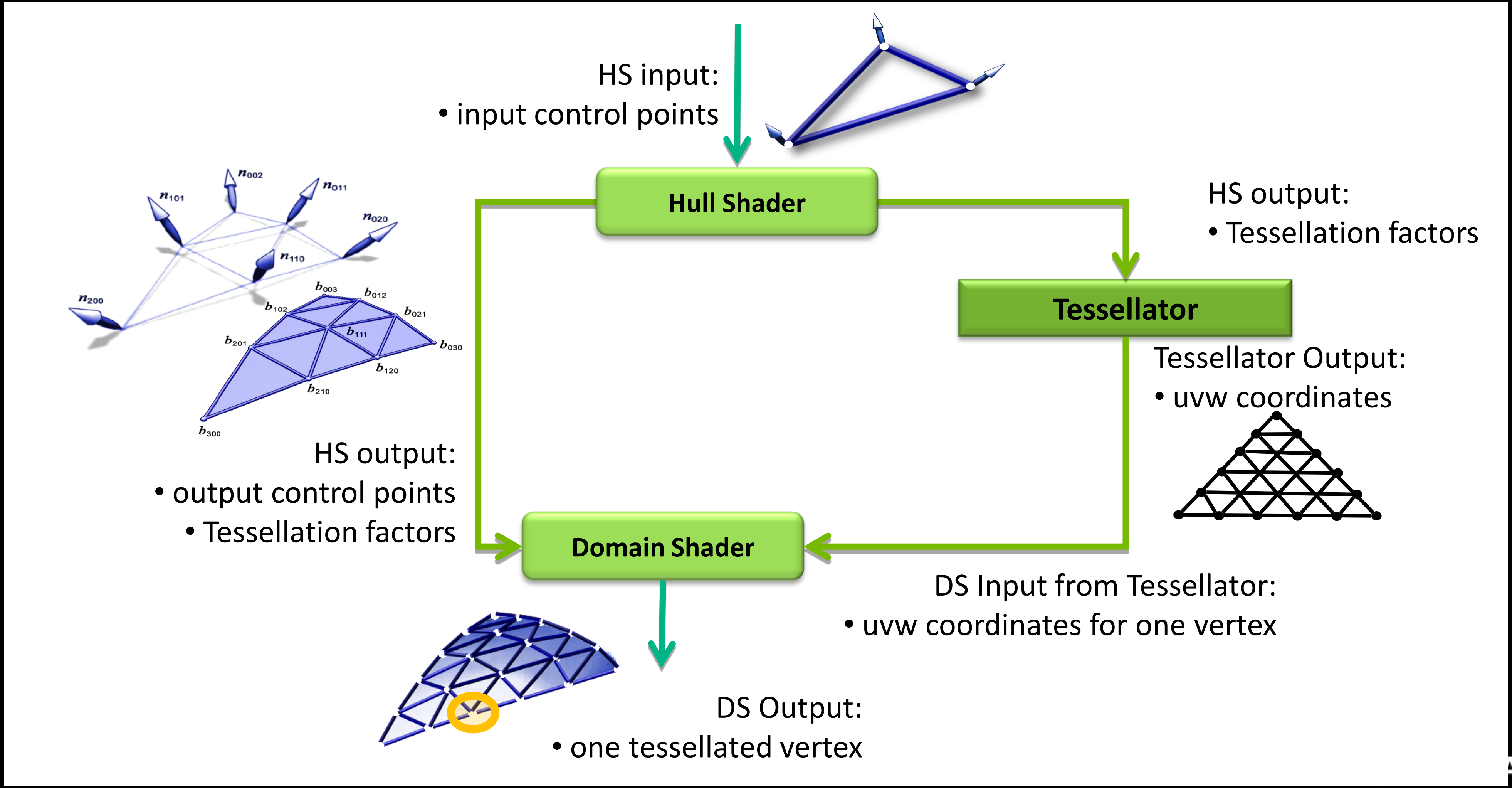


- Normal at a tessellated vertex is a quadratic function of position and normal data

$$w = 1 - u - v$$

$$n(u,v) = n_{200}w^2 + n_{020}u^2 + n_{002}v^2 + n_{110}wu + n_{011}uv + n_{101}wv$$

Tessellation Pipeline



Hull Shader Stages

- Main Hull Shader
    - Calculate control point data
    - Invoked once per output control point
- Patch Constant Function
    - Must calculate tessellation factors
    - Has access to control point data calculated in the Main Hull Shader
    - Executes once per patch

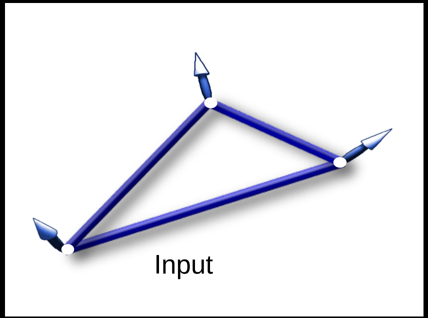
PN Triangles - Hull Shader

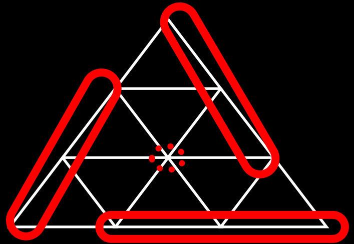
- Compute control point positions and normals in main Hull Shader

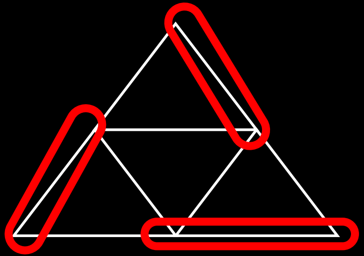
- Compute tessellation factors and center location in patch constant function
  - The center location needs to average all the other control point locations so it belongs in the patch constant function

PN Triangles - Hull Shader

- Partitioning the computation
- To balance the workload across threads we partition the control points into 3 uber control points
- Each uber control point computes
  – 3 positions
  – 2 normals







PN Triangles - Hull Shader

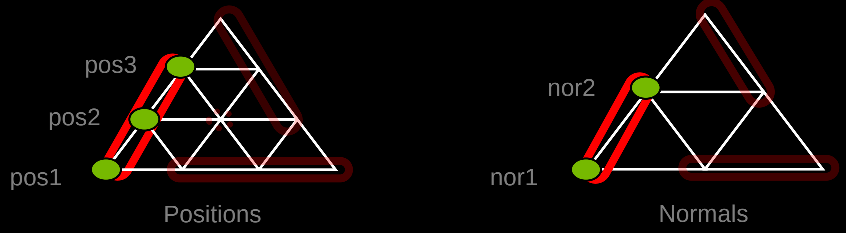
```
struct HS_PATCH_DATA
{
      float edges[3]   : SV_TessFactor;
      float inside     : SV_InsideTessFactor;
      float center[3]  : CENTER;
};
```

Data output by the patch constant function

```
struct HS_CONTROL_POINT
{
      float pos1[3]  : POSITION1;
      float pos2[3]  : POSITION2;
      float pos3[3]  : POSITION3;
      float3 nor1    : NORMAL0;
      float3 nor2    : NORMAL1;
      float3 tex     : TEXCOORD0;
};
```

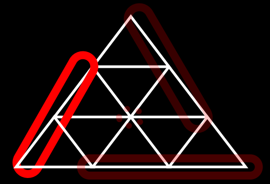
Data output by main tessellation function

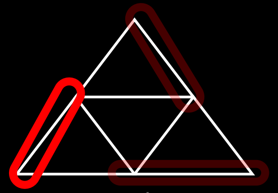
Control point 1



GPU TECHNOLOGY CONFERENCE

PN Triangles - Hull Shader


Positions


Normals

Control point 1

```
[domain("tri")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[partitioning("fractional_odd")]
[patchconstantfunc("HullShaderPatchConstant")]
HS_CONTROL_POINT HullShaderControlPointPhase( InputPatch<HS_DATA_INPUT, 3> inputPatch,
                              uint tid : SV_OutputControlPointID, uint pid : SV_PrimitiveID)
{
      int next = (1 << tid) & 3;   // (tid + 1) % 3

      float3 p1 = inputPatch[tid].position;
      float3 p2 = inputPatch[next].position;
      float3 n1 = inputPatch[tid].normal;
      float3 n2 = inputPatch[next].normal;

      HS_CONTROL_POINT output;

      //control points positions
      output.pos1 = (float[3])p1;
      output.pos2 = (float[3])(2 * p1 + p2 - dot(p2-p1, n1) * n1);
      output.pos3 = (float[3])(2 * p2 + p1 - dot(p1-p2, n2) * n2);

      //control points normals
      float3 v12 = 4 * dot(p2-p1, n1+n2) / dot(p2-p1, p2-p1);
      output.nor1 = n1;
      output.nor2 = n1 + n2 - v12 * (p2 - p1);

      output.tex = inputPatch[tid].texcoord;
```

