Let’s Get Small

Understanding MIP Mapping

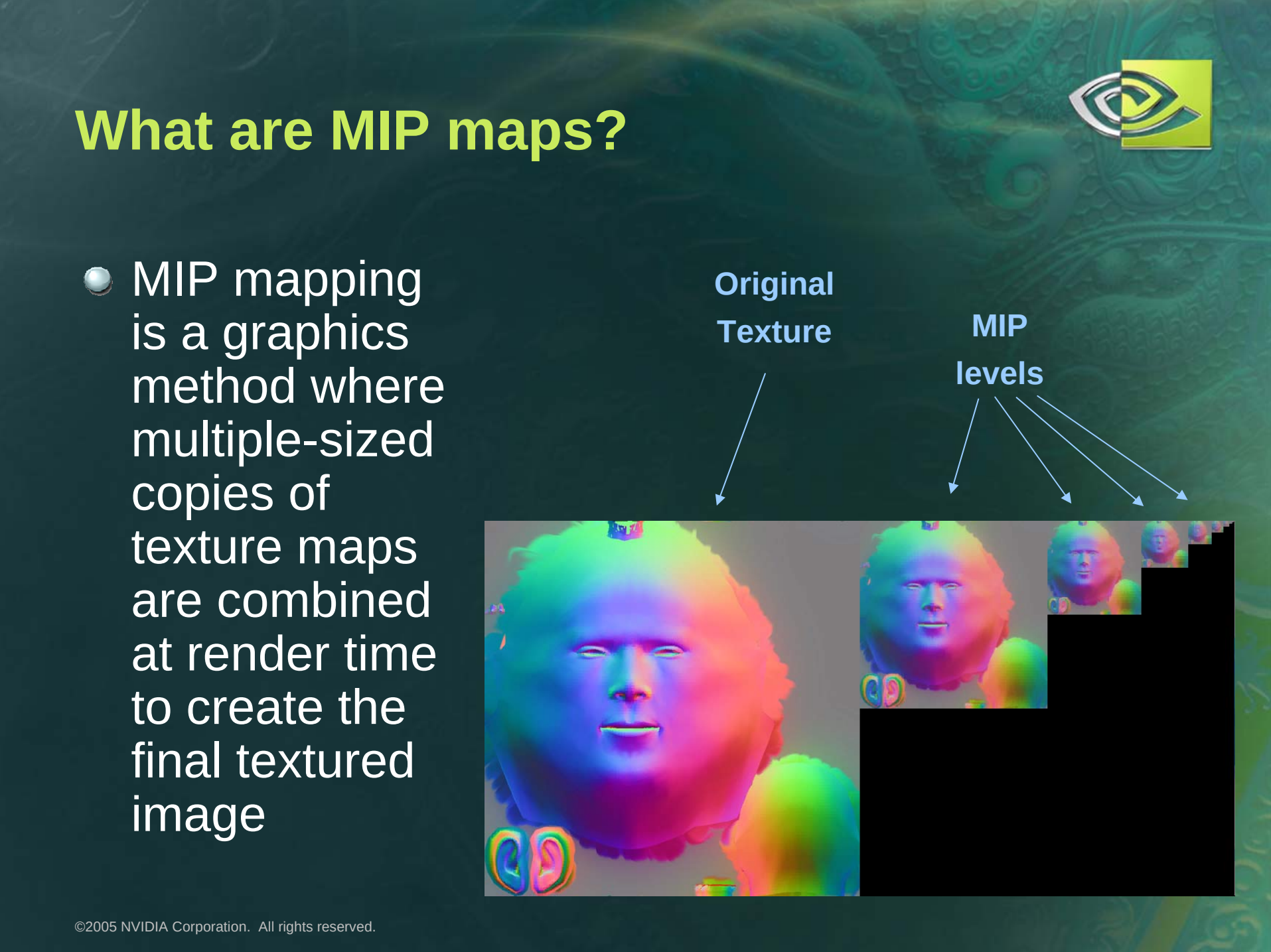
Kevin Bjorke, NVIDIA

June 2005

Understanding MIPs and Texturing

- What they are
- How they work
- How to use them
- How to abuse them
- How they affect different kinds of maps
- Bump maps, Normal maps, Color Maps
- Workflow & making your own MIP maps
- More!

What are MIP maps?



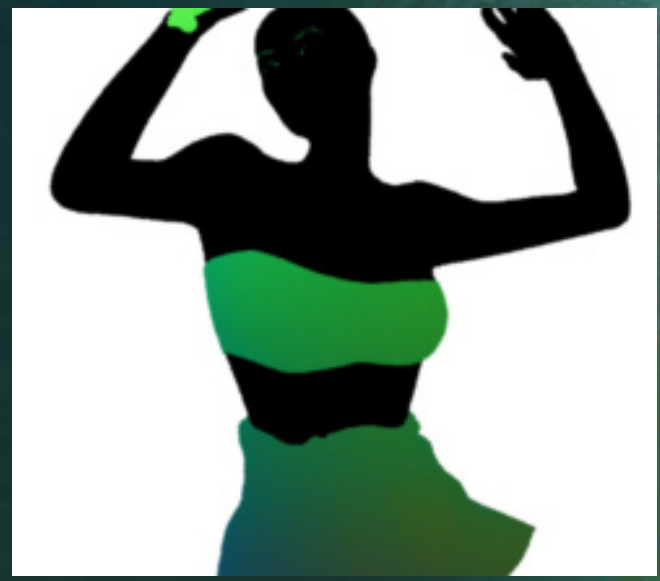
- MIP mapping is a graphics method where multiple-sized copies of texture maps are combined at render time to create the final textured image

Why Artists Should Care About MIPs

- Quality
  - Controlling the MIP map means giving the artist control over the full range of model appearance
- Performance
  - Savvy use of MIP maps gives games the highest texturing performance
- Avoid Wasting Artist's Time
  - Don't waste time painting details that will never be seen! Understanding MIPs lets you stay focused on what's visually important

Texture Mapping Basics

- Texture mapping coordinates are assigned in 3D per-vertex, but maps actually get applied in 2D
- Each rasterized triangle will have a specific range of texture *derivatives* – that is, the amount of texture image that will be used for that pixel



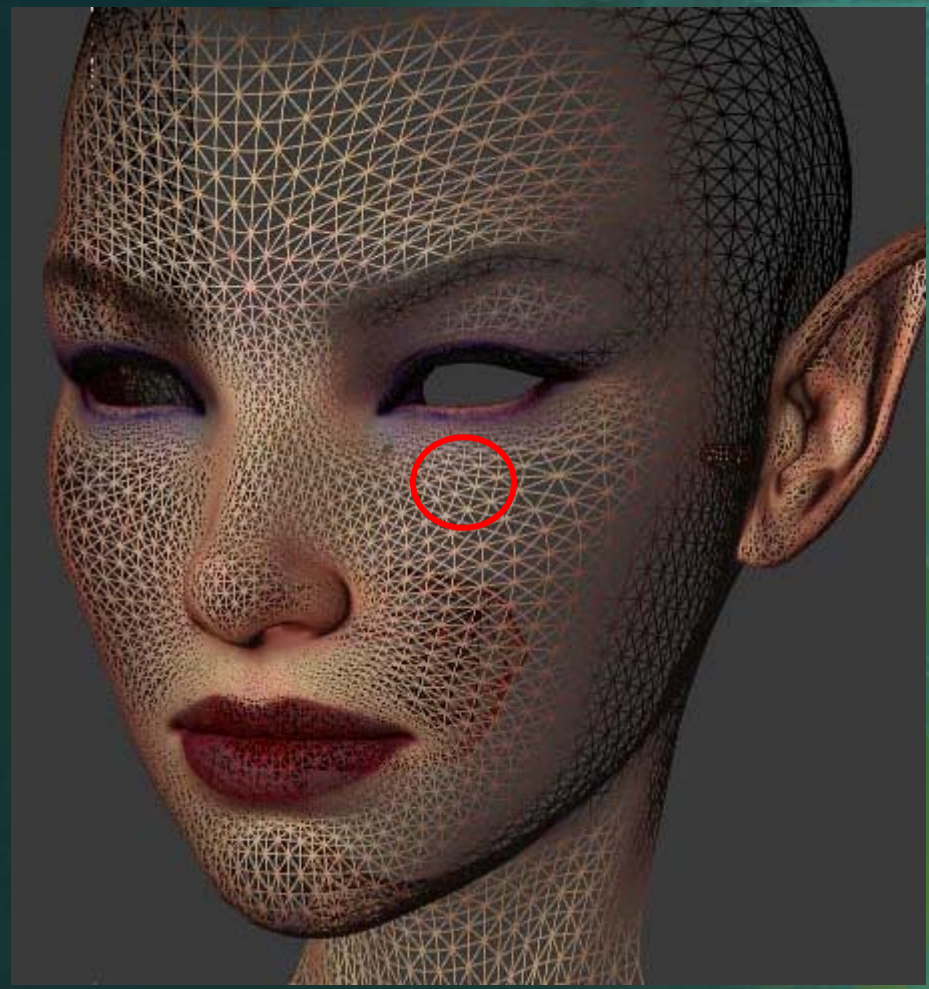
"MRT" visualization
of texture coordinates

Mapping a 2D texture to a triangle

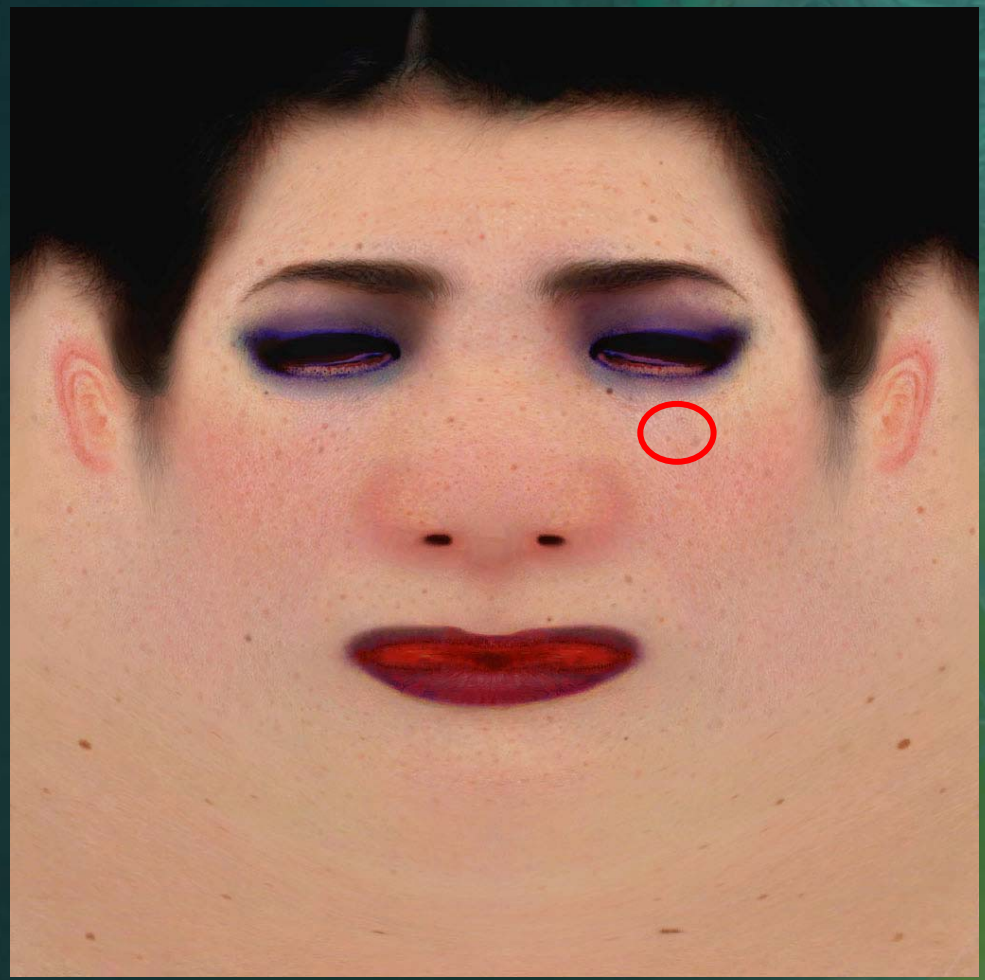
- For each area of a surface, there is a single triangle in 3D space, mapped onto screen 2D space, and also mapped into 2D texture space

Mapping a 2D texture to a triangle

- Each 3D triangle is also a 2D triangle on-screen



Source Map



- Each screen triangle maps to some triangular area on the texture map
- This area may be rotated, stretched, and/or squeezed, but it’s still a triangle

We can look at the UVs directly

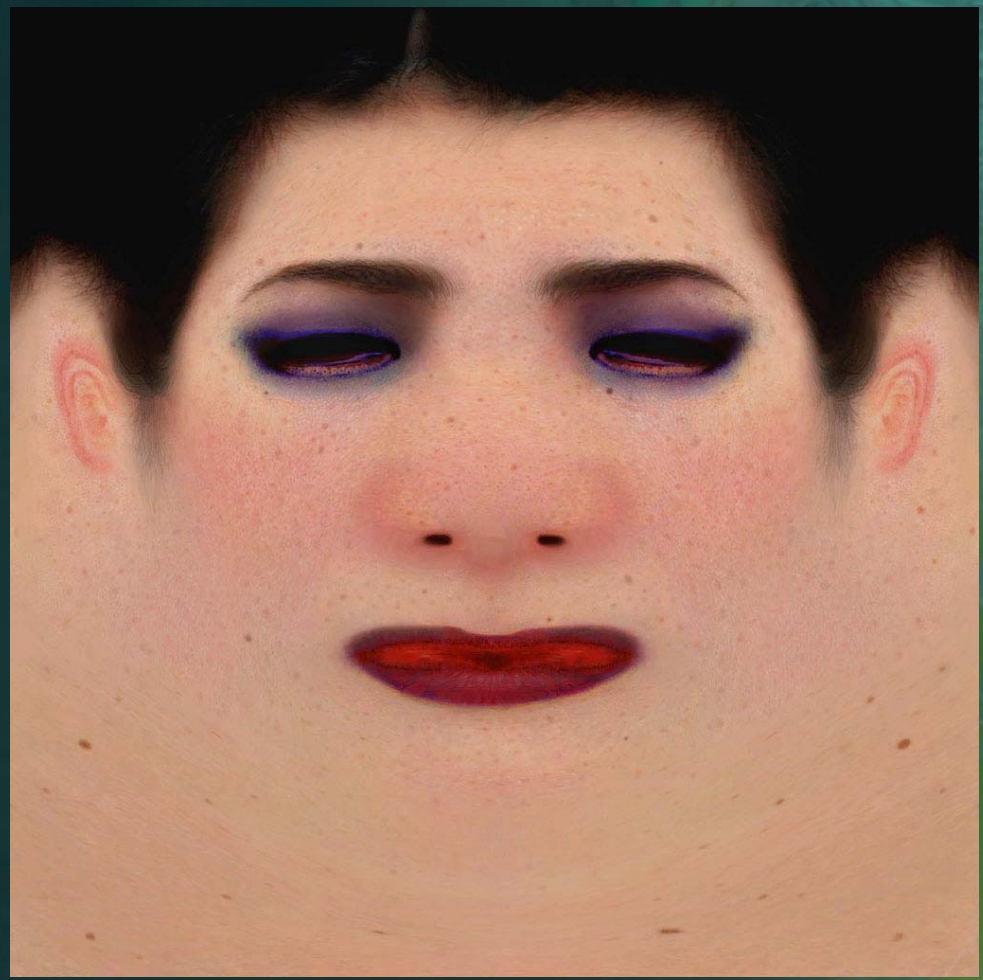
- Here shown as colors (red/green)
- We can see areas on-screen of faster or slower gradation in the color
- The gradation speed is the derivative

Looking directly at the UV derivatives

- UV Derivatives are *how much the UVs change per pixel*
- We can also think of them as “the amount of texture stretching”
  - Math trivia: we only get first-order derivs, second-order is always zero. See the faceting?