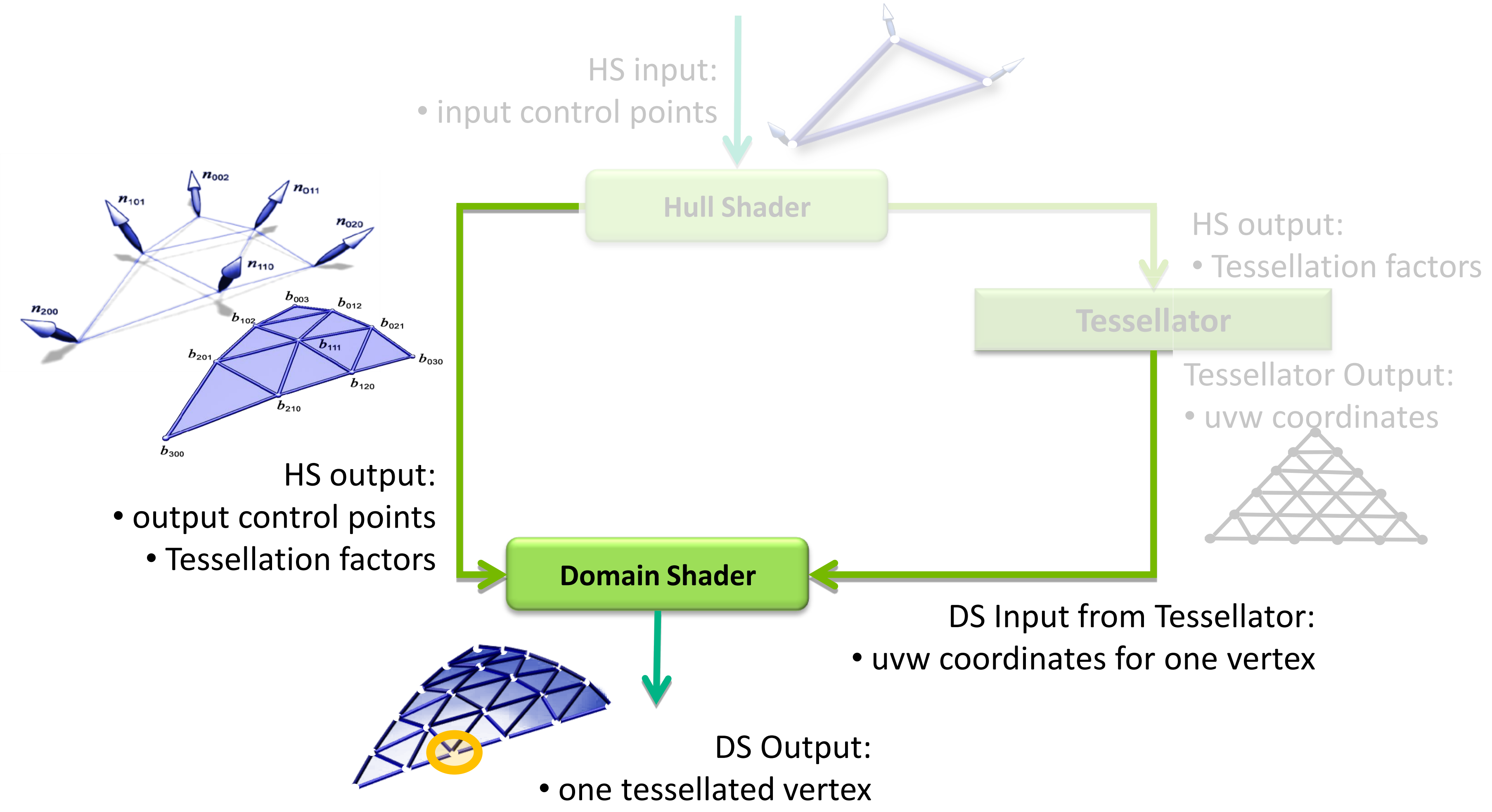
Read input data

Compute control points

PN Triangles - Hull Shader

```
//patch constant data
HS_PATCH_DATA HullShaderPatchConstant( OutputPatch<HS_CONTROL_POINT, 3> controlPoints )
{
    HS_PATCH_DATA patch = (HS_PATCH_DATA)0;
    //calculate Tessellation factors
    HullShaderCalcTessFactor(patch, controlPoints, 0);
    HullShaderCalcTessFactor(patch, controlPoints, 1);
    HullShaderCalcTessFactor(patch, controlPoints, 2);
    patch.inside = max(max(patch.edges[0], patch.edges[1]), patch.edges[2]);

    //calculate center
    float3 center = ((float3)controlPoints[0].pos2 + (float3)controlPoints[0].pos3) * 0.5 -
                     (float3)controlPoints[0].pos1 +
                    ((float3)controlPoints[1].pos2 + (float3)controlPoints[1].pos3) * 0.5 –
                     (float3)controlPoints[1].pos1 +
                    ((float3)controlPoints[2].pos2 + (float3)controlPoints[2].pos3) * 0.5 –
                     (float3)controlPoints[2].pos1;

    patch.center = (float[3])center;
    return patch;
}

//helper functions
float edgeLod(float3 pos1, float3 pos2) { return dot(pos1, pos2); }
void HullShaderCalcTessFactor( inout HS_PATCH_DATA patch,
                 OutputPatch<HS_CONTROL_POINT, 3> controlPoints, uint tid : SV_InstanceID)
{
    int next = (1 << tid) & 3;   // (tid + 1) % 3
    patch.edges[tid] = edgeLod((float3)controlPoints[tid].pos1,
                       (float3)controlPoints[next].pos1);
    return;
}
```

Tessellation Pipeline



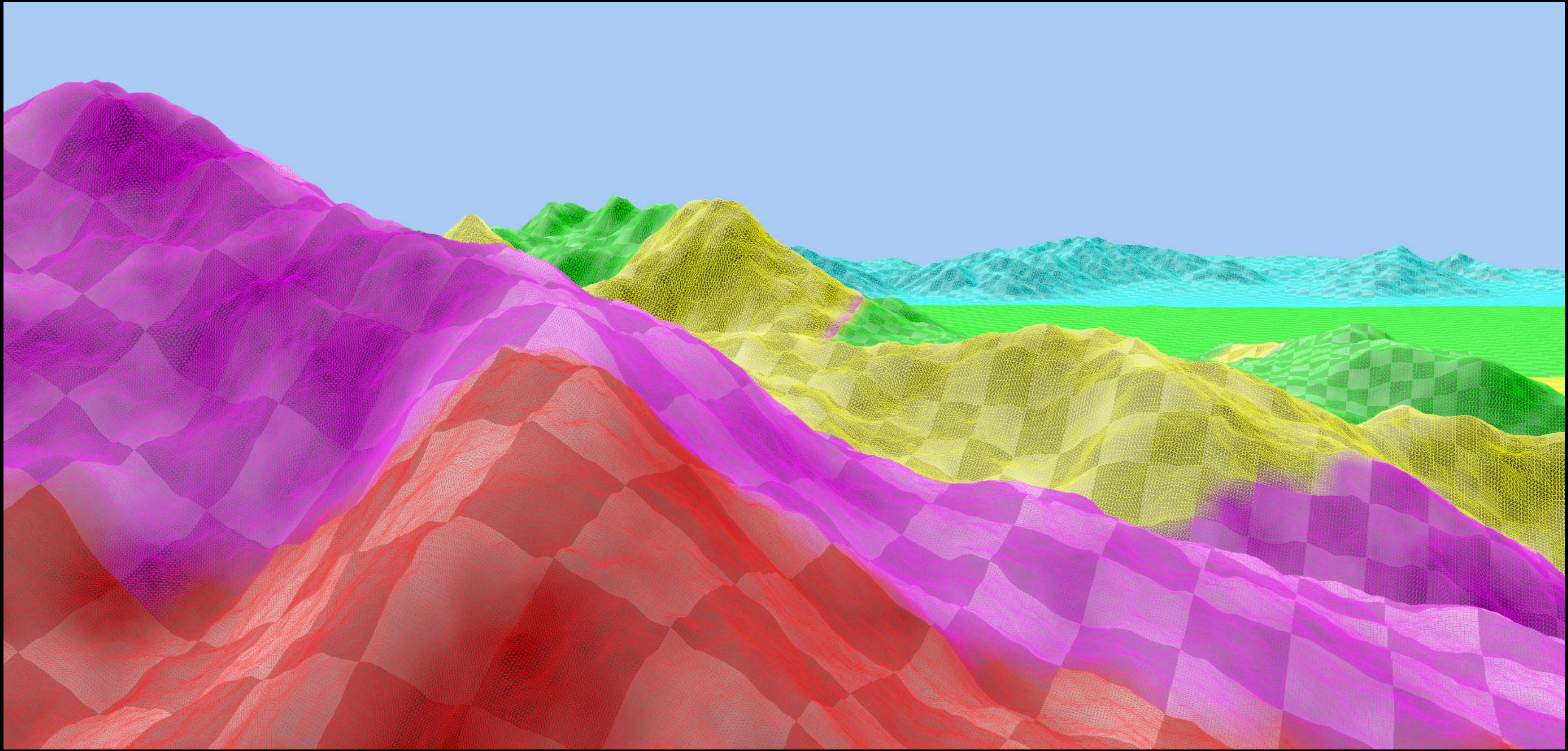
PN-Triangles - Domain Shader

```
DS_DATA_OUTPUT DomainShaderPN(HS_PATCH_DATA patchData,
             const OutputPatch<HS_CONTROL_POINT, 3> input, float3 uvw : SV_DomainLocation)
{
    DS_DATA_OUTPUT output;
    float u = uvw.x;
    float v = uvw.y;
    float w = uvw.z;

    //output position is weighted combination of all 10 position control points
    float3 pos = (float3)input[0].pos1 * w*w*w +(float3)input[1].pos1 * u*u*u +(float3)input[2].pos1 * v*v*v +
                 (float3)input[0].pos2 * w*w*u +(float3)input[0].pos3 * w*u*u +(float3)input[1].pos2 * u*u*v +
                 (float3)input[1].pos3 * u*v*v +(float3)input[2].pos2 * v*v*w +(float3)input[2].pos3 * v*w*w +
                 (float3)patchData.center * u*v*w;

    //output normal is weighted combination of all 6 normal control points
    float3 nor = input[0].nor1 * w*w + input[1].nor1 * u*u + input[2].nor1 * v*v +
                 input[0].nor2 * w*u + input[1].nor2 * u*v + input[2].nor2 * v*w;

    //transform and output data
    output.position  = mul(float4(pos,1), g_mViewProjection);
    output.view = mul(float4(pos,1),g_mView).xyz;
    output.normal = mul(float4(normalize(nor),1),g_mNormal).xyz;
    output.vUV = input[0].tex * w + input[1].tex * u + input[2].tex * v;
```

GPU TECHNOLOGY CONFERENCE

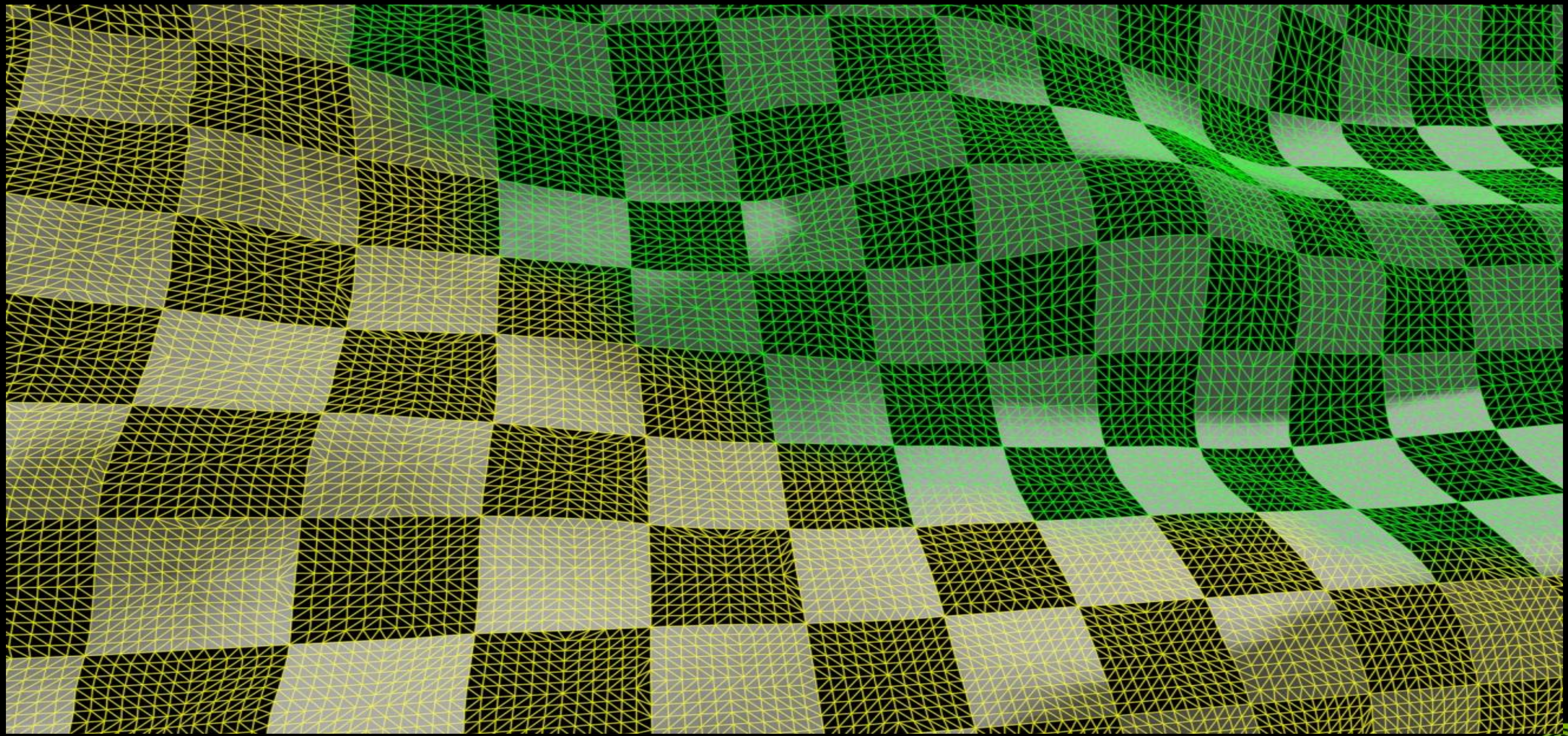
PRESENTED BY NVIDIA.

Terrain Tessellation



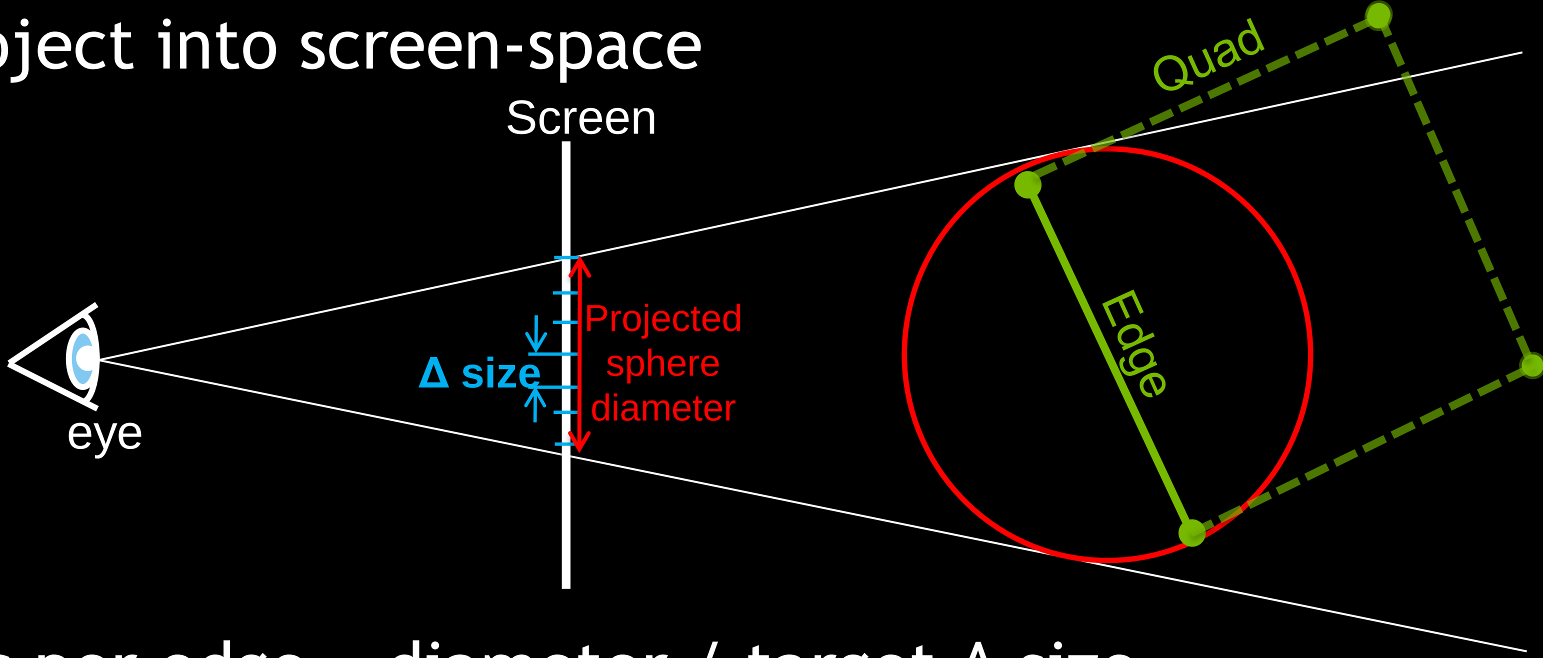
Terrain Tessellation Basics

- Flat quads; regular grid; can be instanced
- Height map; vertical displacement; sample in DS



Screen-space-based LOD (Hull shader)

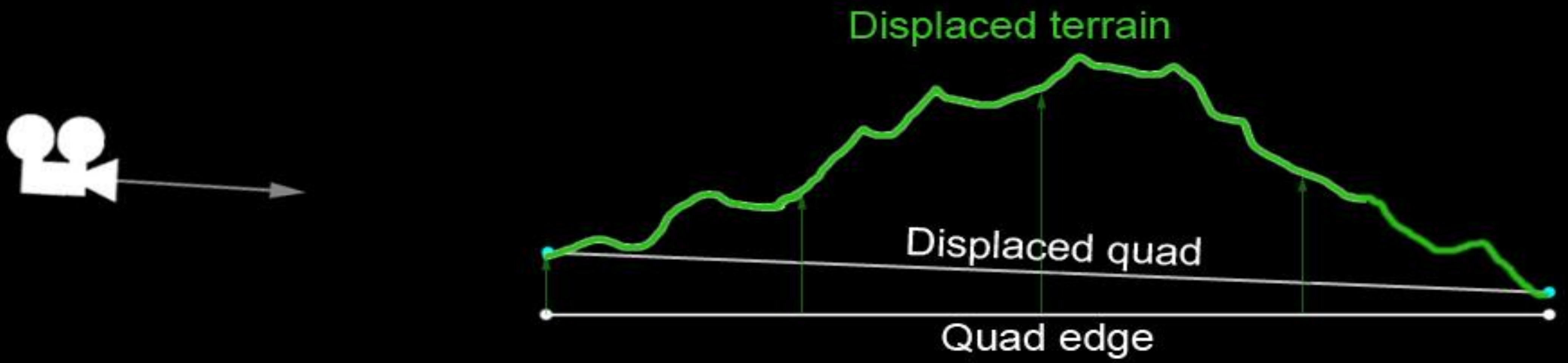
- Enclose quad patch edge in bounding sphere
- Project into screen-space



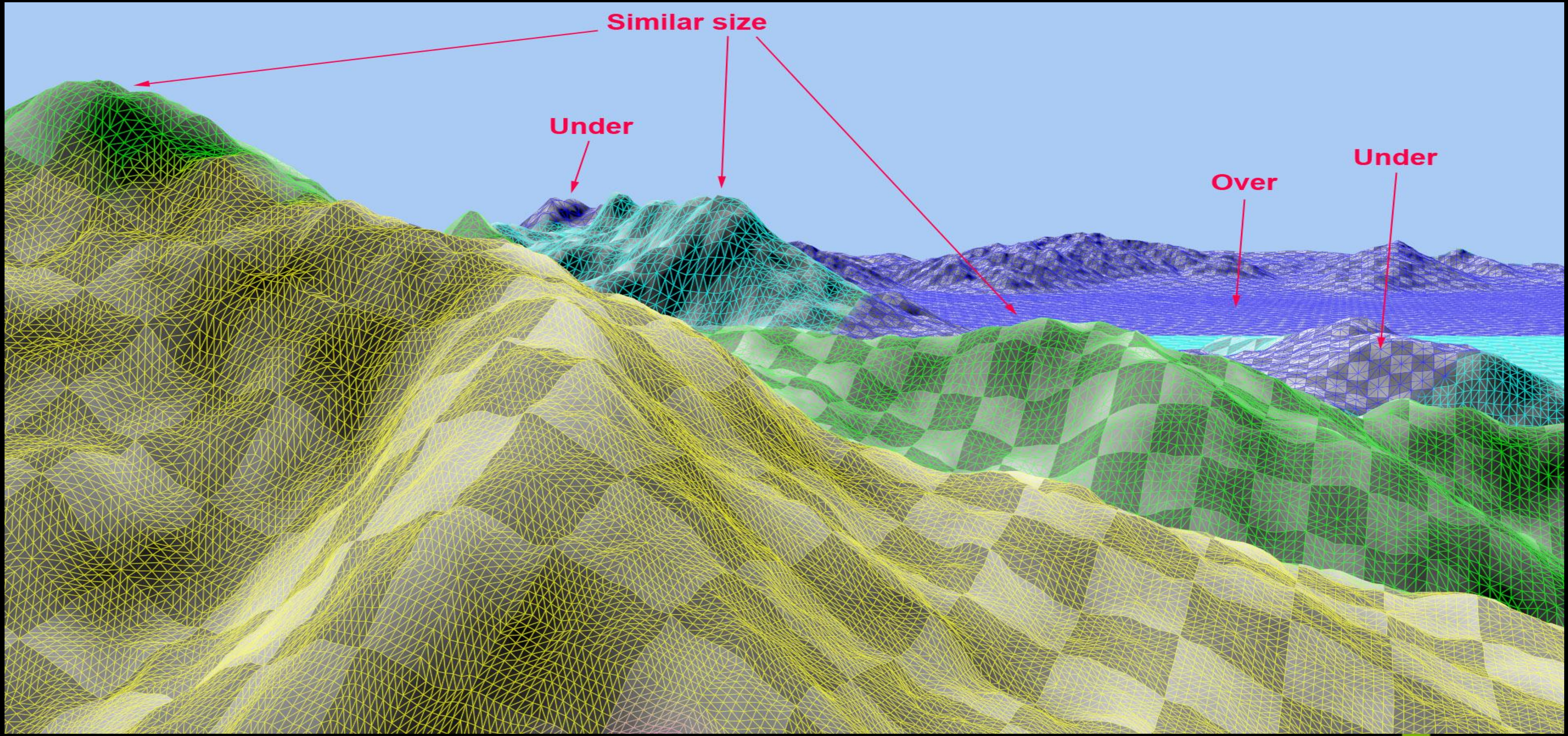
- Δs per edge = diameter / target Δ size
- (diameter & target size in pixels)
- Fully independent of patch size

Screen-space-based LOD

- Why quad-edge bounding sphere?
- Projected edges seen edge-on:
  - → zero width in screen-space
  - → min tessellation & bad aliasing
- Spheres = orientation independent