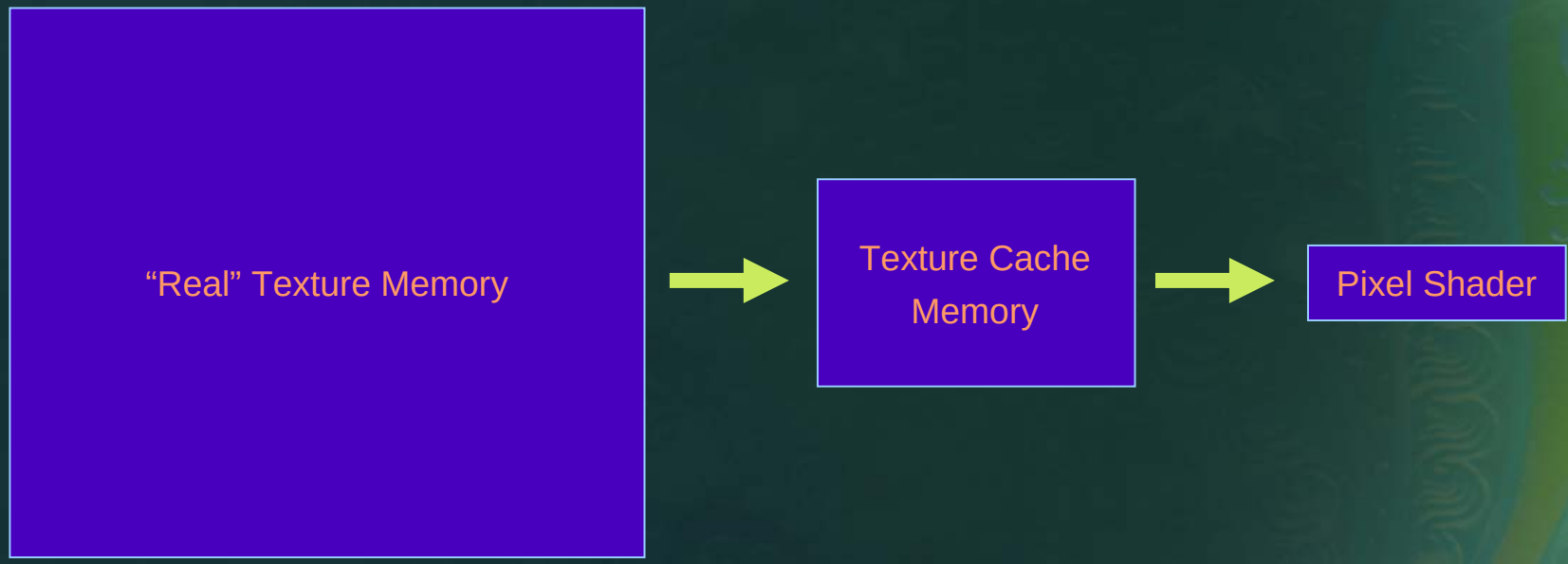
Texture Filtering

- Stretching and squeezing of texture triangles is “image filtering”
- The hardware can do simple “linear” filtering between texels



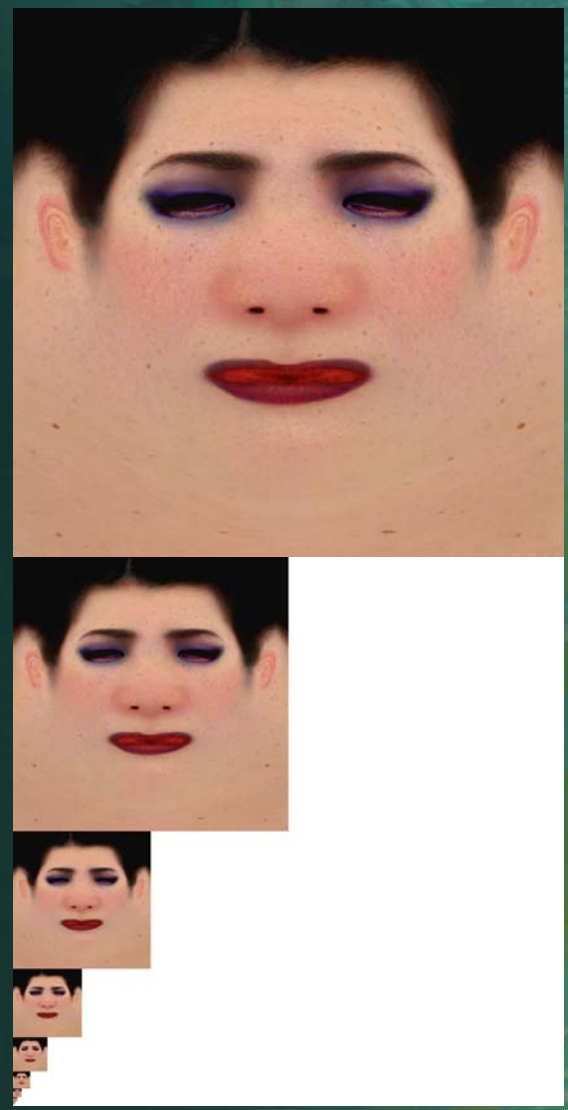
GPU Texture Caching

- GPUs are designed to minimize texture *latency* – that is, the amount of time between a shader makes a texture request and the time the shader receives a usable color
- Shader units run faster than memory, so a key strategy is *caching,* based on “best guesses” of access patterns
- When the guess is wrong, performance suffers



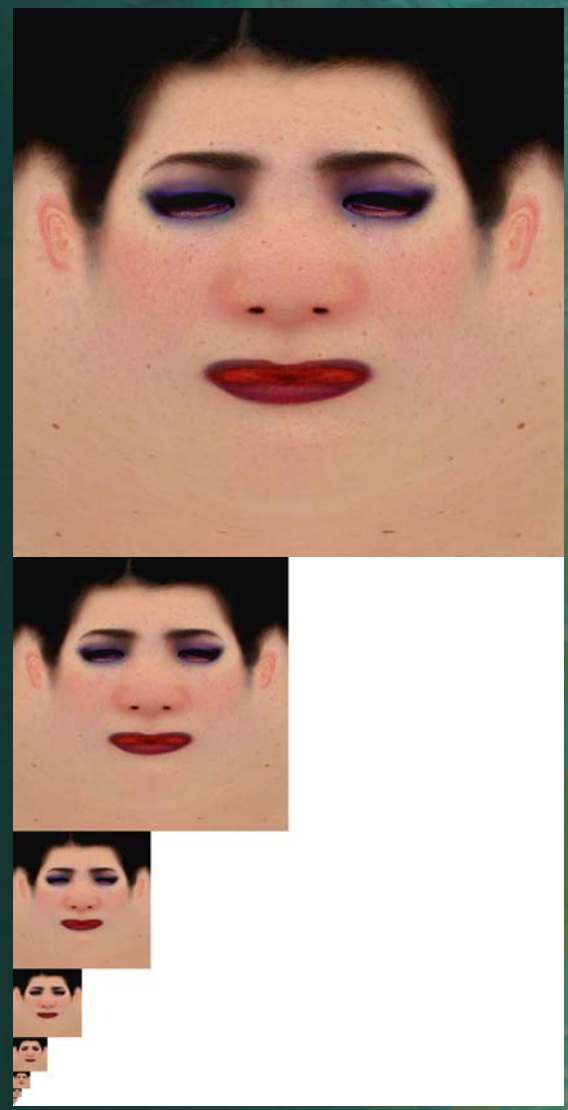
MIP Maps

- MIP maps provide pre-filtered values for texture compression and good caching
- The tradeoffs are performance & control versus texture memory
- A MIP'd texture occupies twice the total texture memory…
- *…but* may occupy far less active cache! So overall perf is better
  - Only the MIP levels being actively used will be in the cache



MIP Level Creation

- MIP levels (or just “MIPs”) *can* be automatically created in the driver at texture-load time – or better, explicitly-defined in the texture file
- File MIP levels can be created automatically, or tweaked by hand
- DDS is the most common MIP-able file format in games, though others (e.g., multi-resolution TIFF) do exist



MIP bias

- “MIP bias” is a way to force the GPU to use a different MIP level than the one automatically chosen by DirectX
- Positive bias pushes to lower (smaller) MIP levels, negative to higher
- Negative bias may *look* sharper in a still frame – but it will alias and sparkle when in motion!
- Worse, it makes poor use of the texture cache, resulting in degraded performance
- *Don’t mistake aliasing for detail*

Render-Time Filter Methods

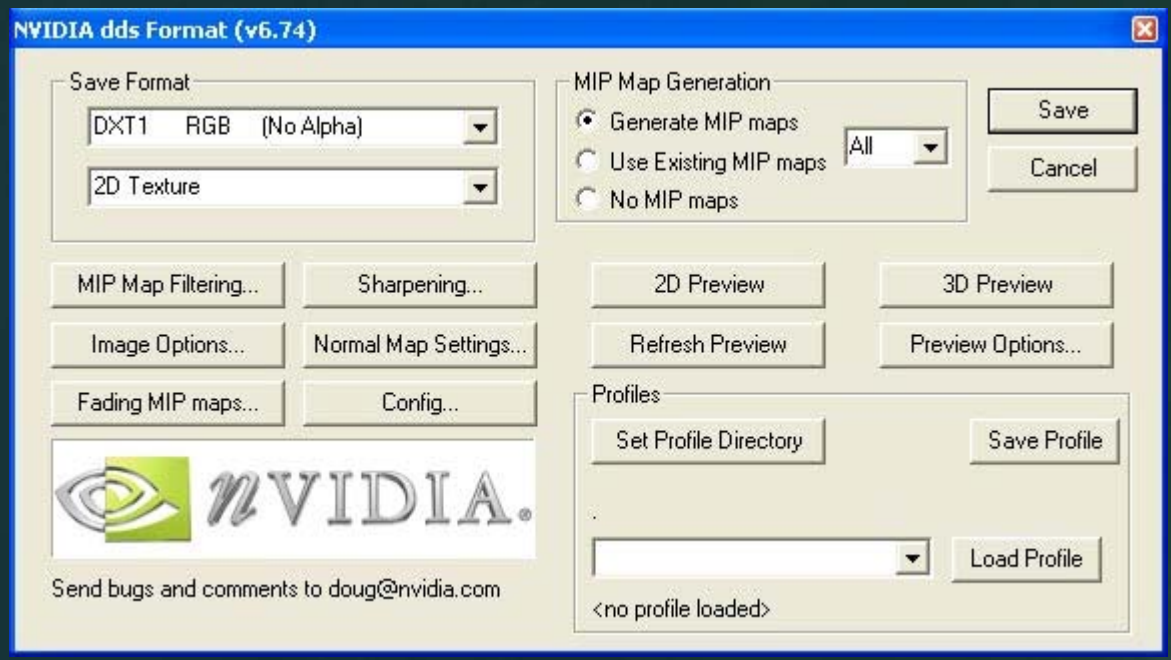
- Aniso, trilinear, bilinear…
- Artists can’t control these
- There are performance and quality issues that ultimately should be balanced by the game programmers at run time
- And the final user may over-ride the programmers’ settings anyway!
- In general: trilinear will go twice as slow as linear, and both are faster than aniso

NVIDIA DDS Conversion Tool

- http://developer.nvidia.com/object/dds_utilities.html
  - tool improves regularly, so be sure you have an up-to-date version
- **RULE #1: ALWAYS SAVE THE .PSD, WITH LAYERS**
  - You never know what shaders might come along later – you might need to re-arrange the channels, add specular maps, who knows?
  - You might need the .psd's to build atlases or other formats later, too.

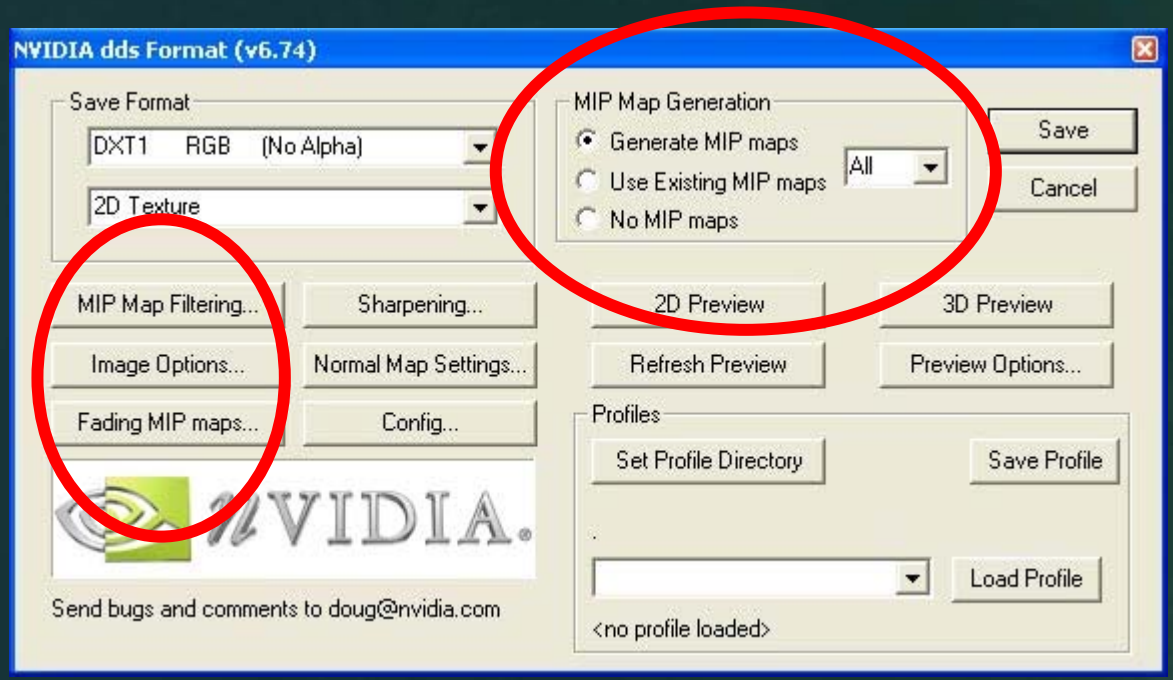
Making Automatic MIPs in Photoshop

- NVIDIA DDS Plug-in can export to DDS & build MIPs
- http://developer.nvidia.com/object/photoshop_dds_plugins.html
- “Save as…” then select file format: DDS
- After choosing a filename, this dialog appears:



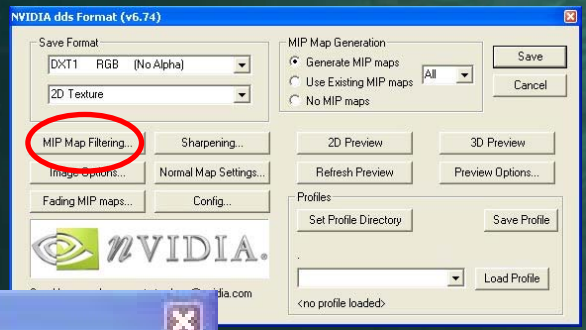
Creating MIP Maps in Photoshop

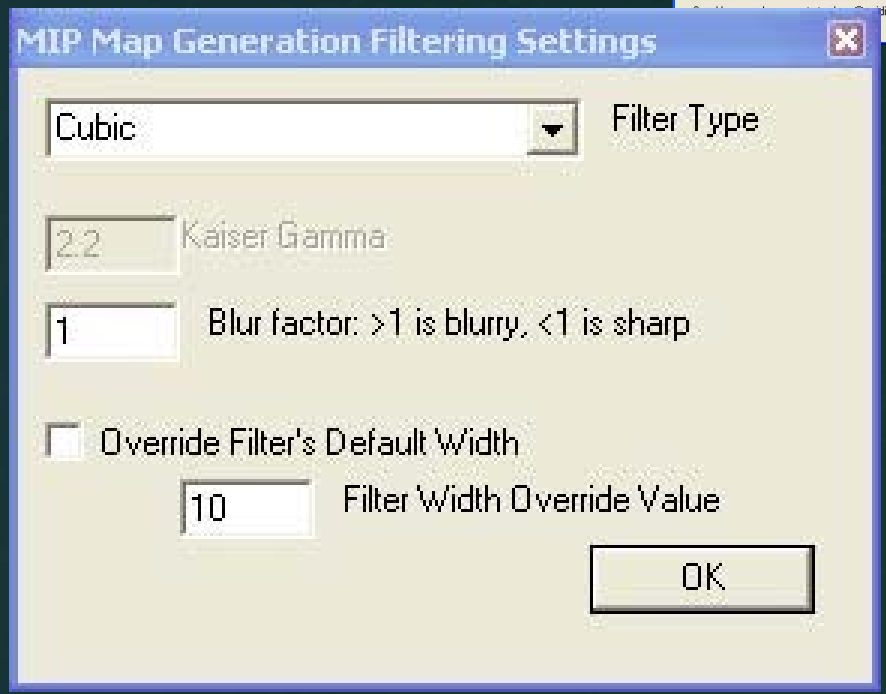
- For MIP generation, these controls are key:



MIP Map Filtering…

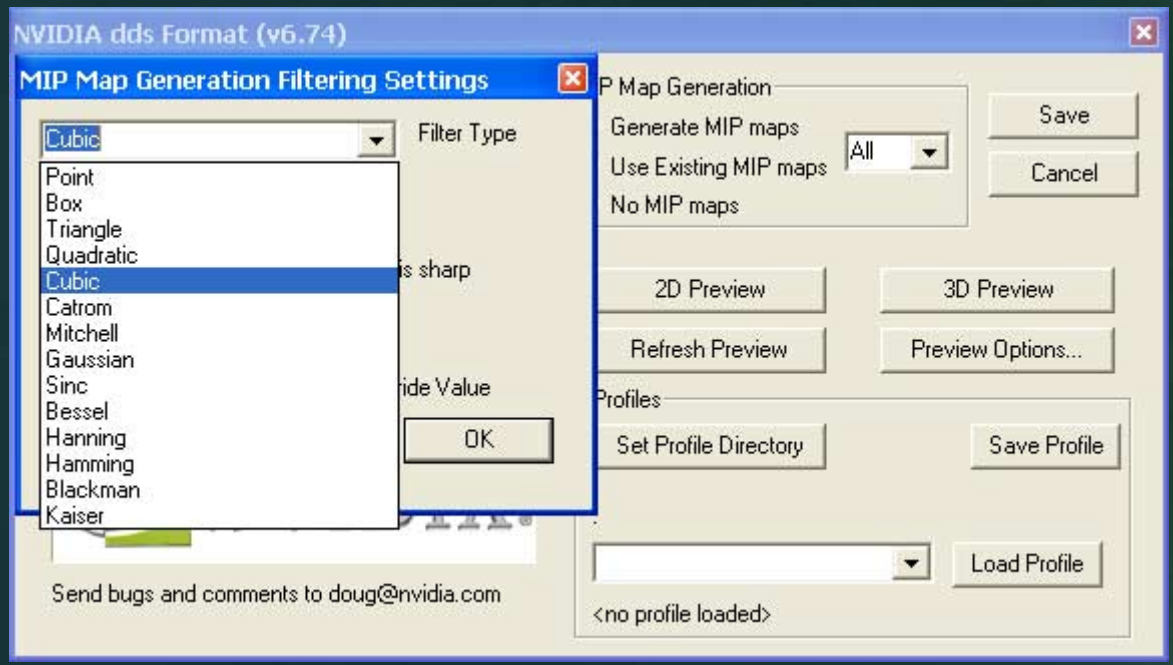
- Select Filter and Size





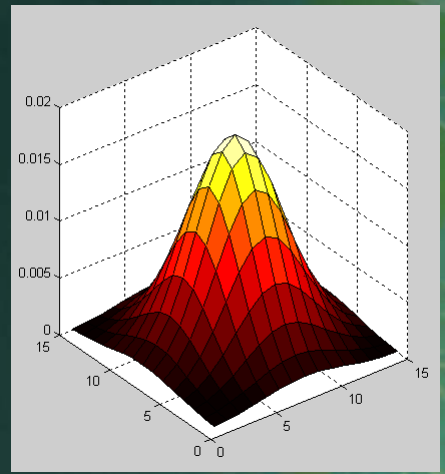
Many Choices for Filtering

- Each provides a different character in terms of sharpening and smoothness
- But which one to choose?



Favored Filter Types

- Each filter has its own character, which is why there are choices. The most-common choices are:
  - Point – “Nearest Neighbor” – that is, no filtering
  - Box – Simplest, naïve and fastest
  - Cubic – Like Photoshop’s own “bicubic”
  - Mitchell – good balance of quality/speed
  - Kaiser – slowest (?)
    - “Kaiser Gamma” currently unimplemented
- Each gives a different pattern of weights to neighboring pixels



Sharpening Filters

- An alternative to MIP Filters, or can be used in tandem
- Sharpening can help preserve “valuable” details, esp. edges (more on this idea later)

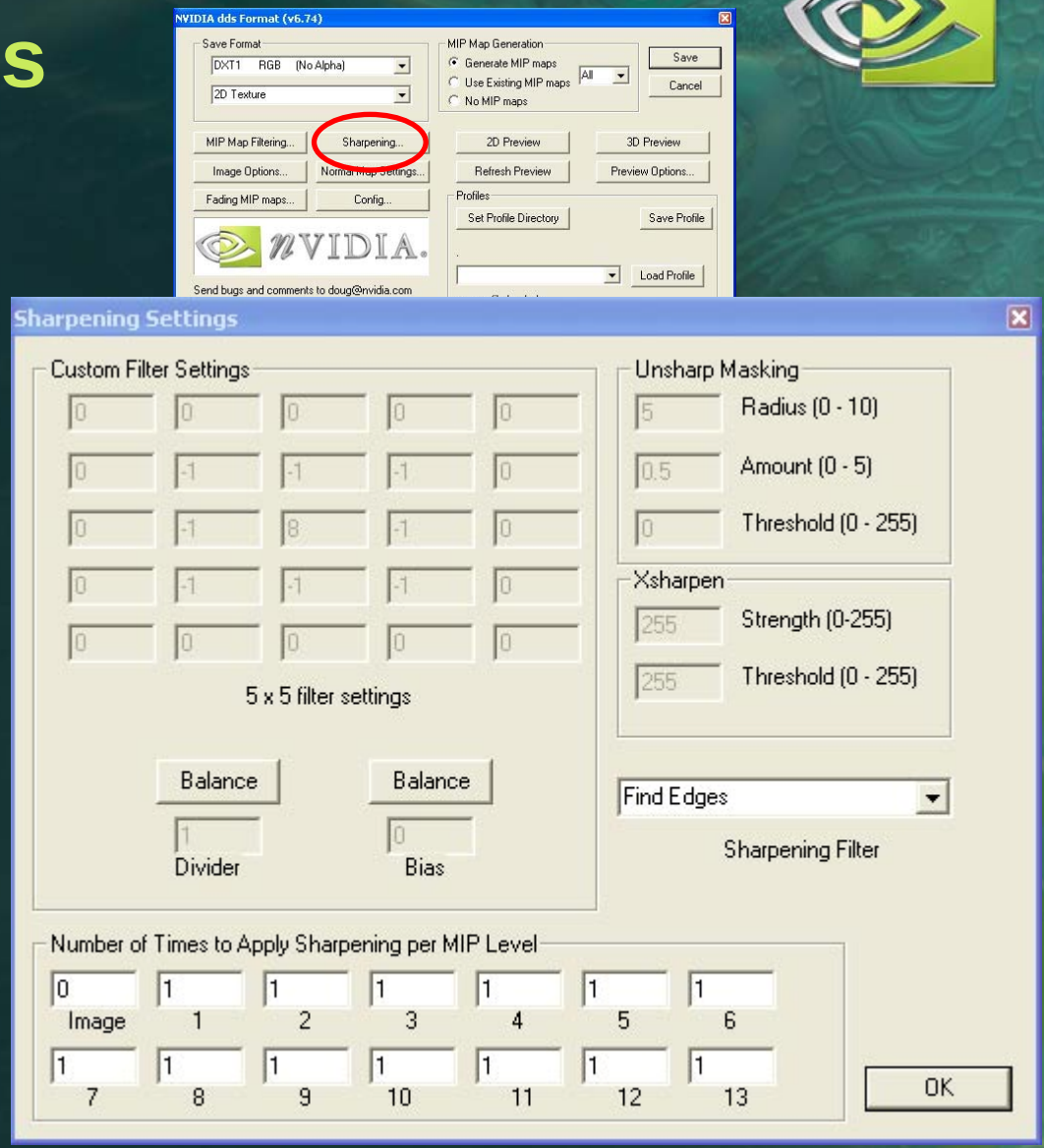
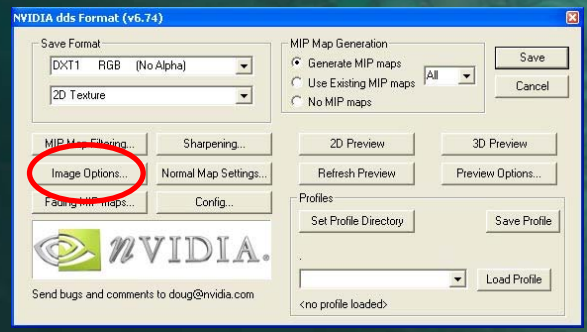
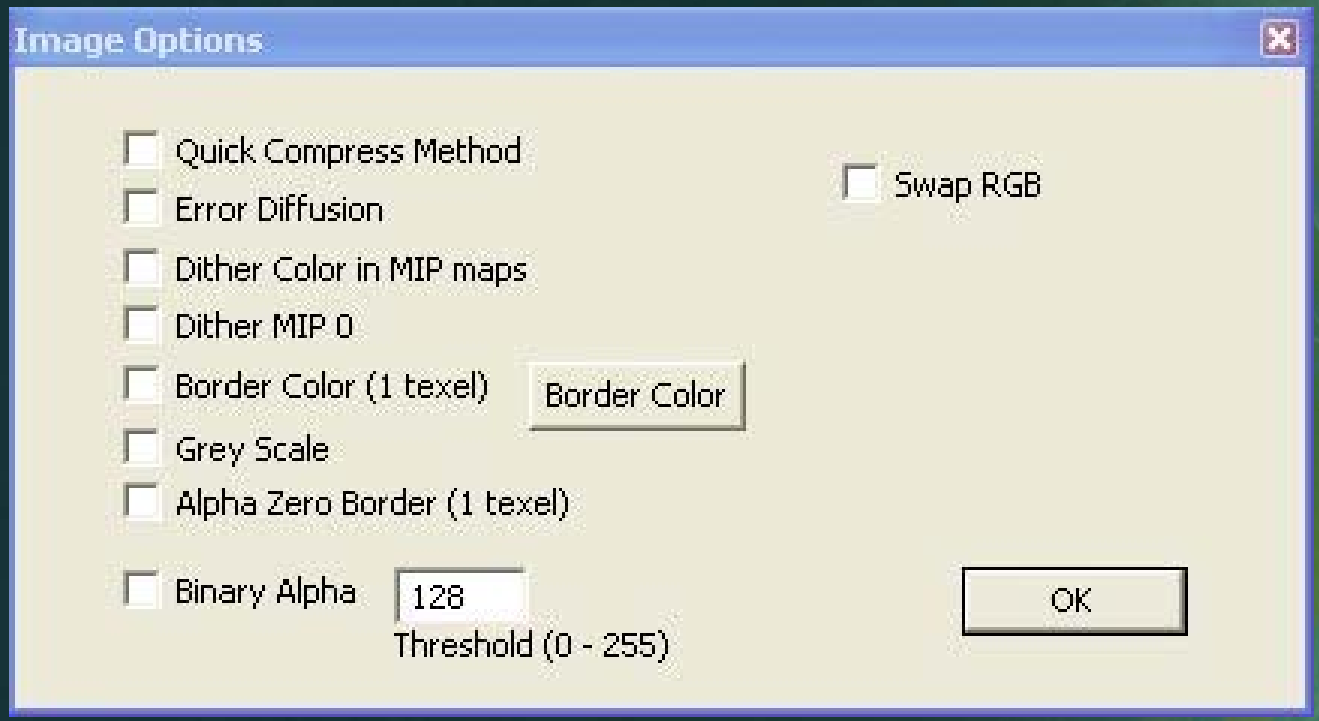