Screen-space-based LOD Results

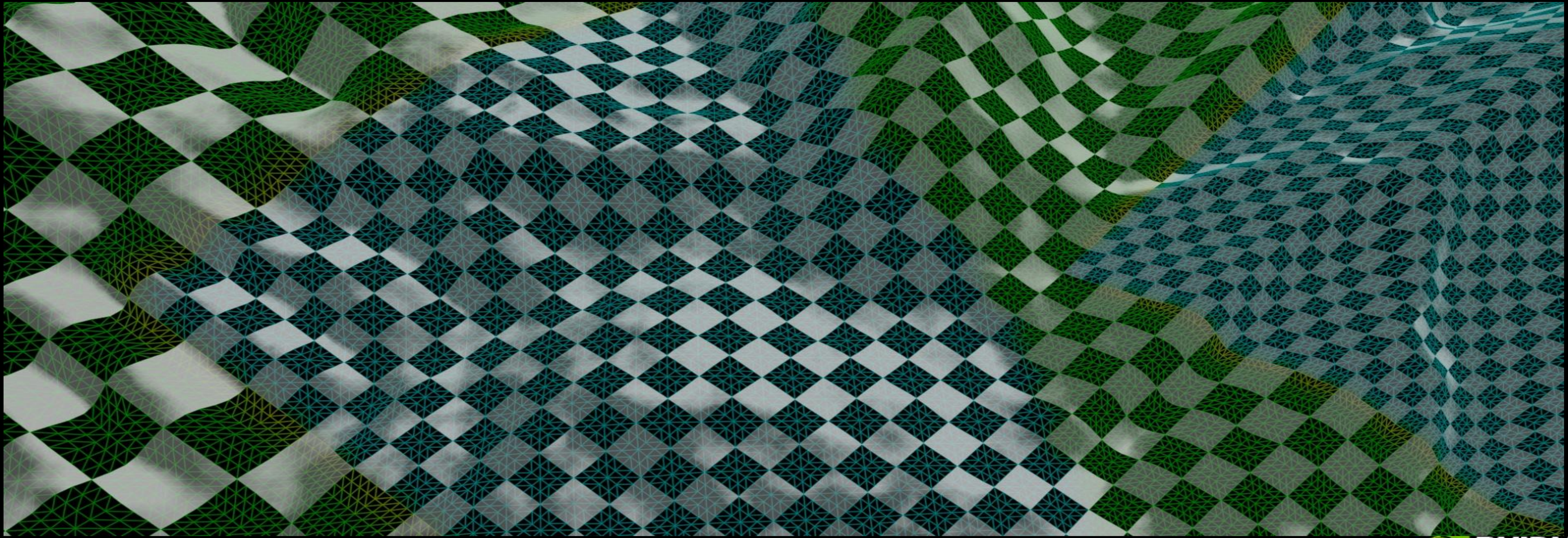


Crack-free Tessellation

- Match edge data between adjacent patches
- Match HS LOD calculations
- Easy to break accidentally
  - Cracks are small & subtle
  - Check very carefully
- Debug camera, independent matrices for:
  - Projection
  - LOD

Non-uniform Patches

- Max tessellation = 64 → limited range of LODs
- Patches of different sizes required
- Recall: screen-space LOD *independent* of patch size



Crack-free Non-uniform Patches

- Gets tricky
- Encode adjacent neighbours' sizes in VB
- In HS: detect different size neighbours
- Match their LOD calculations
- Result: long HS = 460 hs_5_0 instructions

Data Problems

- Large world, say 60x60km
- Fine tessellation, say 2m Δs
- Naïve height map is 100s Mb to Gb

- Migrate existing engine to DX11
- DX9/10: coarse data relative to tessellation capabilities

Data Solution: Fractal “Amplification”

- Coarse height map defines topographic shape
- Fractal detail map adds high-LOD detail
- Cheap memory requirements
- Can reuse coarse assets from DX9 or DX10 engine
- **Old diagram from**
  - “Computer Rendering of Stochastic Models”, Fournier Fussell & Carpenter, 1982

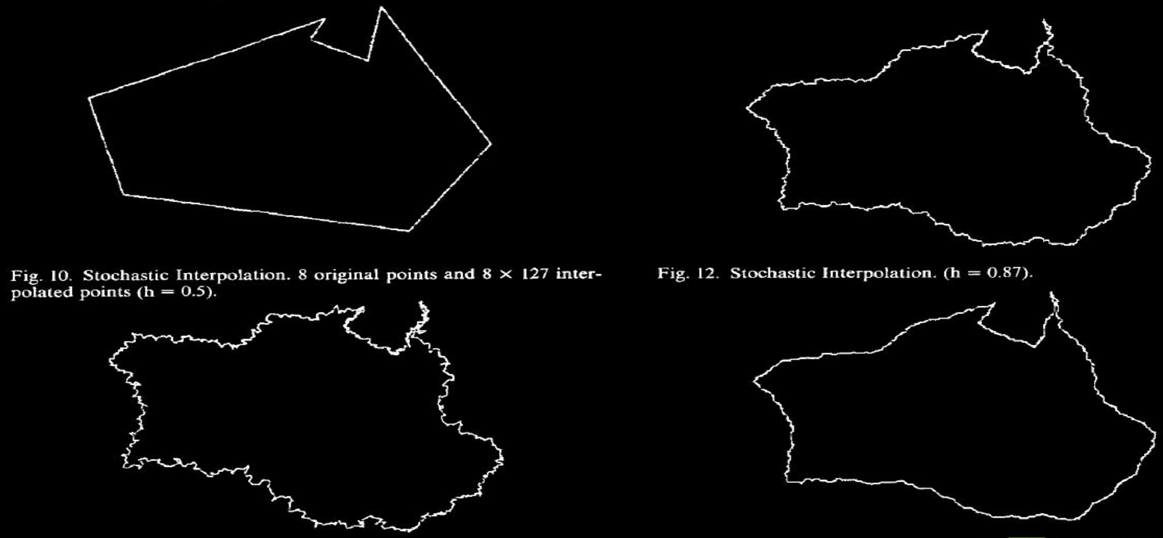


Fig. 9. Australia: 8 Sample Points.

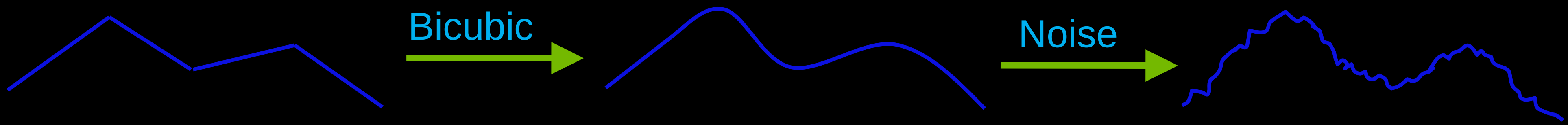
Fig. 11. Stochastic Interpolation. (h = 0.7).

Fig. 10. Stochastic Interpolation. 8 original points and 8 × 127 interpolated points (h = 0.5).

Fig. 12. Stochastic Interpolation. (h = 0.87).

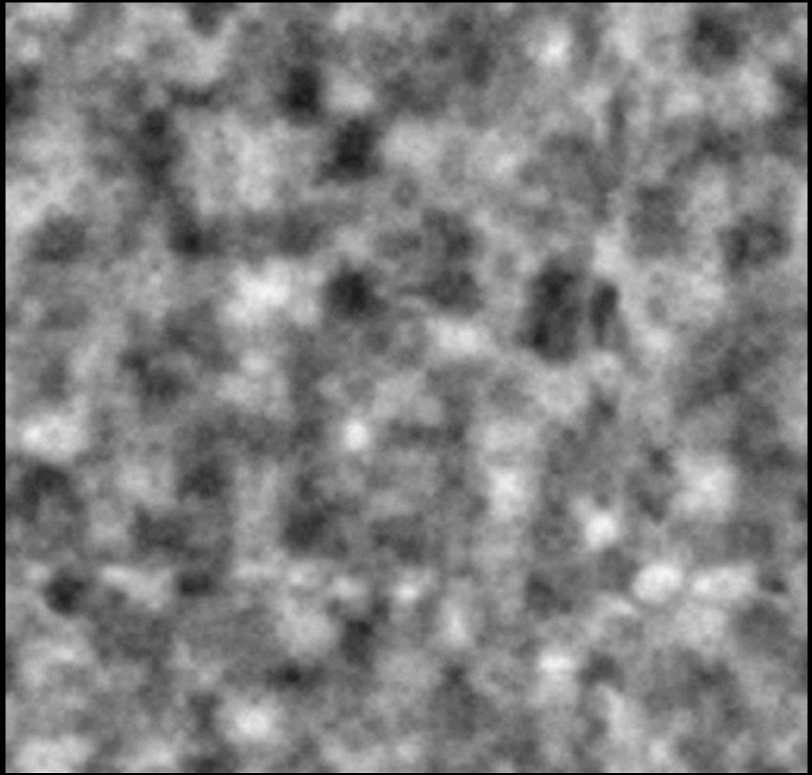
Data Solution: Fractal “Amplification”

- Coarse height map defines topographic shape
- Upsample
- High-quality filter to smooth
  - We used bicubic