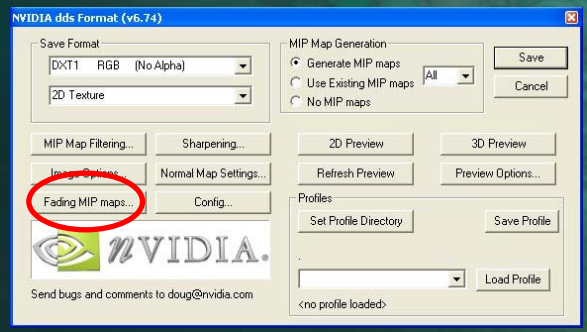
Image Options…

- Dithered color can give greater apparent color range to lower MIP levels

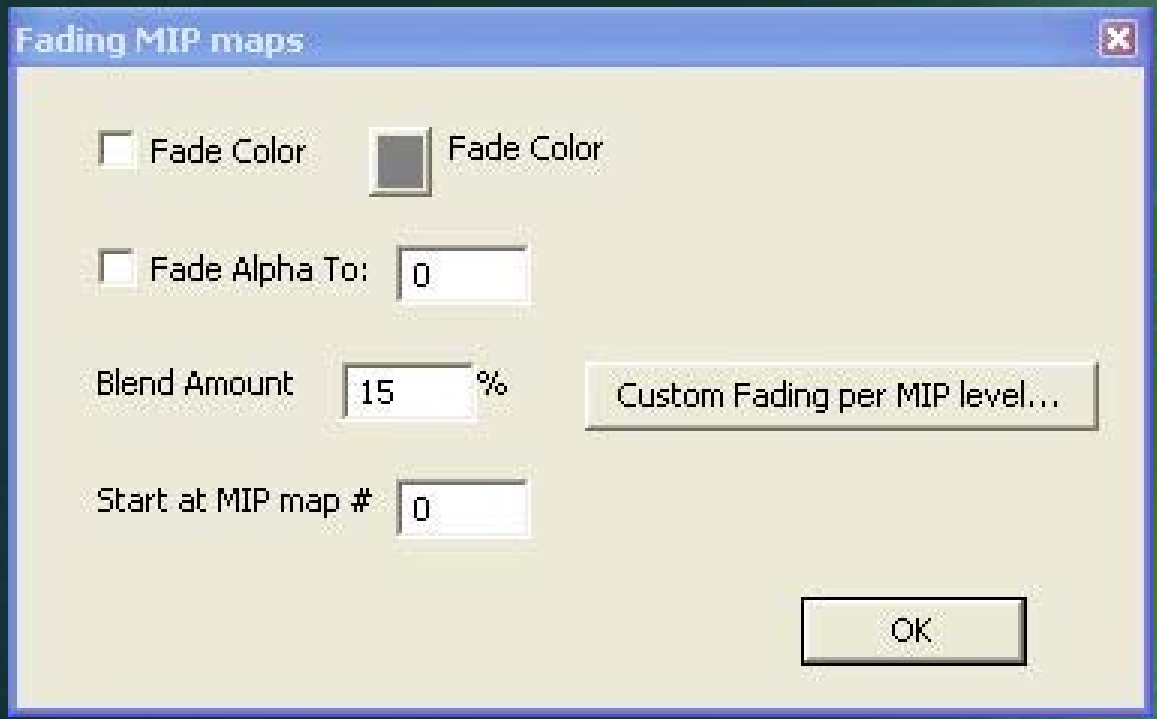




Fading MIP maps…

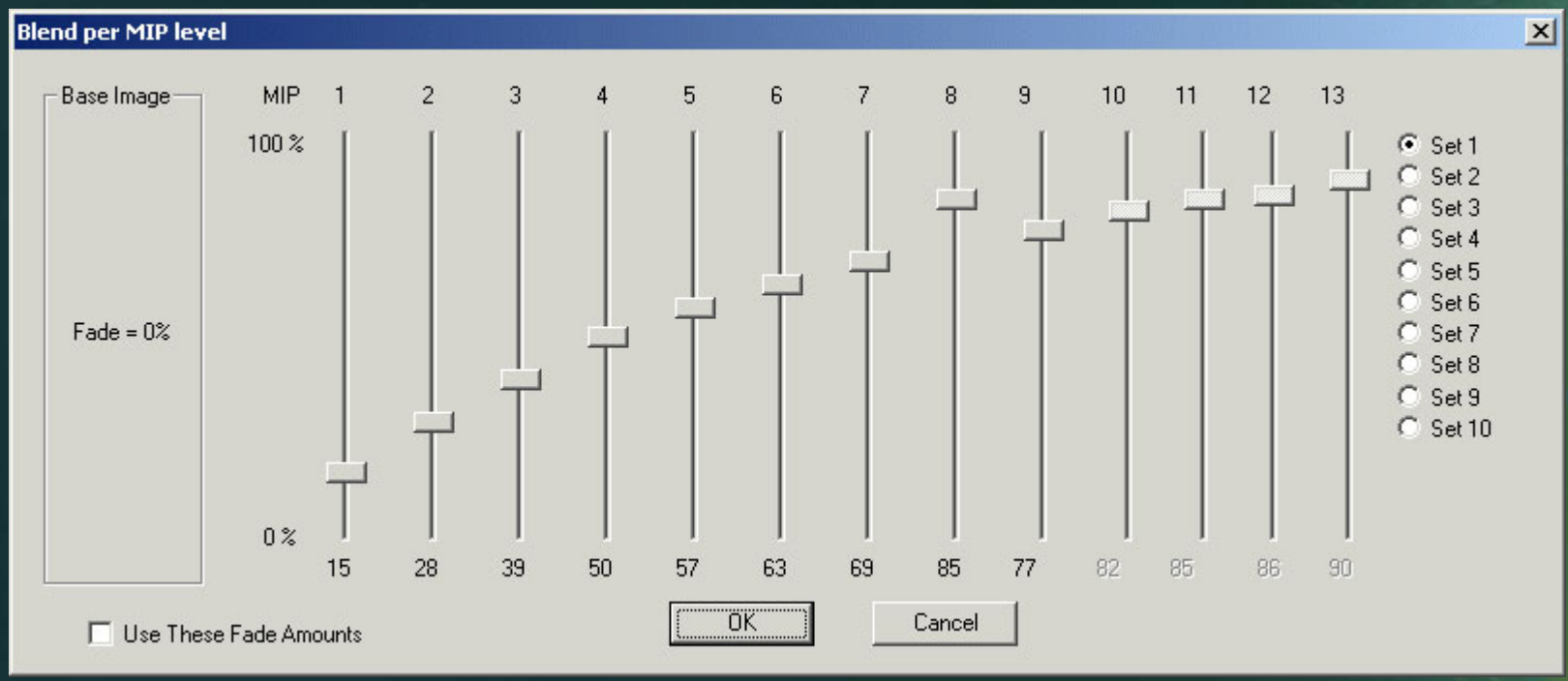


- Fade to gray for Normal Maps
- For color, consider the likely background colors
- “Poor Man’s Fresnel” – give objects a halo by picking a bright color

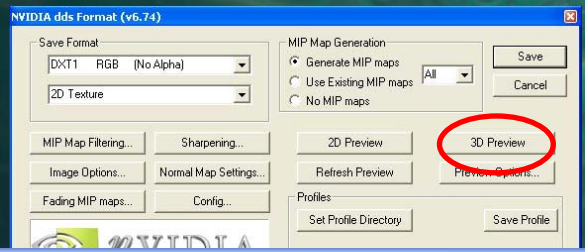


Advanced Fading Options

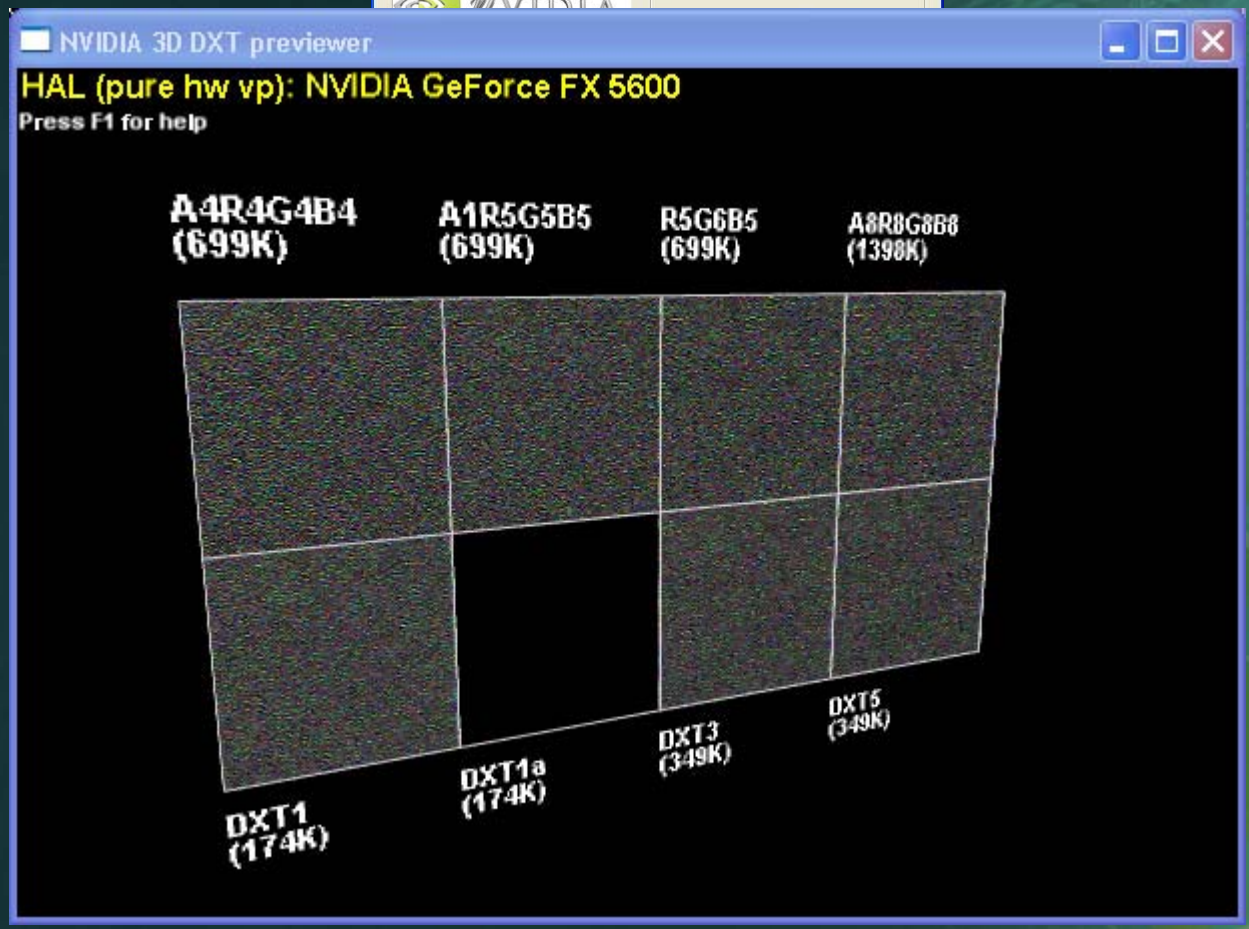
- At the extreme, specific levels can get exact fading for effect



3D Texture Preview



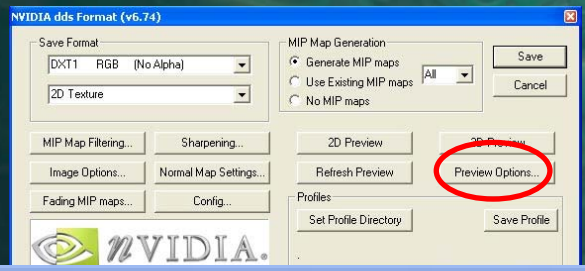
- Great Feature – can see results of all MIP and format choices
- Use 3 mouse buttons for three types of motion



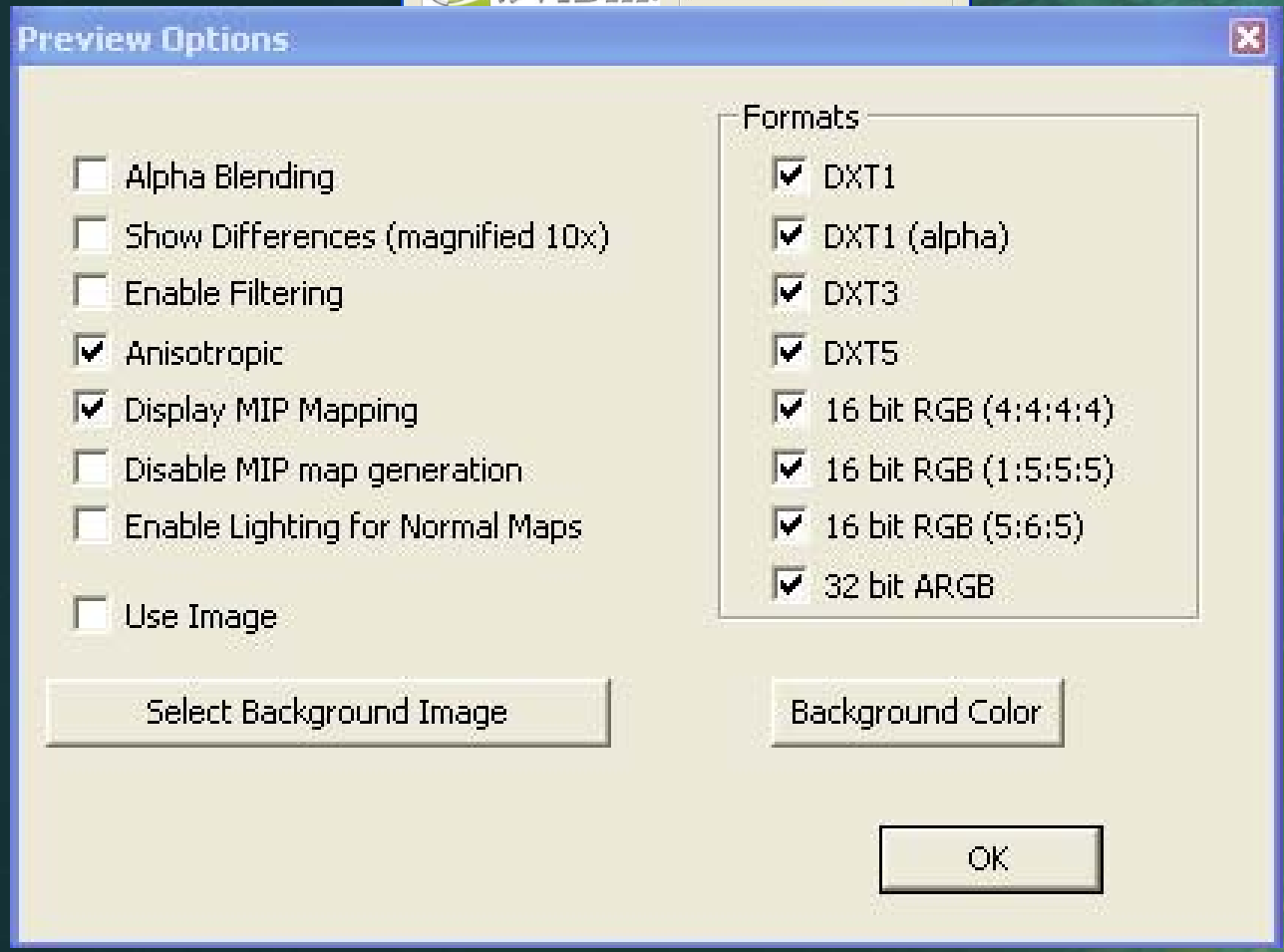
3D Preview Features

- Draw against real backgrounds etc

Preview Options



- Preview only the formats you're likely to actually use, potentially against a specific background color or image



Formats: DXT Compression Schemes

- DXT1-DXT5 all represent the same data
- DXT1 compresses 8::1 – no alpha or with single transparent color
- DXT3 & DXT5 compress 4::1 and have alpha
- DXT1 can exist compressed even in GPU cache, providing the best performance

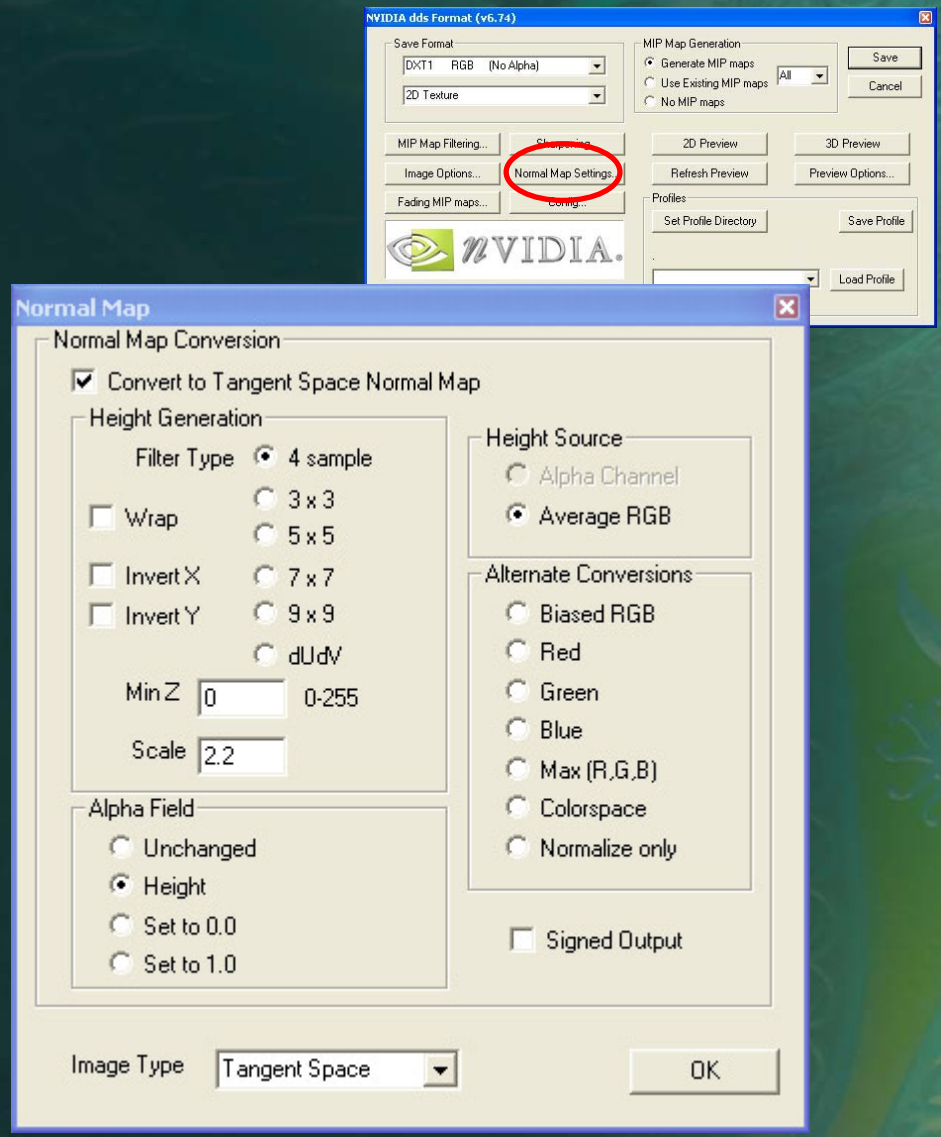
	EXPLICIT ALPHA	INTERPOLATED ALPHA
Premultiplied RGB	**DXT2**	**DXT4**
Unpremultiplied RGB	**DXT3**	**DXT5**

Special Bit Depth Choices

- **4:4:4:4, 5:5:5:1, 5:6:5** – worth the trouble?
  - For “straight” color maps, few games (that we knew offhand) using 4:4:4:4, or 5:5:5:1, or 5:6:5 formats
  - But these formats *were* being used to pack low-precision maps together for “live” recombining in shaders for terrain, sets, etc
- **FP16x4** – same as “half” in HLSL and the type used by OpenEXR (http://www.openexr.org) format used at ILM et al. Blendable and filterable on GeForce 6+
- **FP32x1, FP32x4** – IEEE 32-bit floating point. Required by GeForce 6 for Vertex Texture Fetch (VTF) – that is, texture access in *vertex* shaders for effects like true displacement mapping & dynamic terrain generation.

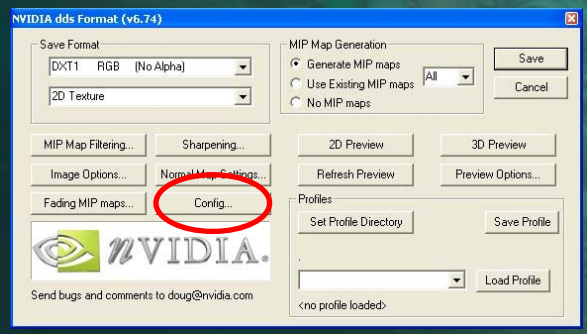
Convert/Export Normal Map

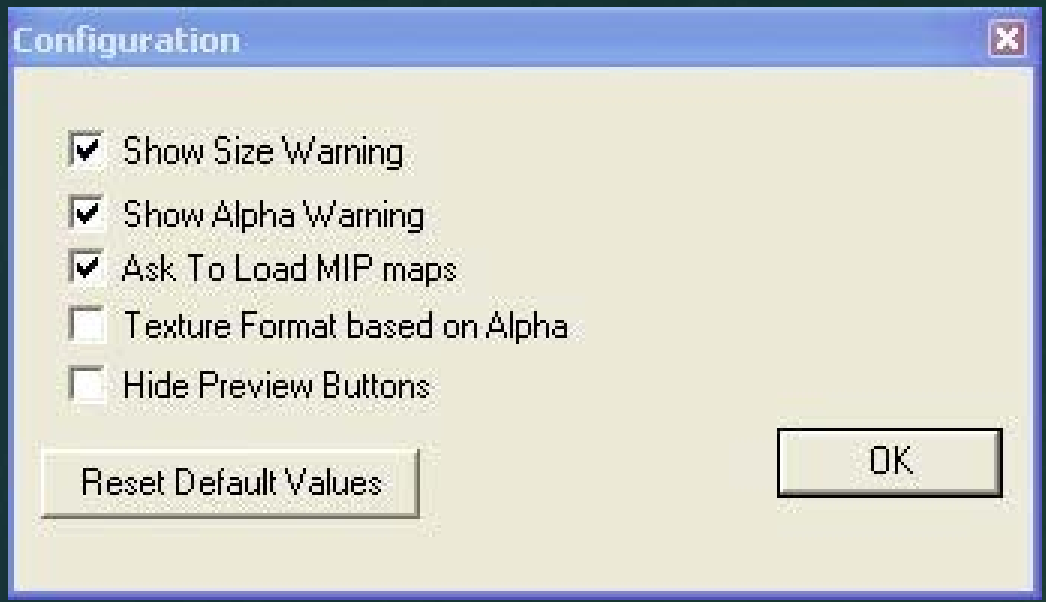
- Preferred over using the dedicated Photoshop image filter… often a valid choice! (details later)
- Hint: use good sharpening with this



Plugin Config

- These UI Choices Don’t Affect Image Quality



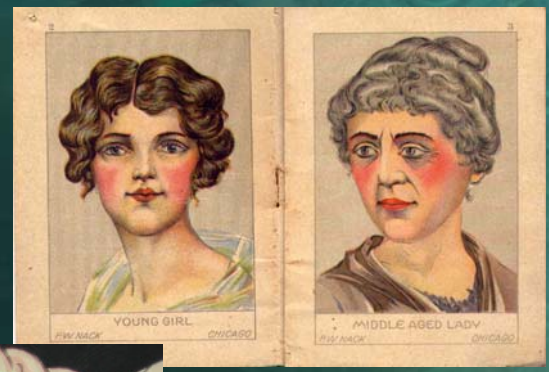


Hand-Tweaking MIP Levels

- Sometimes automatic filtering isn’t as good as hand-tweaking individual levels
- Texture painting for specific MIP-levels can accentuate the features that are most important – e.g., important symbolic elements like Flags, Eyes, Numbers & Letters, etc.
- Workflow is simplified by using the NVIDIA “Mipster.js” Photoshop script (for PC or Mac)

Stage Makeup as “MIP LOD”

- Stage makeup has to look good from 20 meters+
- It *doesn’t* have to look good up close!
- Close-up makeup is for TV, photography, and everyday life
- Stage makeup can accentuate symbolically in place of scientific realism, either overtly or subtly

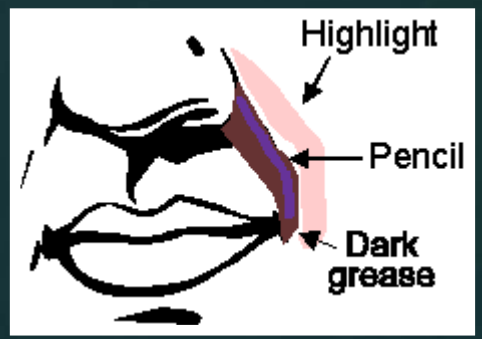


19th Century Stage Makeup Manual

Onnagata actor – makeup is “symbollically” female

“Mipping” in the Physical World

- Stage makeup looks un-natural at “normal” distances, but reads very well when scaled-down (seen from a distance)
- Accentuates the key expressive elements and geometry of the face

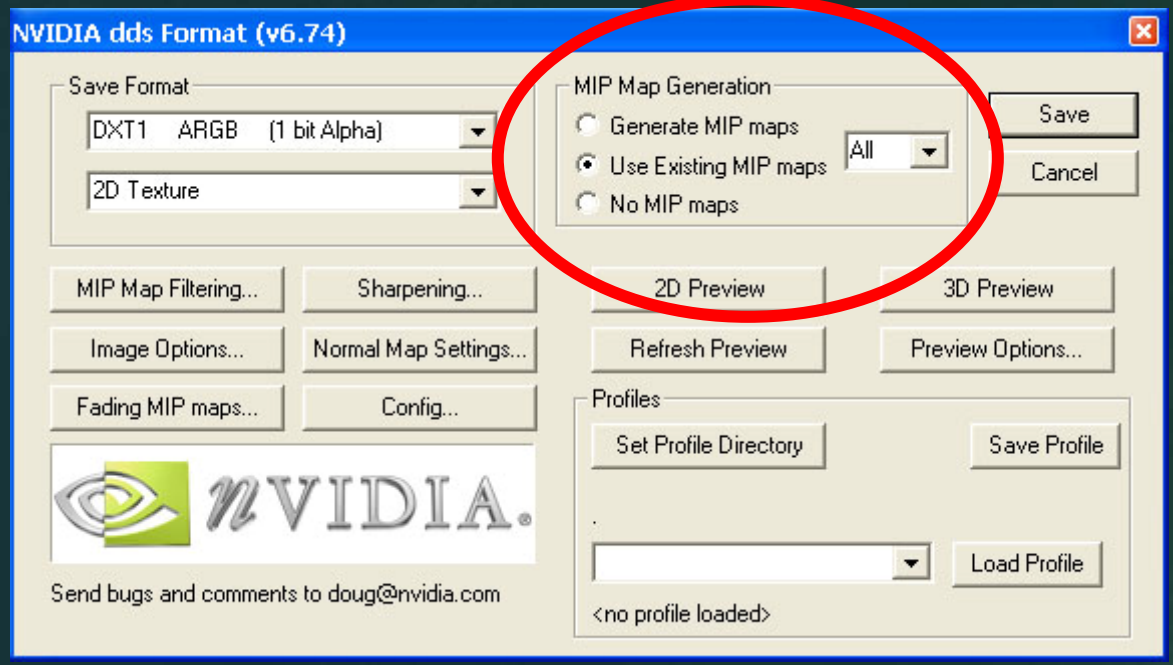


Makeup example courtesy
Tara Maginnis,
http://www.costumes.org

Using “Mipster” to Create MIPs

- Mipster is a Javascript workflow tool – install under the PS “File→Scripts” menu
- Mipster uses Photoshop image-resize to make MIP levels
- Makes a copy of your original, so it’s completely non-destructive
- Handles texture repeat in U, V, or both directions
- Each MIP gets its own Photoshop layer, to make it easier to edit/sharpen/tweak visually in Photoshop.

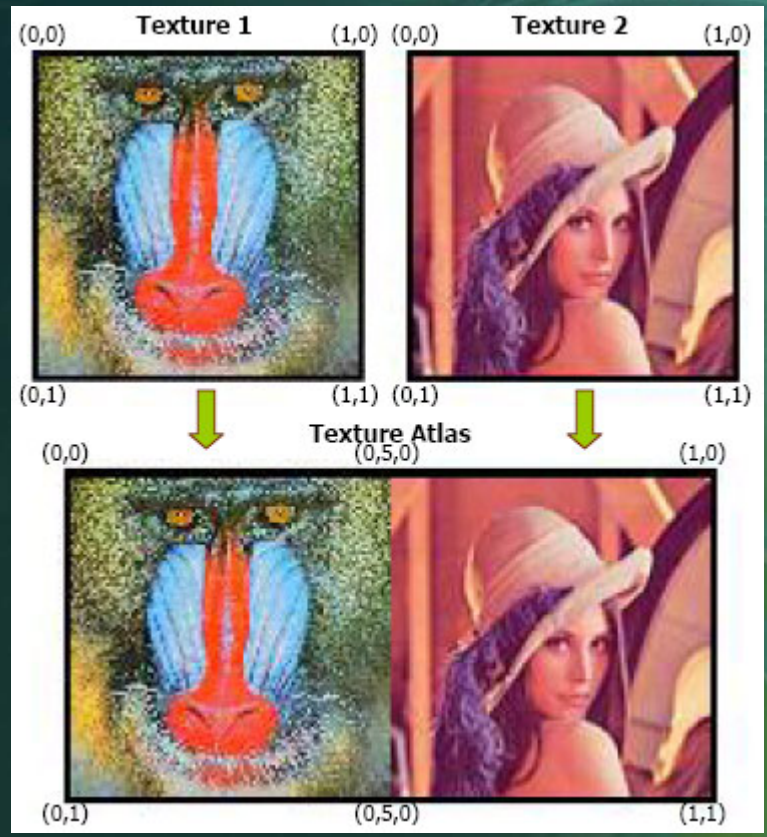
Saving Hand-Tweaked MIP Maps



- Select “Use Existing MIPs”
- All other MIP controls are *ignored*
- MIP arrangement will be figured out automatically, based on image dimensions

Rectangular Texture Atlases

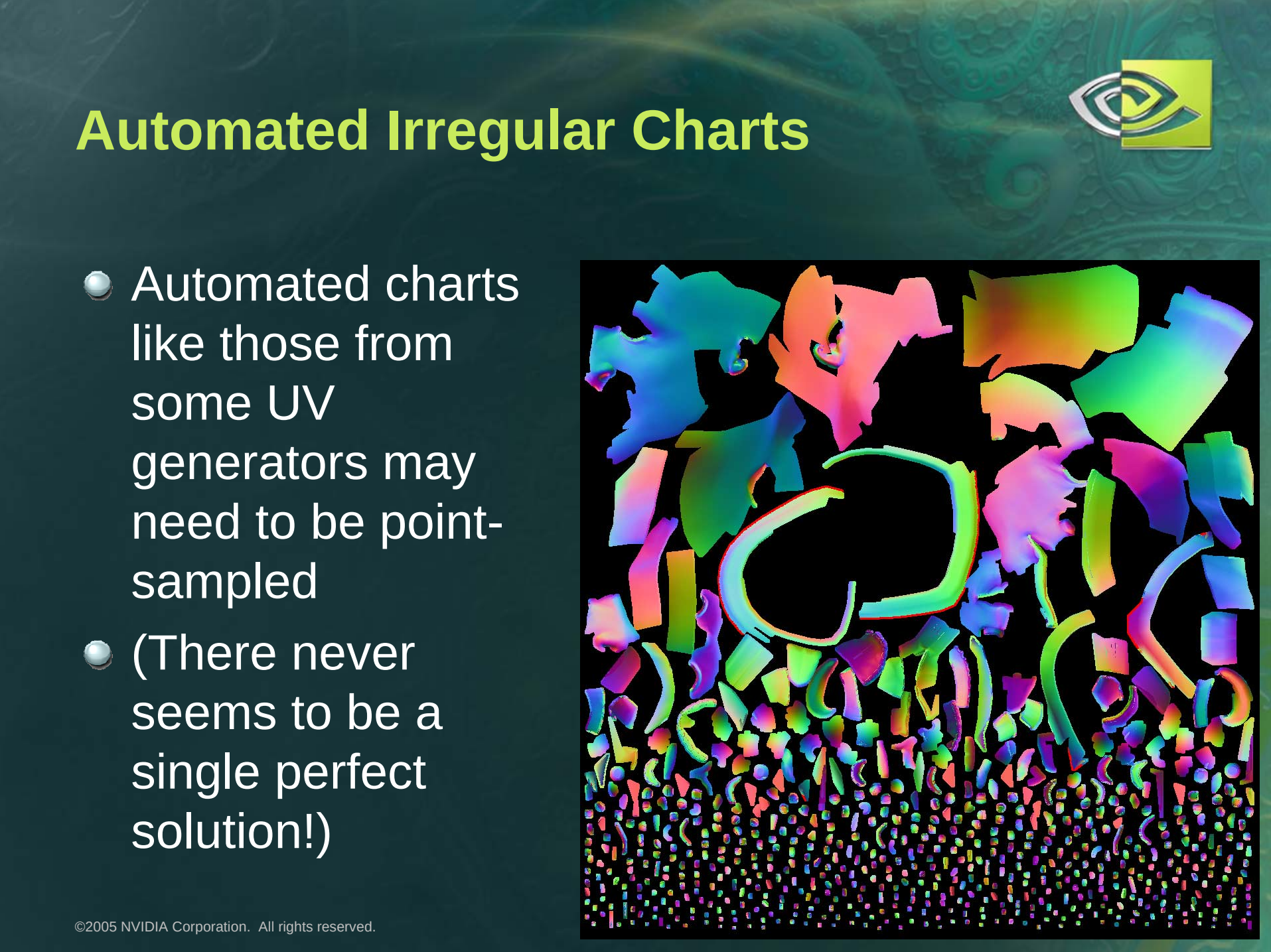
- Creation and Viewer tools on http://developer.nvidia.com/object/texture_atlas_tools.html
- Technique intended to simplify DirectX batching
- Another reason to *save the .PSD* – textures you paint today may be combined into an atlas by someone else, later



Irregular Texture Atlases

- Careful of bg colors – in this example the background is colored similar to the pieces, so that obvious problem colors won’t be introduced by MIP mapping