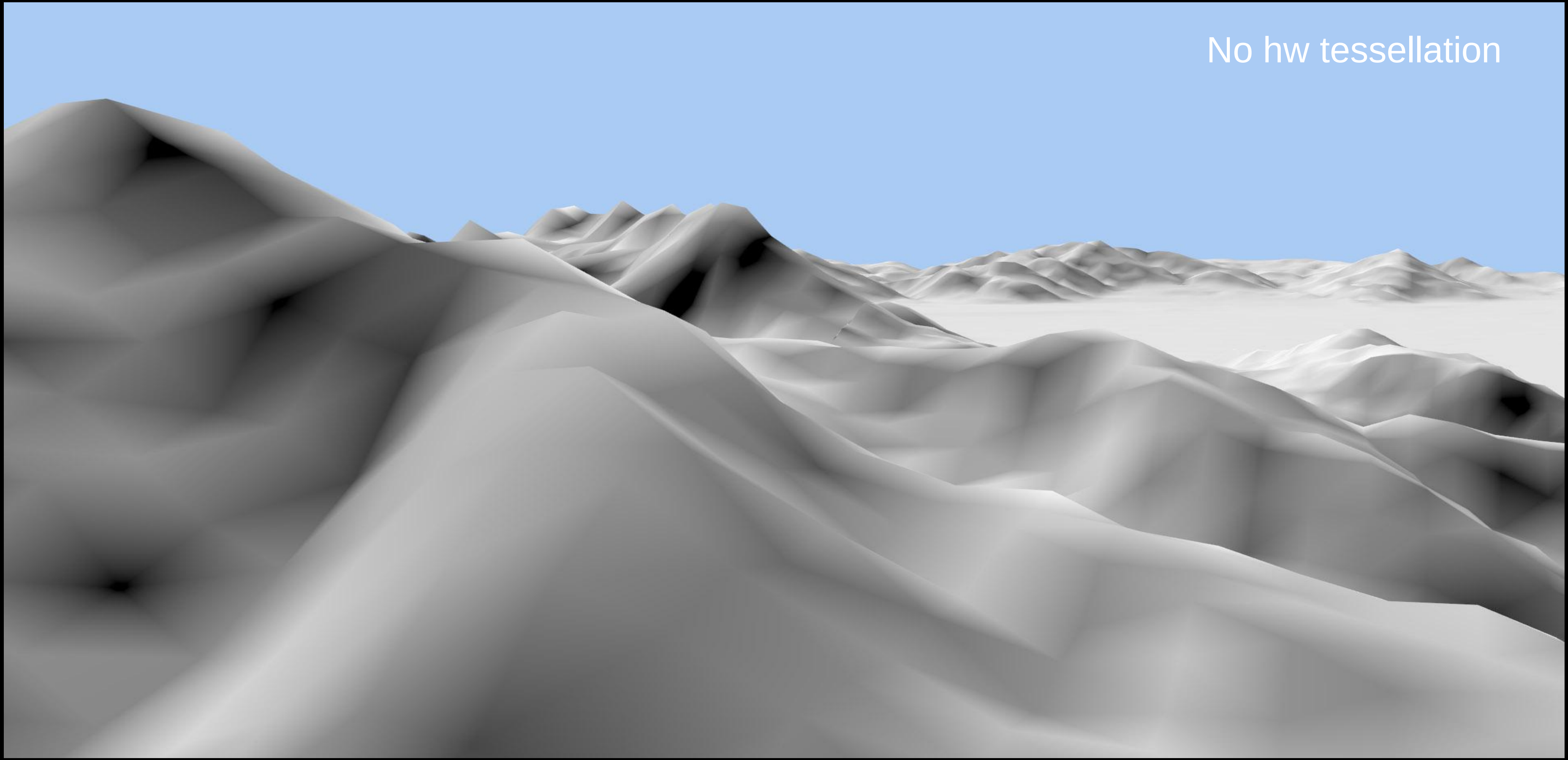


Data Solution: Fractal "Amplification"

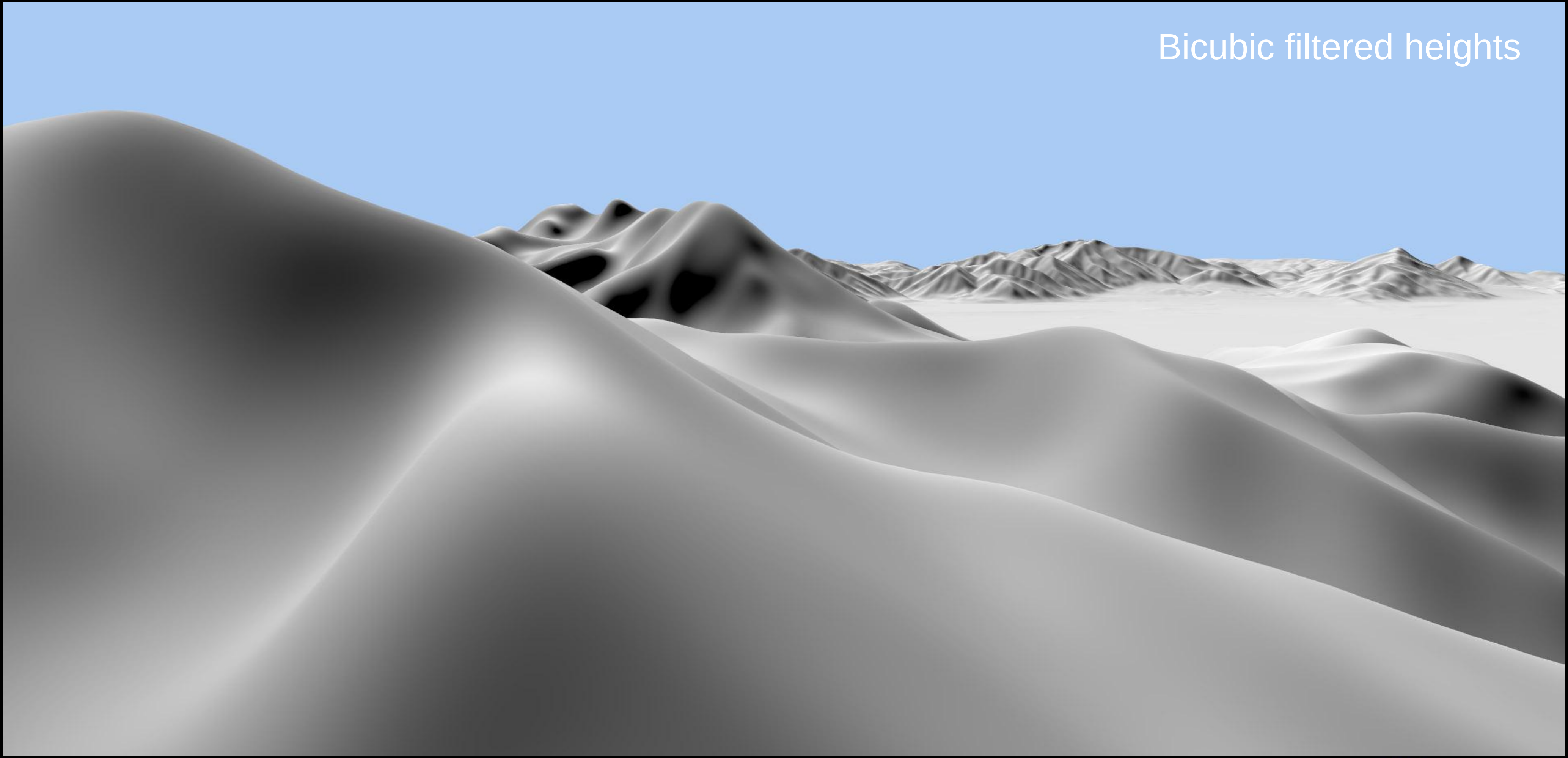
- Add detail height map:
  - fBm noise – fractally self-similar to coarse data
  - Must tile
  - Scale amplitude intelligently – doesn't work everywhere
    - Fn of height (like Musgrave's multi-fractals)
    - As a fn of coarse data roughness (reuse existing normal map)
    - Explicit mask (e.g., under buildings)



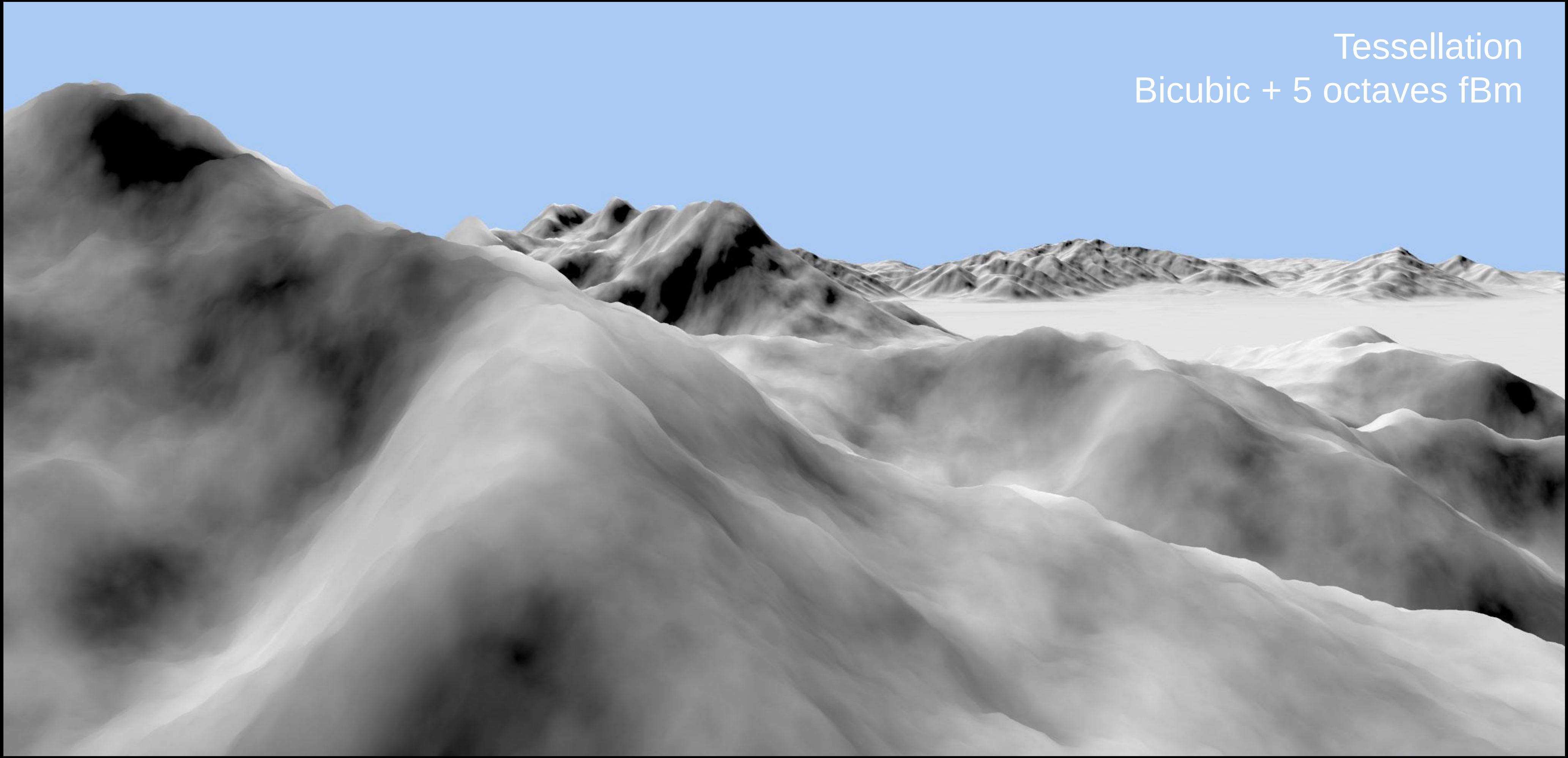
Fractal "Amplification" - Results



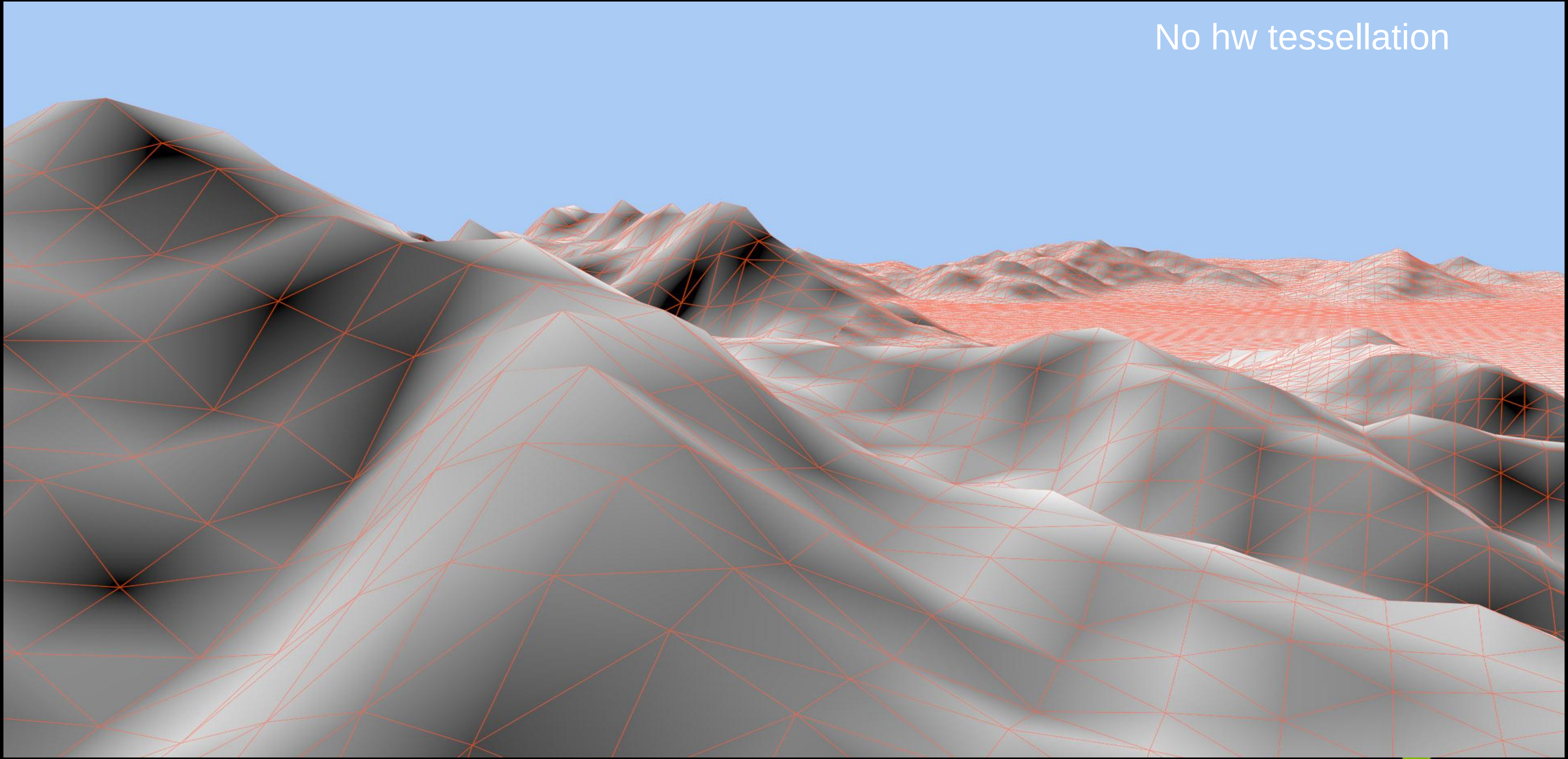
Fractal "Amplification" - Results



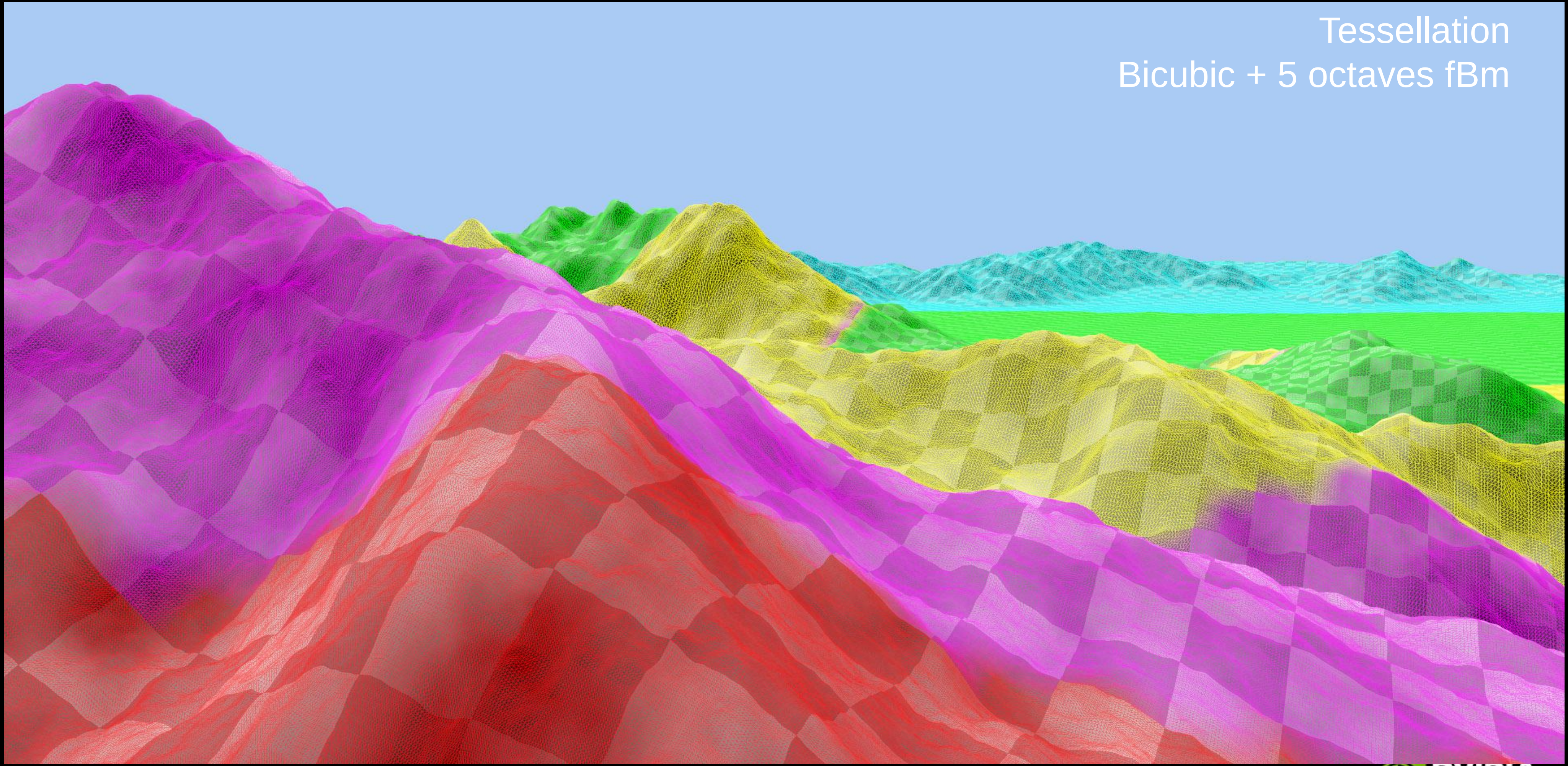
Fractal “Amplification” - Results



Fractal “Amplification” - Results



Fractal “Amplification” - Results

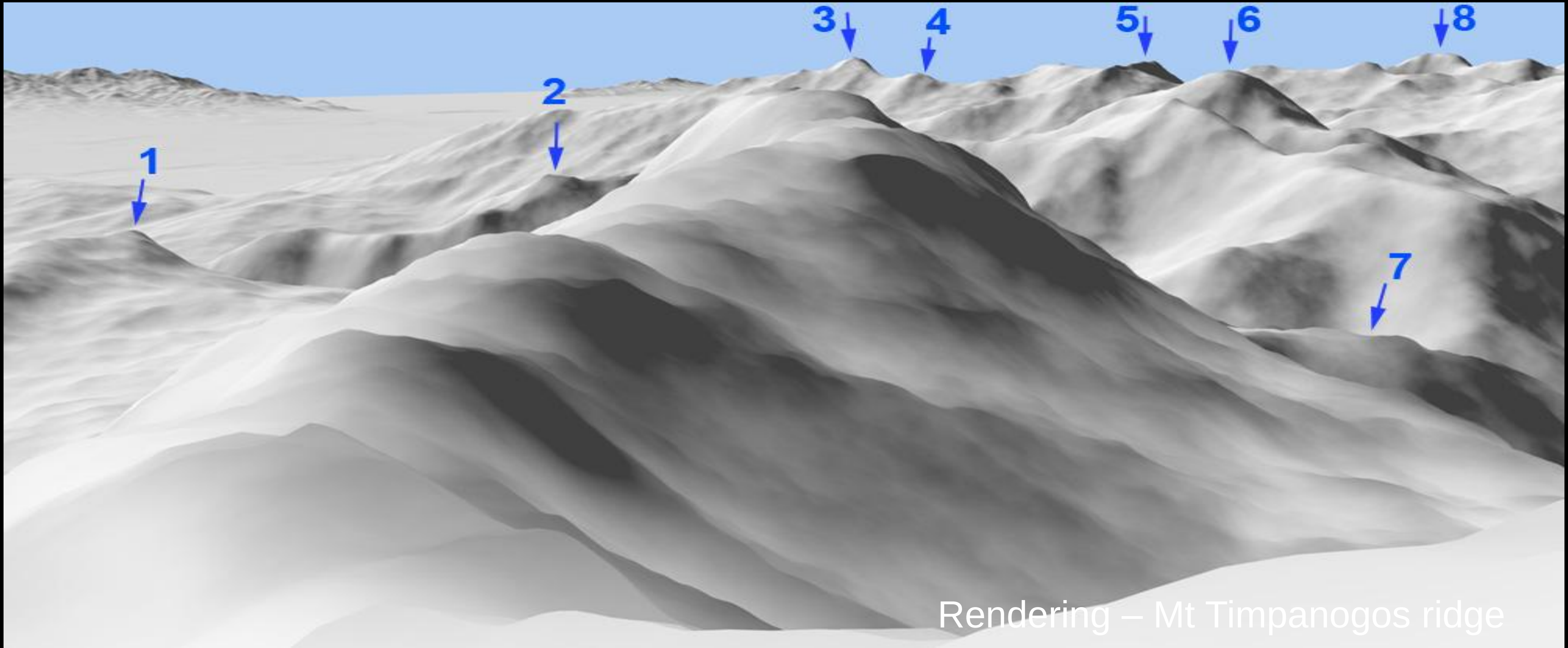


Fractal "Amplification" - Limits

- Real terrain not always fractally self-similar
- Best when coarse data is like fBm
- Erosion features – rivers, gorges, rivulets – difficult/impossible in tiling detail map
- fBm lumps not good model, especially at ~1m scale, e.g. rocks & scree
- Best at mid- and low-LOD
- Acceptable at very fine LOD