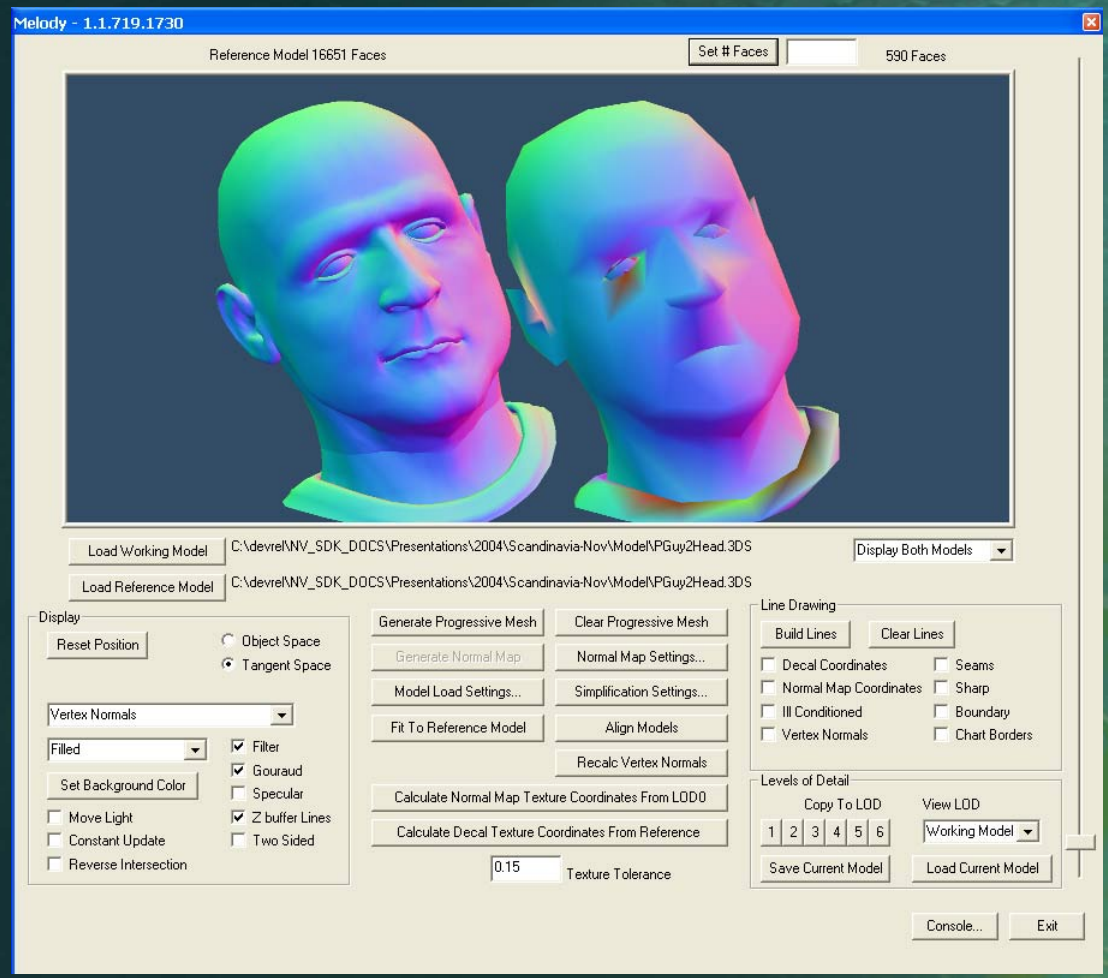
Automated Irregular Charts



- Automated charts like those from some UV generators may need to be point-sampled
- (There never seems to be a single perfect solution!)

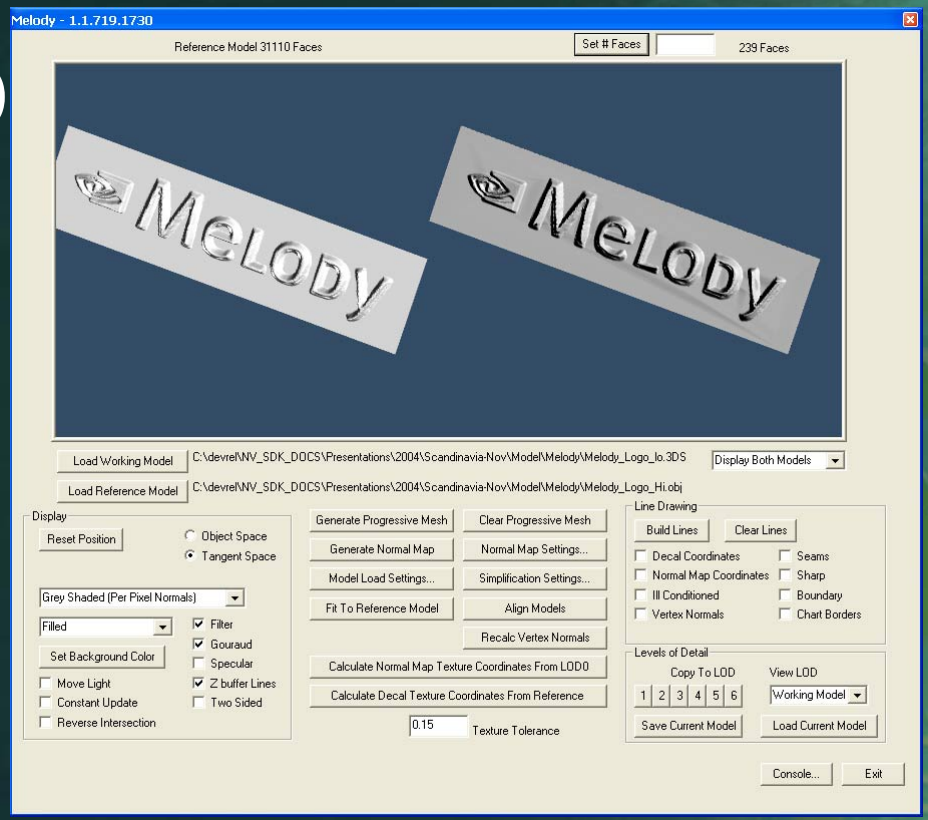
Melody and MIP maps

- MELODY can generate normal maps and assign texture coordinates while simplifying
- http://developer.nvidia.com/object/melody_home.html



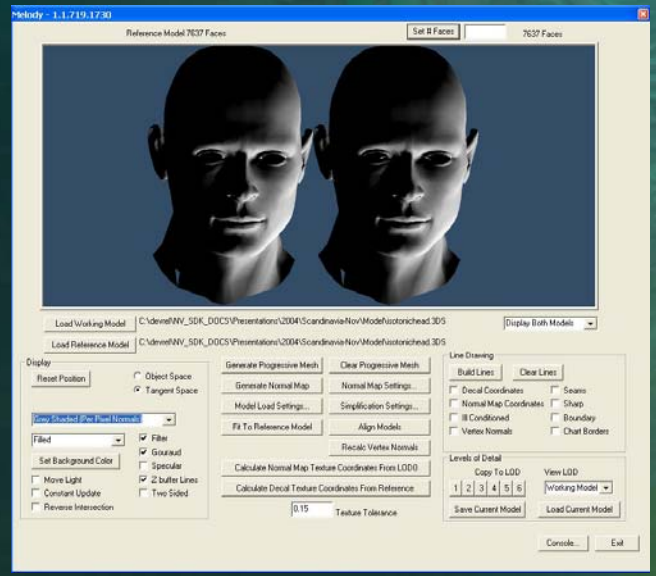
Charts are not MIP'd

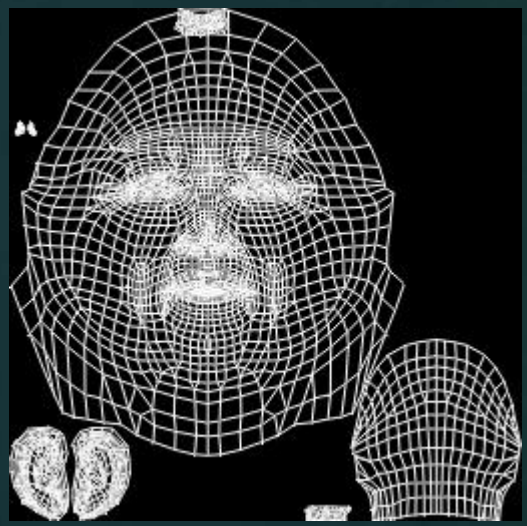
- So try to use regular decal coordinates, if they are valid & monotonic (i.e., no folding or overlapped UVs)



If Working Model is Monotonic….

- Then we will be in good mip-able shape





Decal UVs

Generated Normal Map

Making a chart atlas more mippable

- One decidedly *un*-scientific method!
- Use PS layers, duplicate the charts and use “smudge” and “gaussian blur” on the under-copies, against a gray BG

Hand-Painting Normal Maps

- One advantage of normal maps over bump maps – you can make large areas of continuing gradation (while a height field can only increase for a while before it has to go down again)
- You can tweak areas of high contrast (filter them) or low contrast (touch-up with a brush, or sharpen)
- When using ***Photoshop***, I use a texture like this as a dropper-tool palette:

Hand-Painting in *FX Composer*

- While crude, painting in FX Composer can let you see the direct 3D results in real time
- Plus, it can set the color automatically via “gesture direction” (if only Photoshop had this…)

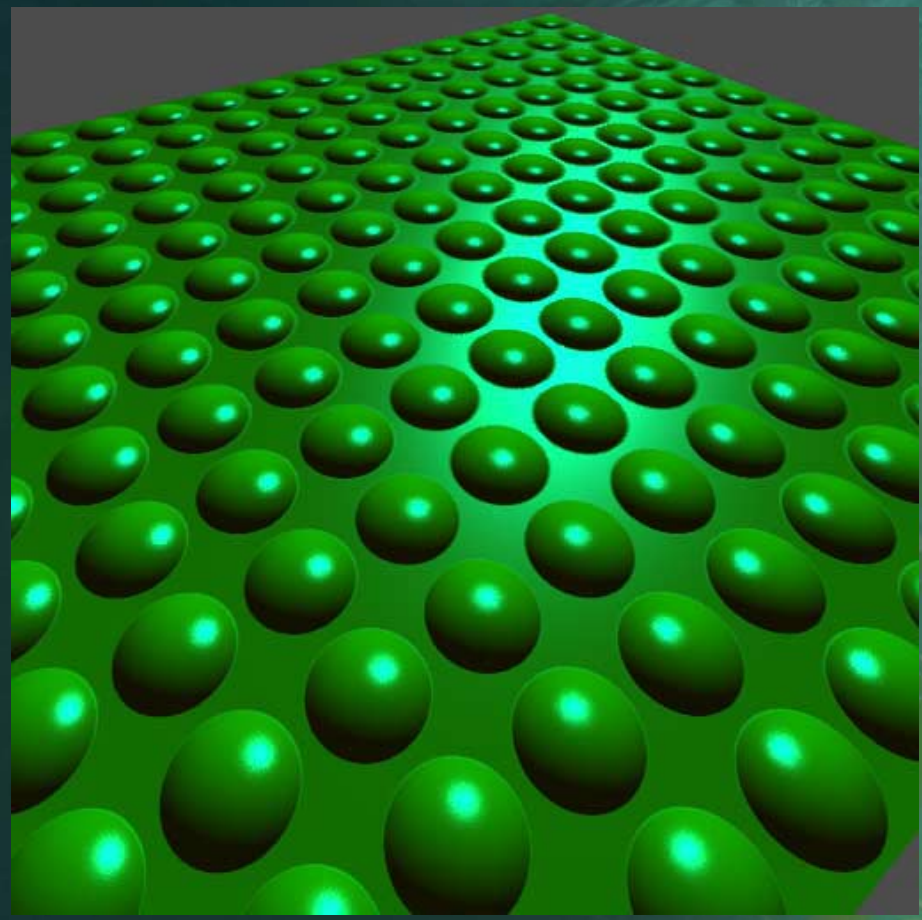
Why Normal Mapping Doesn't Work

- Well, yes, it actually does, but only up to a point!
- One of the shortcomings in bump/normal mapping is that filtering is done in the wrong part of the imaging pipeline for specular calculations
- Texture filtering is designed to deliver a high-quality, filtered color sample
- But when we bump/normal map, we are using that value as an input to *another* function – the lighting equation

Equation Alert!

- Time for gratuitous math:
- ***P*** – our final desired color
  T – Texture values
  filt() – Filtering function (usually tex2D())
  illum() – Lighting function (lit() or other shader code)
- For simple color results,
  P = filt(T)
  is what we want
- For complex, lit pixels,
  P = filt(illum(T))
  is what we want, but
  P = illum(filt(T))
  is what we get from bump/normal mapping
- *Fortunately,* in many cases the error is small enough that games (and even movies) have been getting away with it. So far.

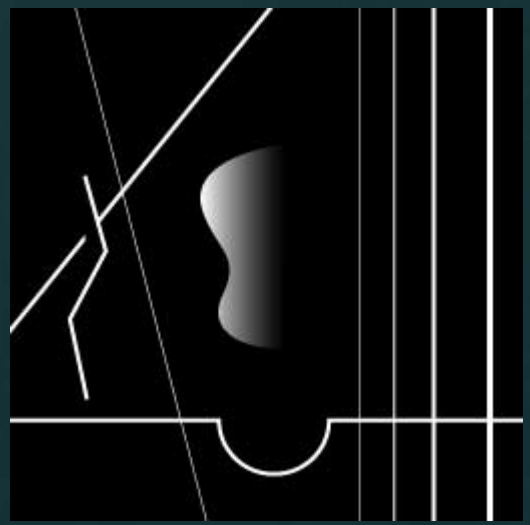
“Toksvig” Normal Maps



- Michael Toksvig of NVIDIA did the math for solving ***filt(illum(T))*** for Phong so that the pre-calculated results can be placed directly into a texture map
- Result: better-quality bump/normal mapping with good performace
- Cost: you need to choose a specific, fixed phong exponent (“specular power”) as part of the build process (artists can choose using a non-Toksvig shader, then switch for the final art assets)
- FX Composer example available
- White Paper: http://developer.nvidia.com/object/mipmapping_normal_maps.html

Normal Map Frequency

- Another reason we sometimes don't notice bump-map aliasing is that normal maps are blurrier than color maps
- They have lower *frequency* – less data per texel
- Making normal maps from height maps *is* a blurring process