Fractal “Amplification” - Limits

Fractal "Amplification" - Limits



Tessellation Shading

- Tessellation can be used for other novel effects
- You can do shading in the DS!
  - Can be used to selectively evaluate low freq functions
  - Examples: caustics, fourier opacity maps

Outline: Multithreading

- Why DirectX 11?
- Direct Compute
- Tessellation
- **Multithreaded Command Buffers**
- Dynamic Shader Linking
- New texture compression formats
- Read-only depth, conservative oDepth, …

Motivation - Multithreading

- In previous Direct3D versions, multithreaded rendering not really possible
  - Device access restricted to one thread unless you force brute force thread safety
  - Difficult to spread driver / runtime load over many cpu cores
- Ideally, you'd like threads for:
  - Asynchronous resource loading / creation
  - Parallel render list creation
- Direct3D 11 supports both of these

Multithreading - Interfaces

ID3D10Device

Check
Create
Draw
GS/IA/OM/PS/RS/SO/VS

ID3D10Buffer

Map/Unmap

ID3D11Device

Check
Create
GetImmediateContext
CreateDeferredContext

ID3D11DeviceContext

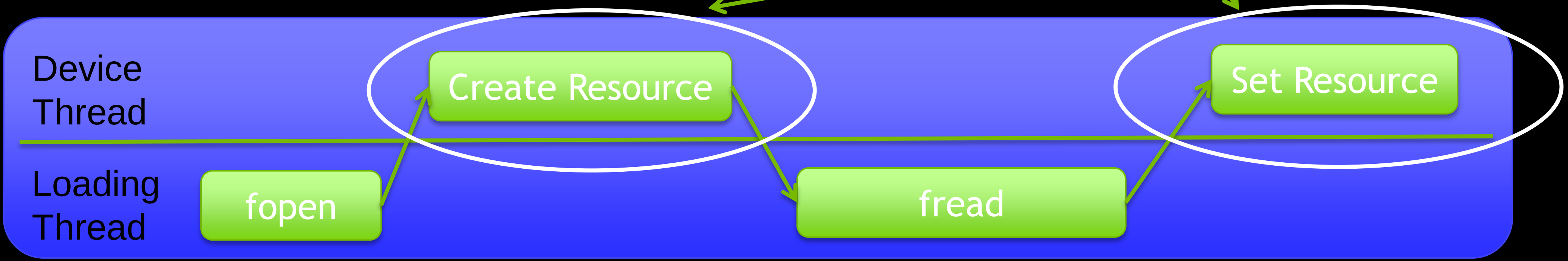
Draw
GS/IA/OM/PS/RS/SO/VS/**HS**/**DS**
Map/Unmap
FinishCommandList
ExecuteCommandList

Async Loading

- Previously, D3D required resource creation and rendering to happen from the same thread.

- So at best, it worked like this:

Potentially costly, D3D11 makes them async

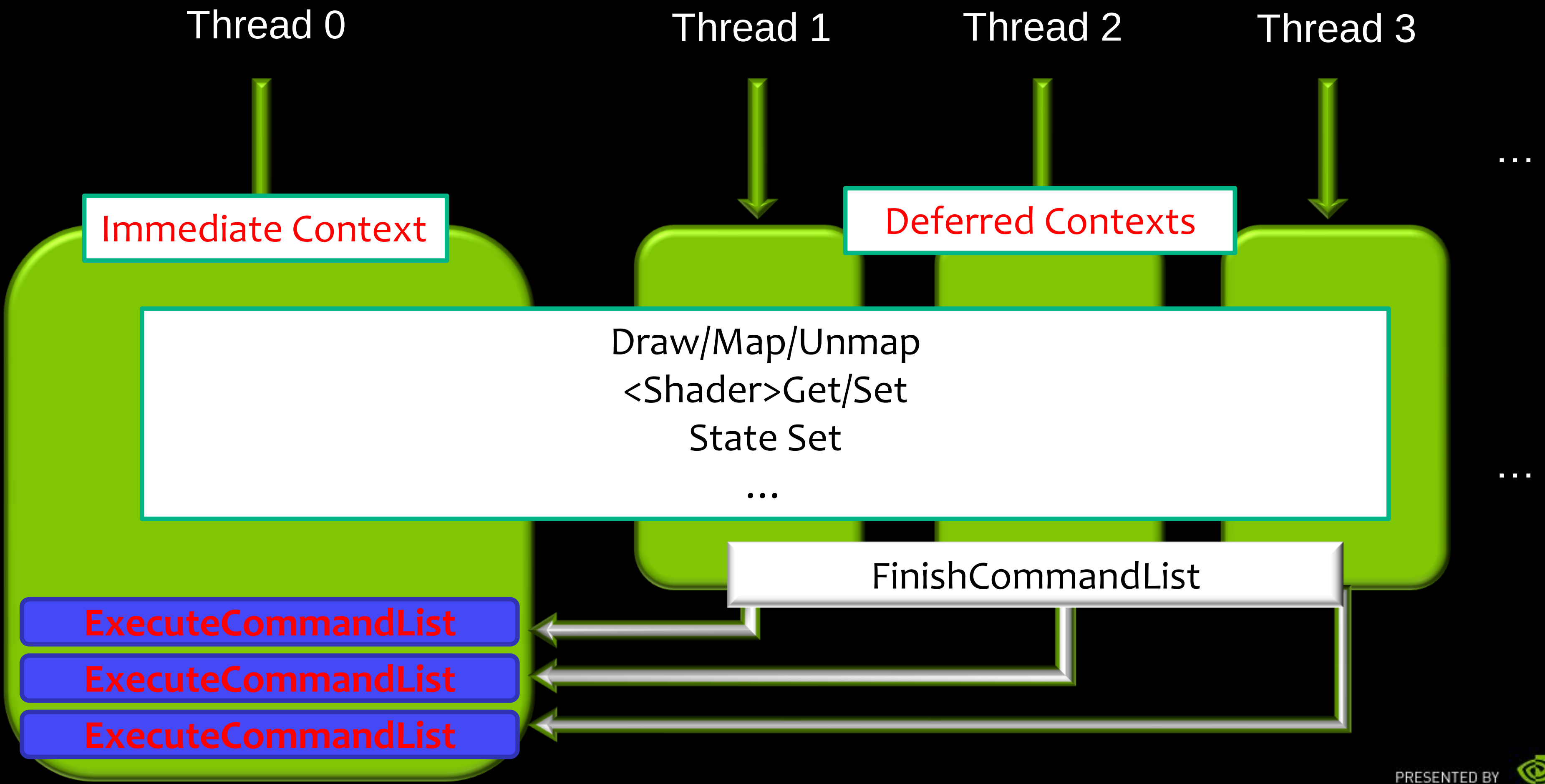


Async Loading

- With D3D11, rendering does not happen on the device, but instead on a *device context*
  - Immediate Context (actual rendering)
  - Deferred Contexts (display list creation)

- So the Device calls (create, etc.) can happen asynchronously

Multithreading - Contexts



Multithreading - Code Snippets

Main Thread

```
pd3dDevice->GetImmediateContext(&MyImmediateContext);

for (i = 0;  i < iNumThread;  ++i) {
    pd3dDevice->CreateDeferredContext(0, &MyDeferredContext[i]);
    thread[i] = _beginthreadex( …. );
}
```

Worker Thread

```
MyDeferredContext[id]->ClearRenderTargetView(pRTV, ClearColor);
        …. // (Draw, Map/Unmap, Shaders …)
MyDeferredContext[id]->FinishCommandList(FALSE, &MyCommandList[id]);
SetEvent(hEvent[id]);
```

Main Thread

```
WaitForMultipleObjects(iNumThread, hEvent, TRUE, INFINITE);
for (i = 0;  i < iNumThread;  ++i) {
    MyImmediateContext->ExecuteCommandList(MyCommandList[i], FALSE);
    MyCommandList[i]->Release();
}
```

Deferred Contexts - Tips

- Deferred Contexts display lists are immutable
- Map is only supported with DISCARD
- No readbacks or getting data back from the GPU
  – Queries, reading from resources, etc.
- No state inheritance from immediate context
  – Start with default state
  – You should still aim to reduce redundant state submission
- Some cost to creating / finishing / kicking off DL
  – Favor large display lists, not tiny ones
  – 100+ draw calls per display list is good

Outline - Dynamic Shader Linking

- Why DirectX 11?
- Direct Compute
- Tessellation
- Multithreaded Command Buffers
- **Dynamic Shader Linking**
- New texture compression formats
- Read-only depth, conservative oDepth, …

Dynamic Shader Linking - Motivation

- With complex materials, you currently have two choices:
  - Über Shader
  - Preprocessor shader combinations

- Neither is ideal

Dynamic Shader Linking - Motivation

Über Shader

```
if ( bLighting )
        doLighting()
if ( bTexture )
        doTexturing()
if ( bFog )
        doFogging()
```

Expensive flow control!

Custom Shaders

```
Shader A:
        doLighting()
Shader B:
        doLighting()
        doTexturing()
Shader C:
        .....
```

Explosion of shaders!

Dynamic Shader Linking

- Dynamic Shader Linking is here to get the best of both worlds
- Allows you to define *interfaces*
- Allows you to define classes which inherit from these interfaces
- Resolves the correct target at runtime with little overhead

Dynamic Shader Linking - Example

```
interface   iLight   {
    float4   Calculate(…);
};

class   cAmbient   :   iLight {
    float4     m_Ambient;
    float4   Calculate(…) {
        return    m_Ambient;
     }
};

class    cDirectional    :   iLight {
    float4     m_Dir;
    float4     m_Col;
    float4     Calculate(…) {
        float  ndotl = saturate(dot(…));
        return   m_Col * intensity;}
};
```

```
iLight     g_Lights[4];
cbuffer  cbData  {
    cAmbient          g_Ambient
    cDirectional    g_Directional0;
    cDirectional    g_Directional1;
    cDirectional    g_Directional2;
    cDirectional    g_Directional3;
 }

float  accumulateLights(…) {
    …
    for (uint    i =   0;  i  <  g_NumLights; … ) {
        col += g_Lights[i].Calculate(….);
    }
…
}
```

Define an implementation of the interface

At this point the concrete function is decided

Outline - New Texture Compression

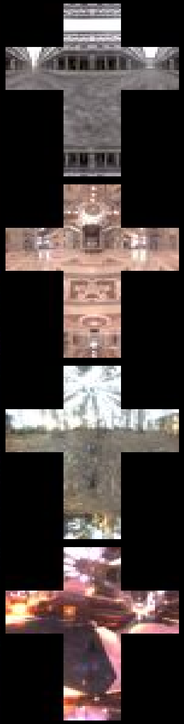
- Why DirectX 11?
- Direct Compute
- Tessellation
- Multithreaded Command Buffers
- Dynamic Shader Linking
- **New texture compression formats**
- Read-only depth, conservative oDepth, …