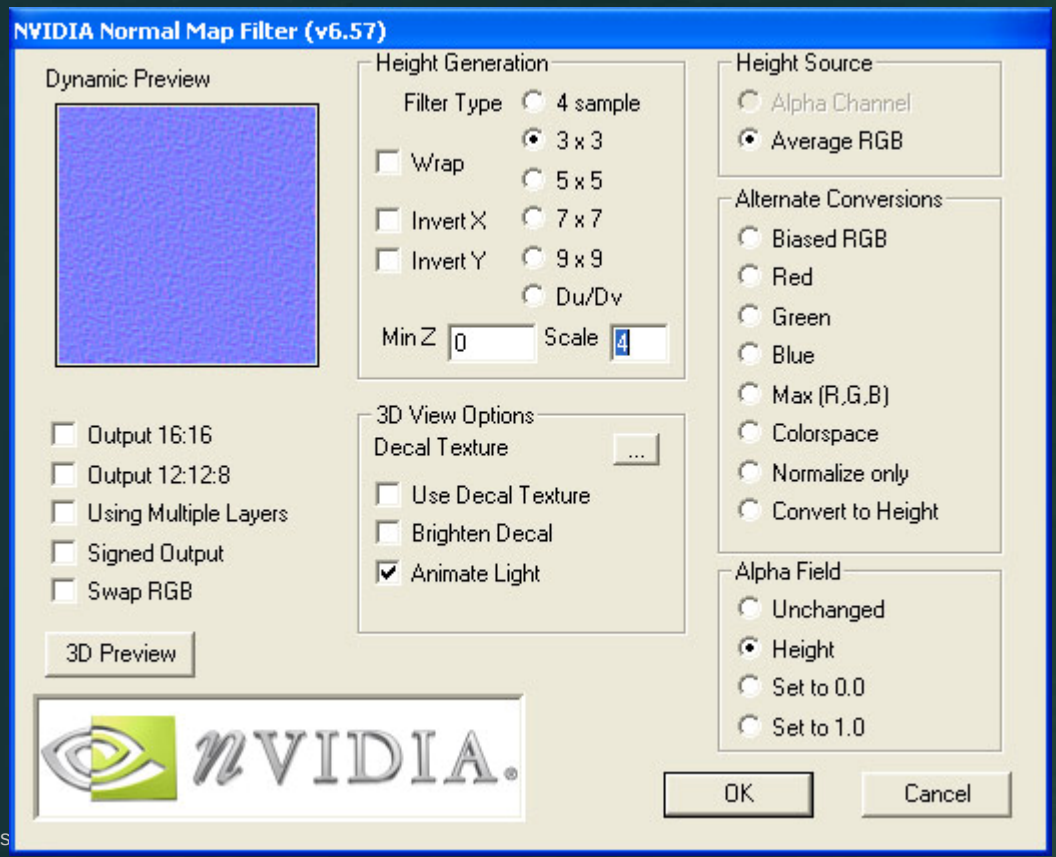
Depth Map

Normal Map

Red normal channel

PhotoShop NormalMap Filter

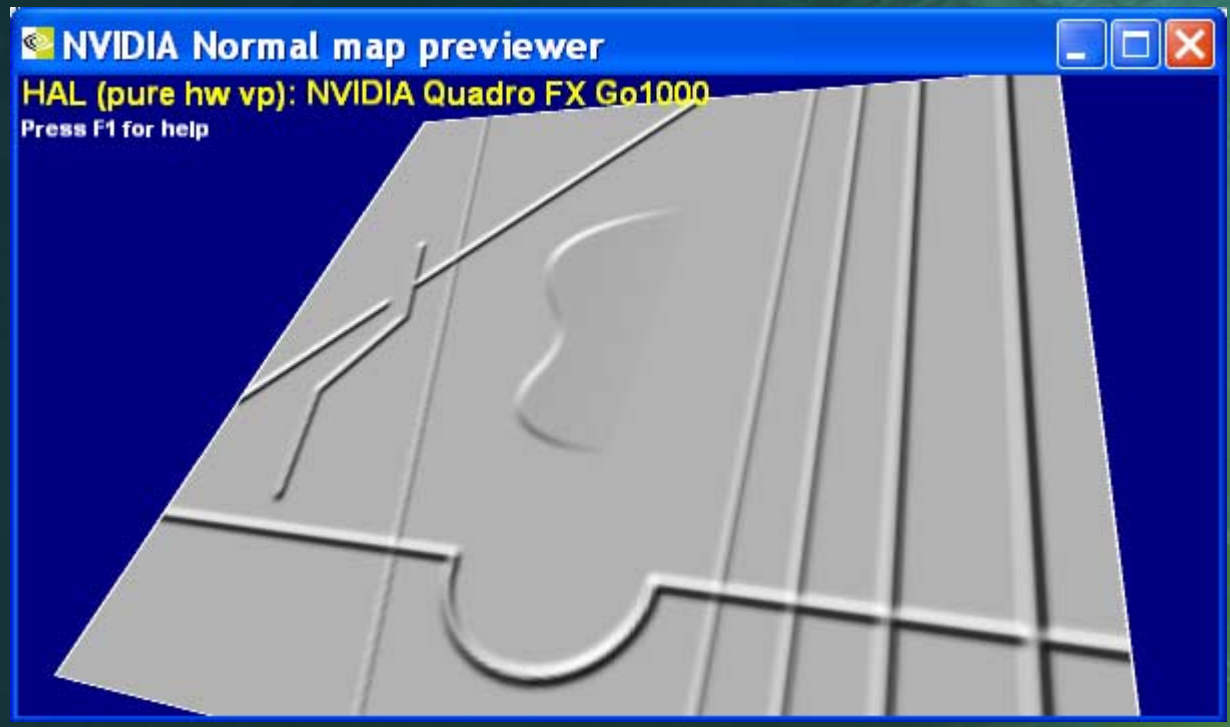
- http://developer.nvidia.com/object/photoshop_dds_plugins.html
- Filters->nvTools->NormalMapFilter…



Choices in Bump Conversion

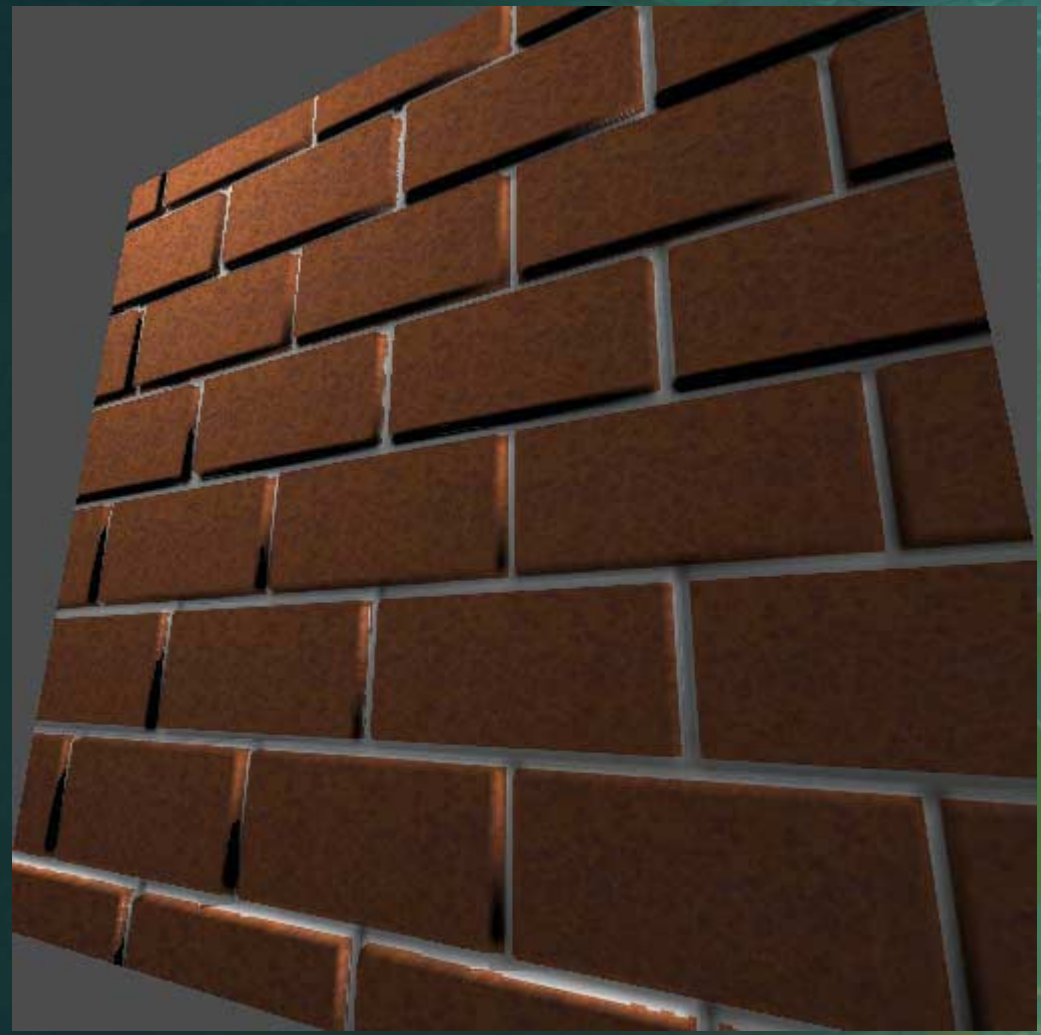
- Gray→Normals→Mips - *Wrong*
  - This is the “naïve” method – often good enough but not optimal
  - When you scale down MIPs that are already turned into normal maps, you are scaling-down the size of the normals filter – small sizes will smudge and sparkle
- Gray→Mips→Normals - *Right*
  - This is the method used by Mipster and by the DDS plugin when saving normal maps, and is usually preferred
  - Normals filter will be the same size for *all* screen sizes, and mid-workflow sharpening can be applied to the grayscale MIPs to keep them looking crisp
- The Normals-Map Filter is good for previewing and has one important feature that I especially like:

Normal Map Preview



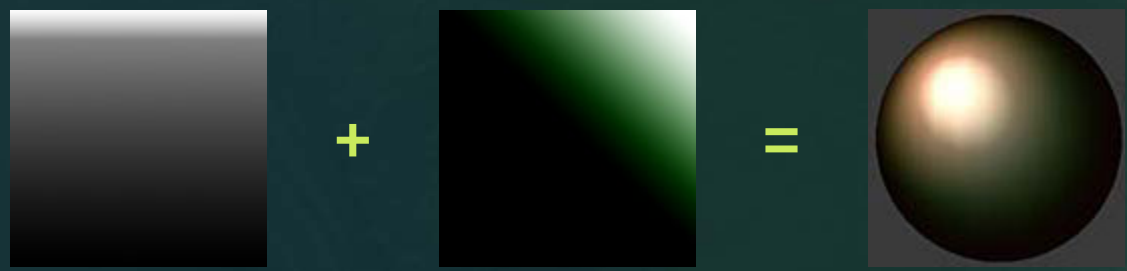
- Key to getting a result that looks good for your data
- You can bring in a matching RGB texture too
- Use “animating light”

Parallax and Displacement Mapping



- Another shortcoming of bump/normal mapping is the lack of perspective changes for large bumps
- Parallax mapping, Relief Mapping, and “displace” mapping can help solve these problems
- None are “real” displacement – note the straight-line edges in the example pic
- Both need to have a normal map *and* a valid height map

Painting BRDFs



- “BRDF” means describing the way a particular material reacts to light as a math function – complex functions can be reduced to combinations of textures
- MIP mapping on BRDF textures can help suppress aliasing even while making complex BRDFs run really fast

Painting BRDFs

- Viewing the results on a live shader is best – FX Composer, CgFX Viewer, etc. There are various possible shaders, each with their own input styles
- You can start from a real-world BRDF, and tweak according to what you want!
- The NVIDIA “Time Machine” truck paint BRDF started from Ford Motor Co “Mystique” paint, but was repainted to look more “1950’s”

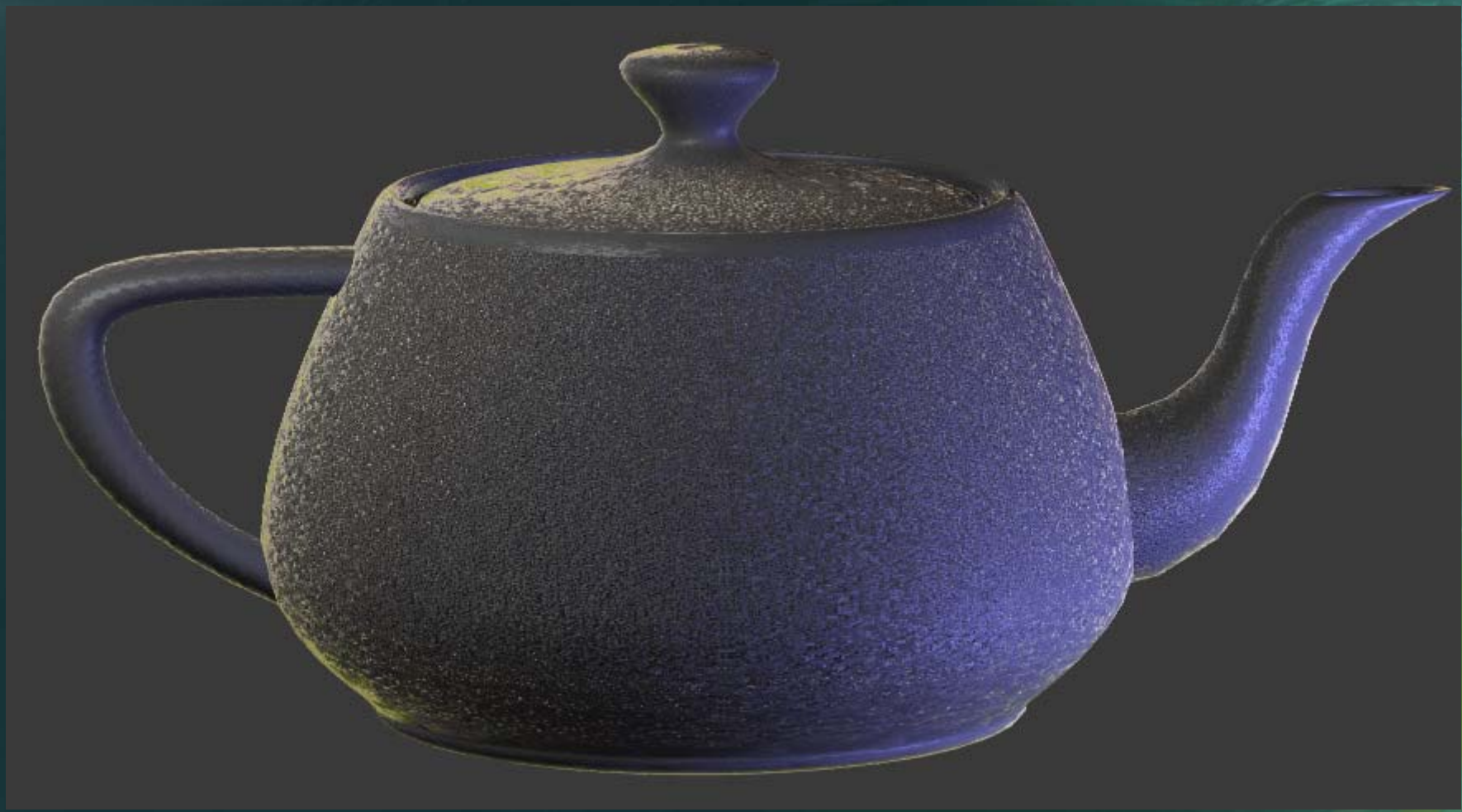
Texturing with Unlit Textures

- Getting rid of shadows
- But not all?
- If an object is black, it will never be lightened by lighting! So be aware of the difference between truly black and merely dark
- If you know an area will truly be ALWAYS black, go ahead and paint it that way – it will save trouble later (e.g., the insides of a character's nostrils probably need never be lit)
- Try to preview with a reasonable version of the shader and lighting ASAP
- *Keep the .PSD files*

Common Problem: MIPS gone bad…

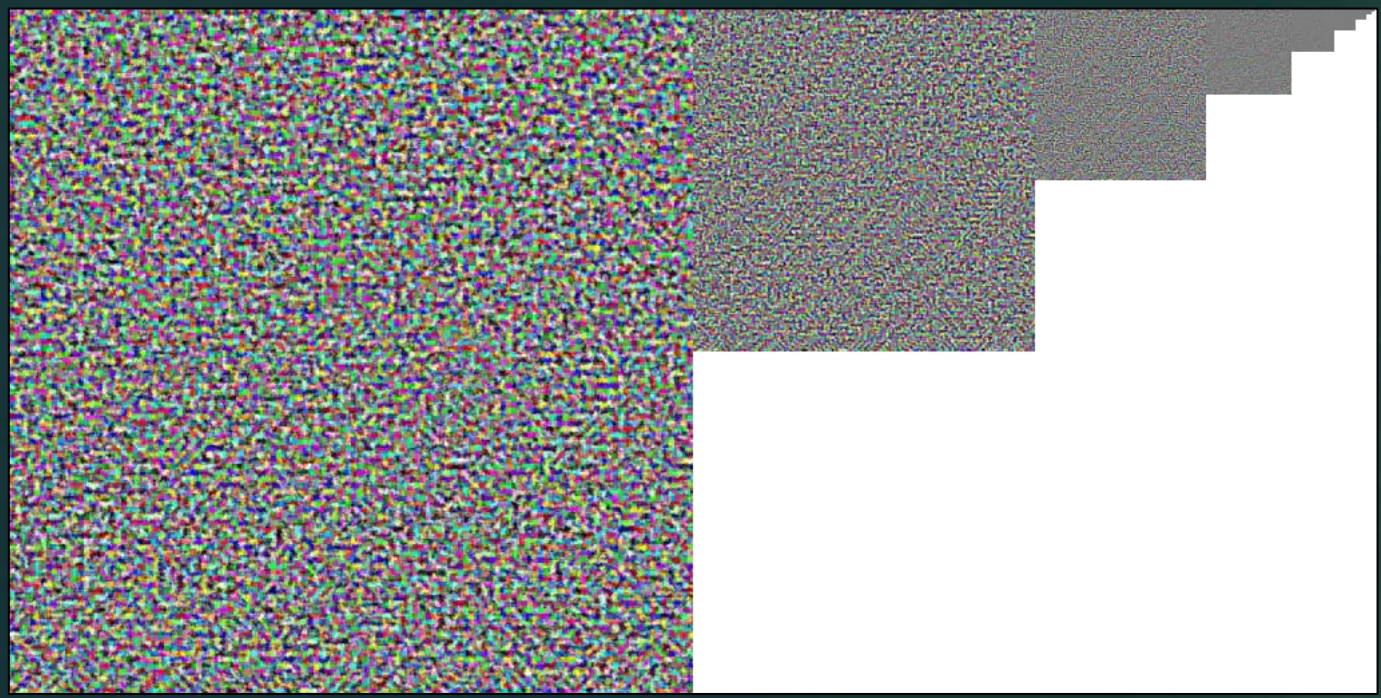
- ..or good?
- Compare these two pots.
- They were made with the SAME shader
- One image is shown reduced in size to match the other