New Compression Formats

- Two new compression formats: BC6H & BC7
- BC6H: HDR texture compression
  - RGB only
  - Signed and Unsigned
  - 16 bit floating point values
  - 6:1 compression
- BC7: High Quality LDR texture compression
  - RGB with optional Alpha
  - 3:1 (RGB) or 4:1 (RGBA) compression

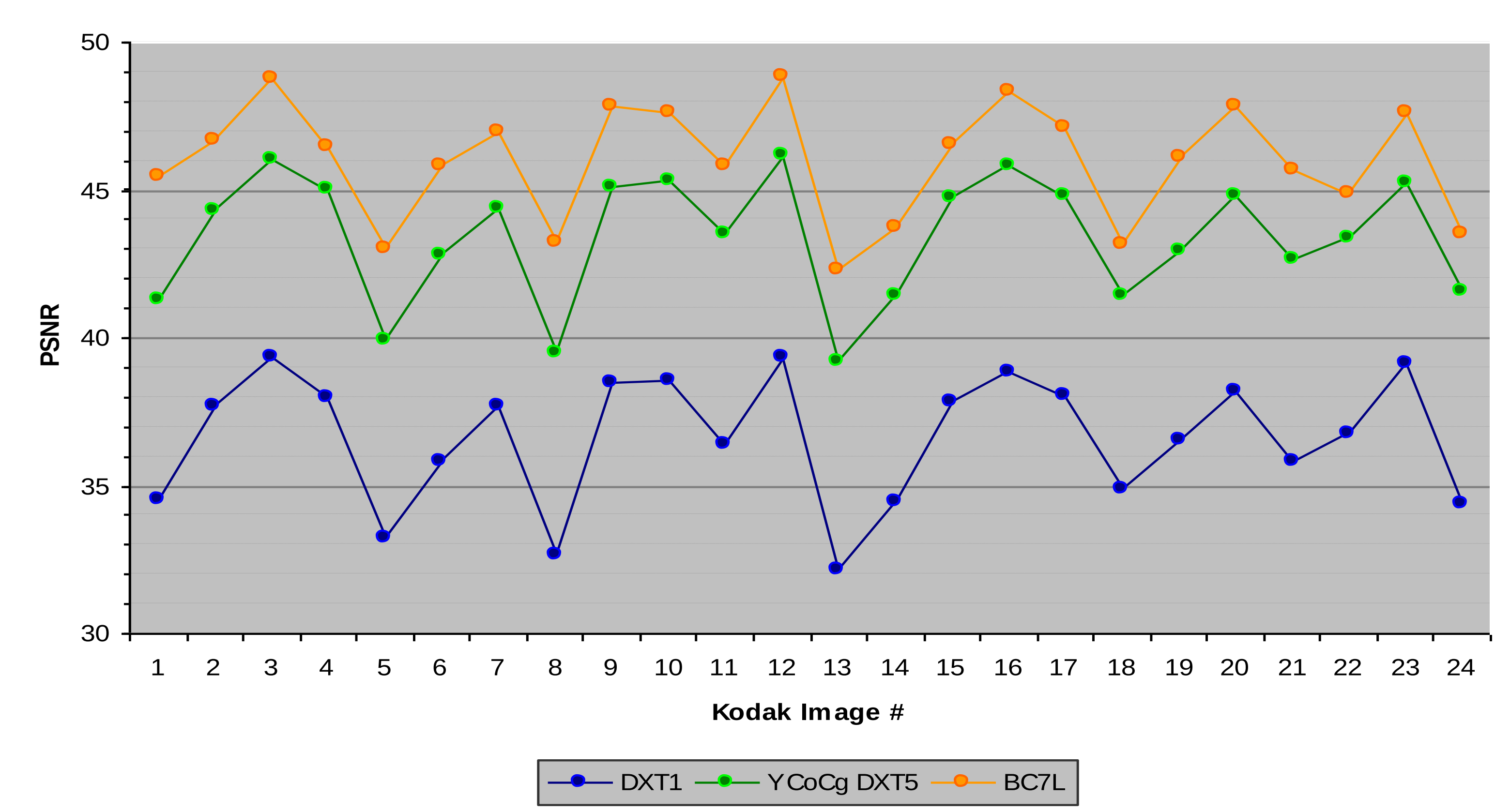
BC6H Compression Quality

- Objective:
  - Replace uncompressed FP16x4 and RGBE textures



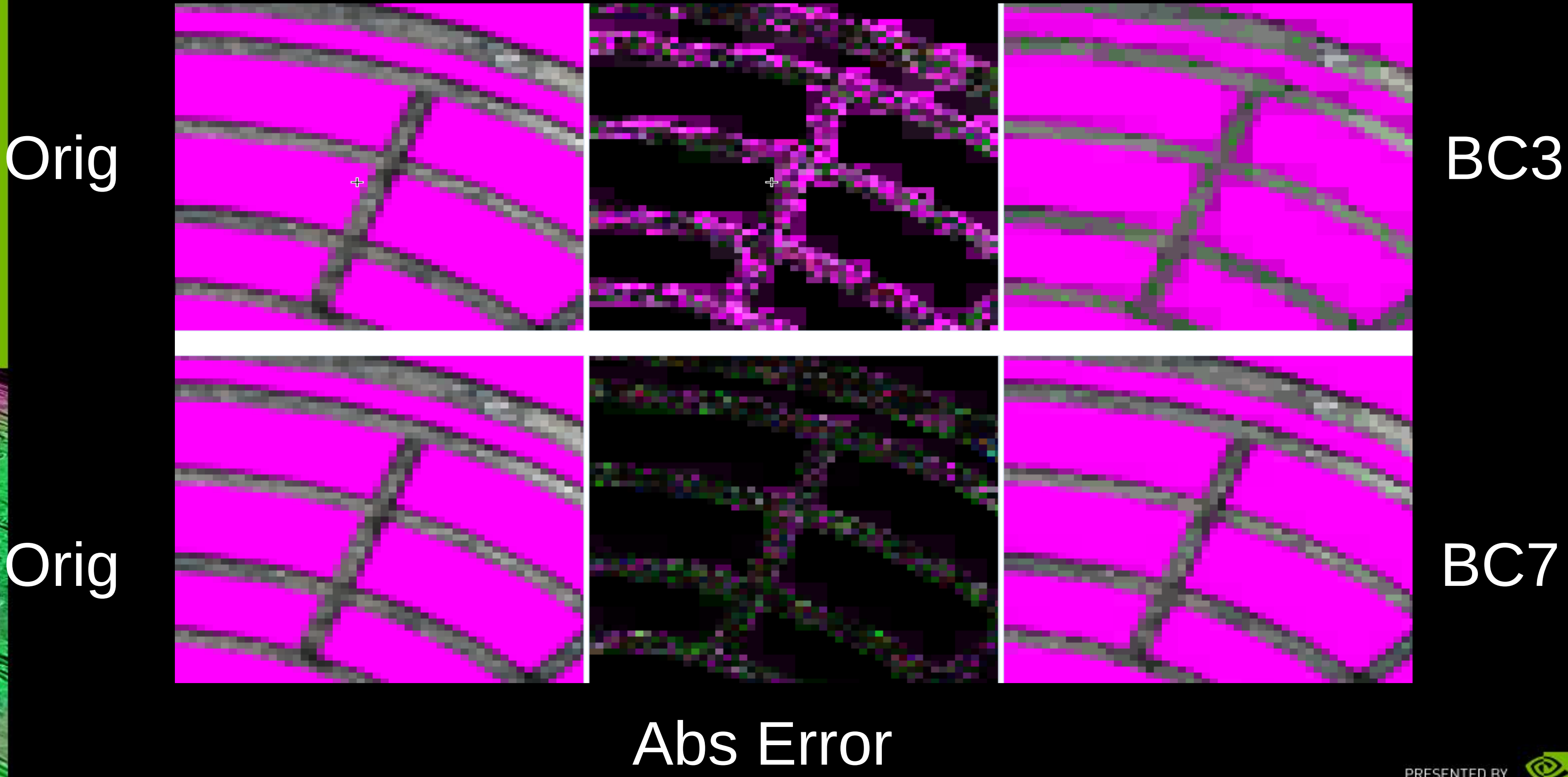
	BC6H	LUVW	RGBE	FP16x4
uffizi cross	63.75	63	70	108
stpeters cross	62.97	66	69	95
rnl cross	62.99	70	72	129
grace cross	61.72	75	64	133
Average PSNR	62.62	68.5	68.75	116.25
Average PSNR / Bits per pixel	7.83	4.28	2.15	1.82

BC7 Compression Quality



PRESENTED BY NVIDIA.

Texture Compression - BC7



Outline - New Depth Features

- Why DirectX 11?
- Direct Compute
- Tessellation
- Multithreaded Command Buffers
- Dynamic Shader Linking
- New texture compression formats
- **Read-only depth, conservative oDepth, …**

Read-Only Depth - Motivation

- In previous Direct3D versions you cannot bind a depth buffer for depth test and also read it in shader
  - Implies potential data hazards
- But if depth writes are disabled, there actually is no hazard
  - API was not expressive enough to capture this

Read-Only Depth - Implementation

```
#define  D3D11_DSV_FLAG_READ_ONLY_DEPTH     0x1;
#define  D3D11_DSV_FLAG_READ_ONLY_STENCIL   0x2;
```

```
typedef struct D3D11_DEPTH_STENCIL_VIEW_DESC
{
        DXGI_FORMAT          Format;
        D3D11_DSV_DIMENSION  ViewDimension;
        DWORD                Flags;
        union
        {
                D3D11_TEX1D_DSV            Texture1D;
                D3D11_TEX1D_ARRAY_DSV      Texture1DArray;
                D3D11_TEX2D_DSV            Texture2D;
                D3D11_TEX2D_ARRAY_DSV      Texture2DArray;
                D3D11_TEX2DMS_DSV          Texture2DMS;
                D3D11_TEX2DMS_ARRAY_DSV    Texture2DMSArray;
         };
} D3D11_DEPTH_STENCIL_VIEW_DESC;
```

Read-Only Depth - Applications

- Soft Particles!
  - Typically alpha blended, so you test depth but don't write
  - Need access to depth buffer to soften edges as you near another surface

Conservative oDepth

- Modifying the depth value in the pixel shader currently kills all early-z optimizations
  - Early-z optimizations are critical to high performance
- But many algorithms do not arbitrarily change depth
  - Direct3D 11 can take advantage of this to improve performance

Conservative oDepth

- Two new system values
- Example (depth comparison func LESS_EQUAL):

```
float depth : SV_DepthGreaterEqual
```

  - You're promising to push the fragment into the scene
  - So Early Z Cull will work!

```
float depth : SV_DepthLessEqual
```

  - You're promising to pull the fragment towards the camera
  - So Early Z Accept will work!

Summary

- Direct3D 11 is fast…
  - Multithreading, new depth functionality
- …flexible…
  - Dynamic shader linking, broad compatibility
- …and enables higher quality effects
  - Tessellation, compute, new texture compression

Questions?

cem@nvidia.com