Here is the original large render

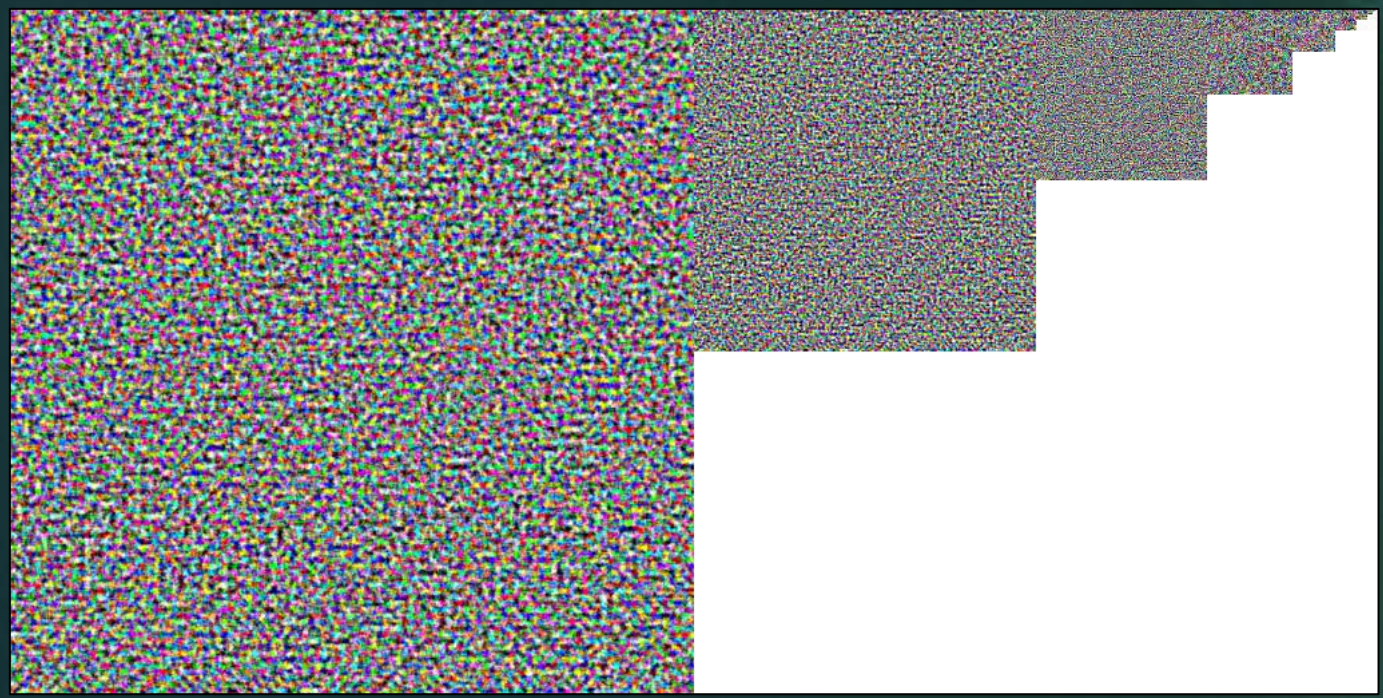


What happened to the bumps?

- MIP mapping from Photoshop has reduced the noisy bump texture to a featureless gray value
- At low MIPs, the bumpiness disappears

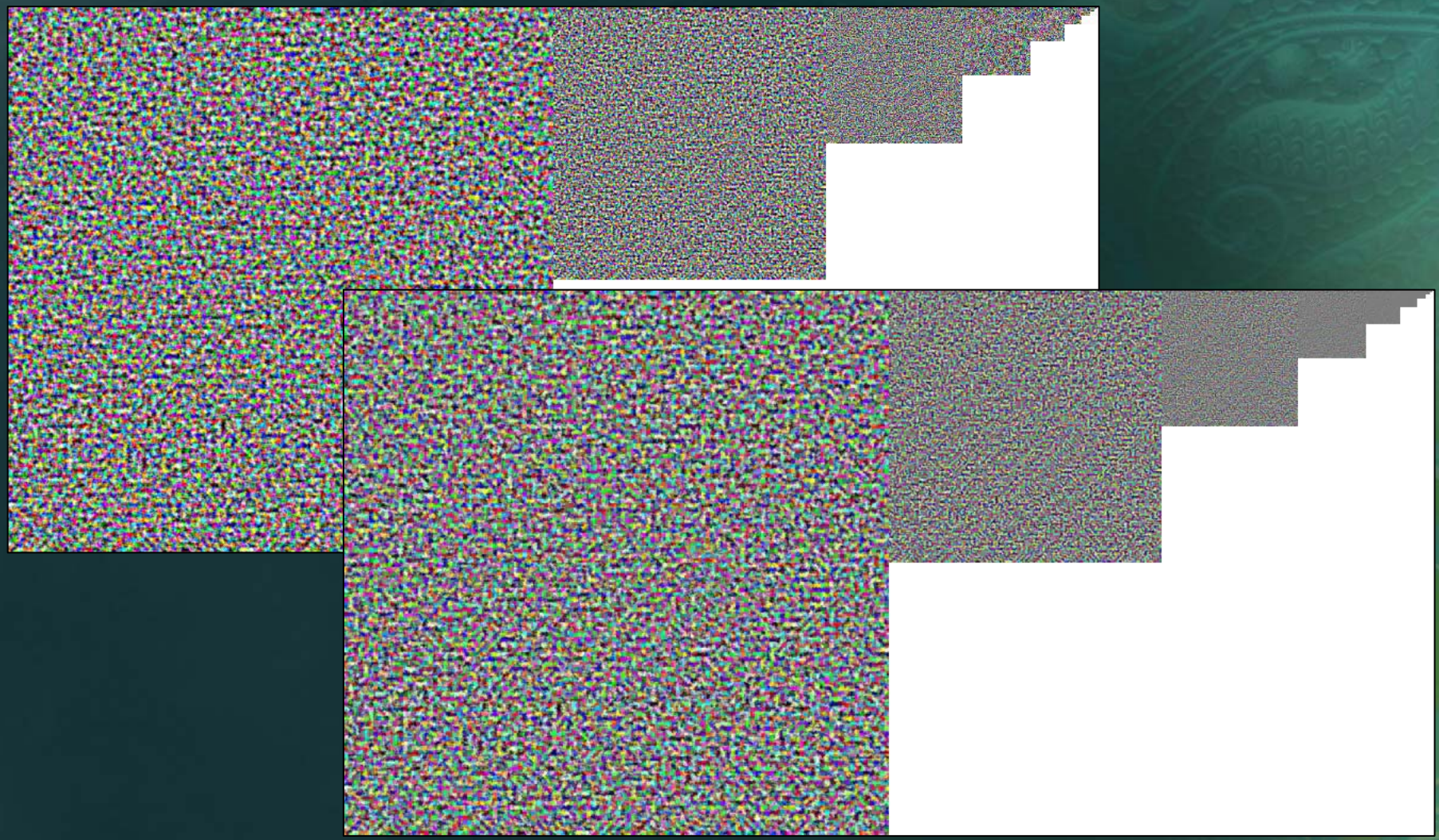


Compare to Procedural MIPs

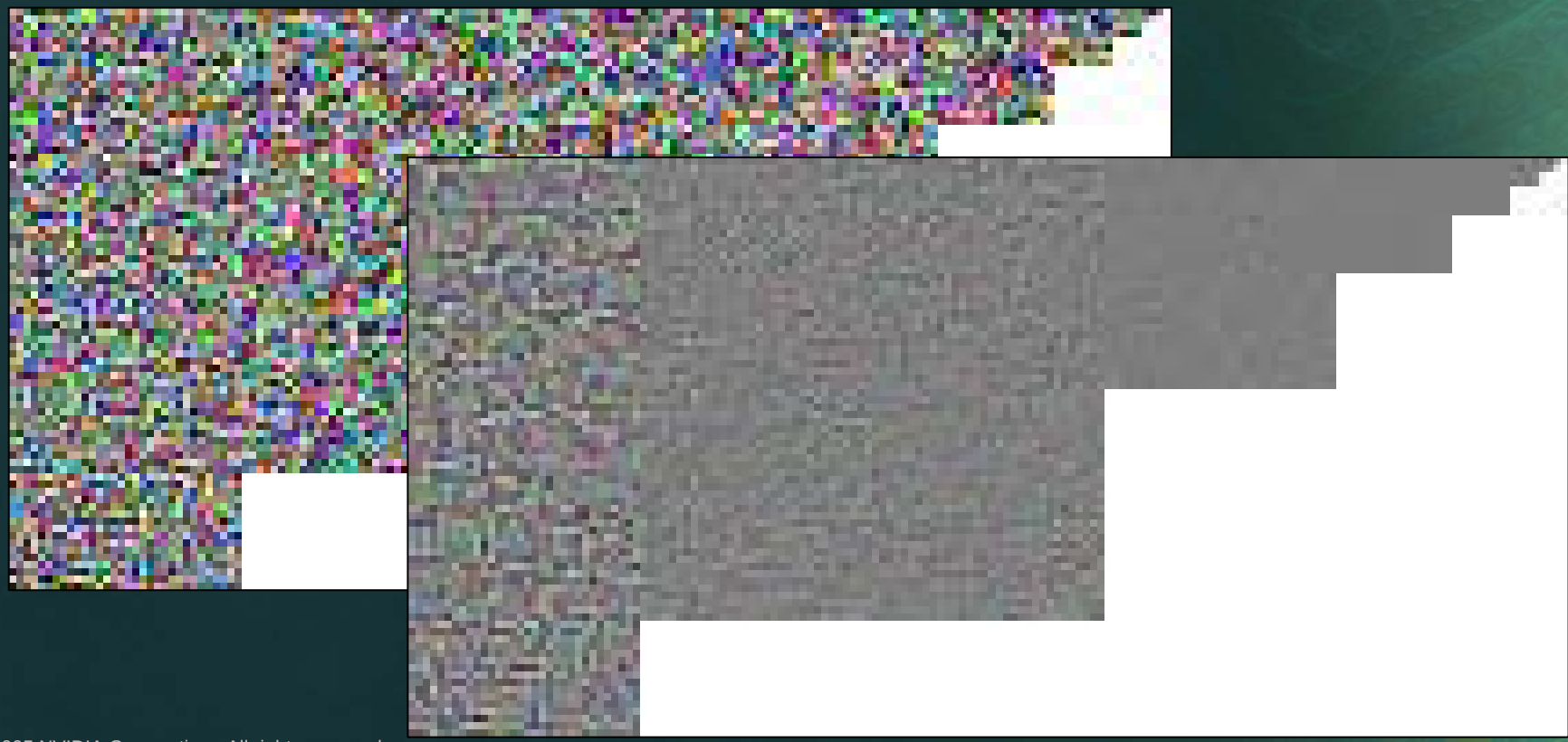


- These MIP maps were created deliberately, rather than as scaled copies of the top MIP

Both Versions for Comparison

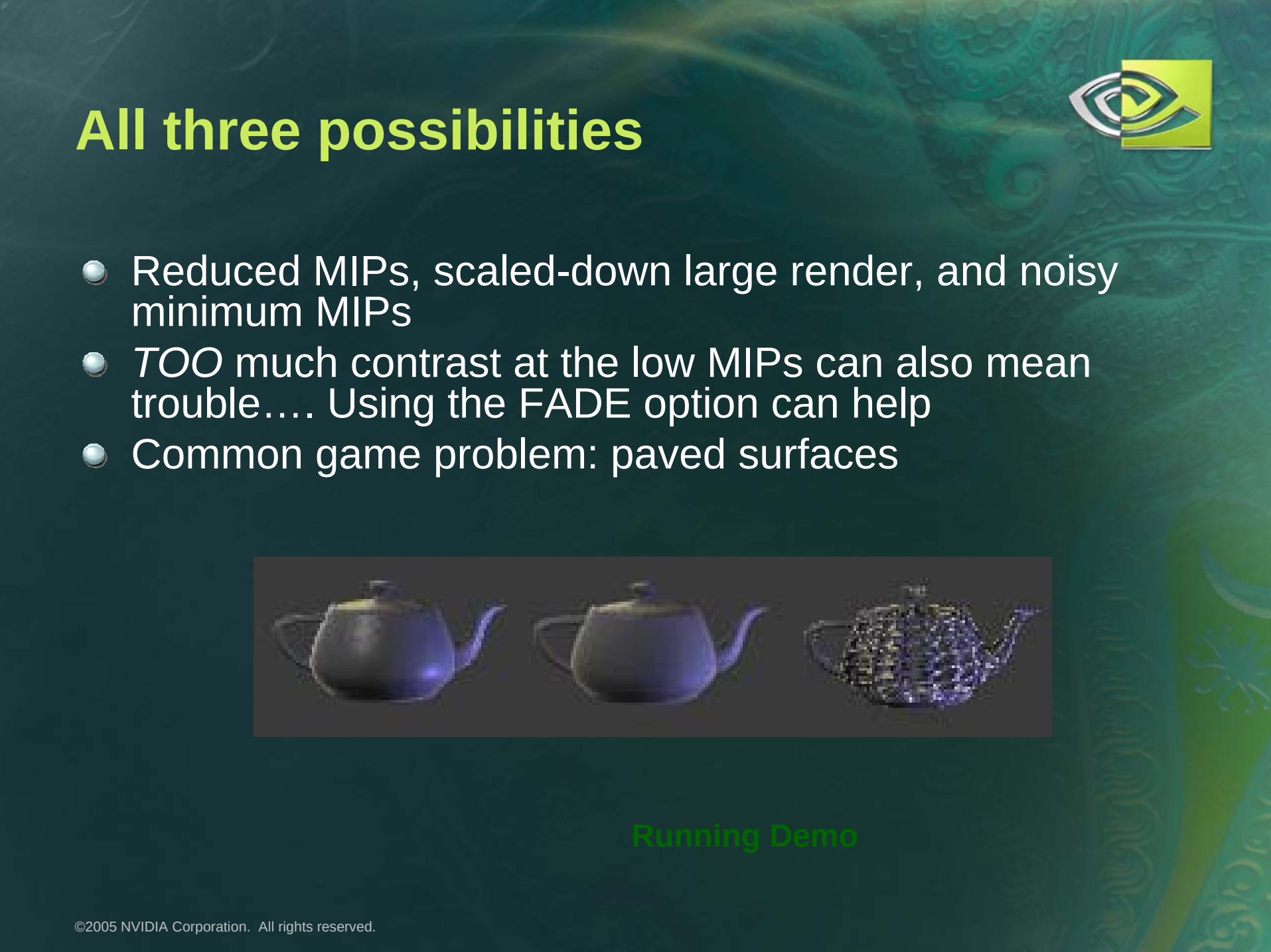


Close-up of Lowest Levels



- Same *overall* color, but the detailed MIP has more contrast

All three possibilities



- Reduced MIPs, scaled-down large render, and noisy minimum MIPs
- *TOO* much contrast at the low MIPs can also mean trouble…. Using the FADE option can help
- Common game problem: paved surfaces

Running Demo

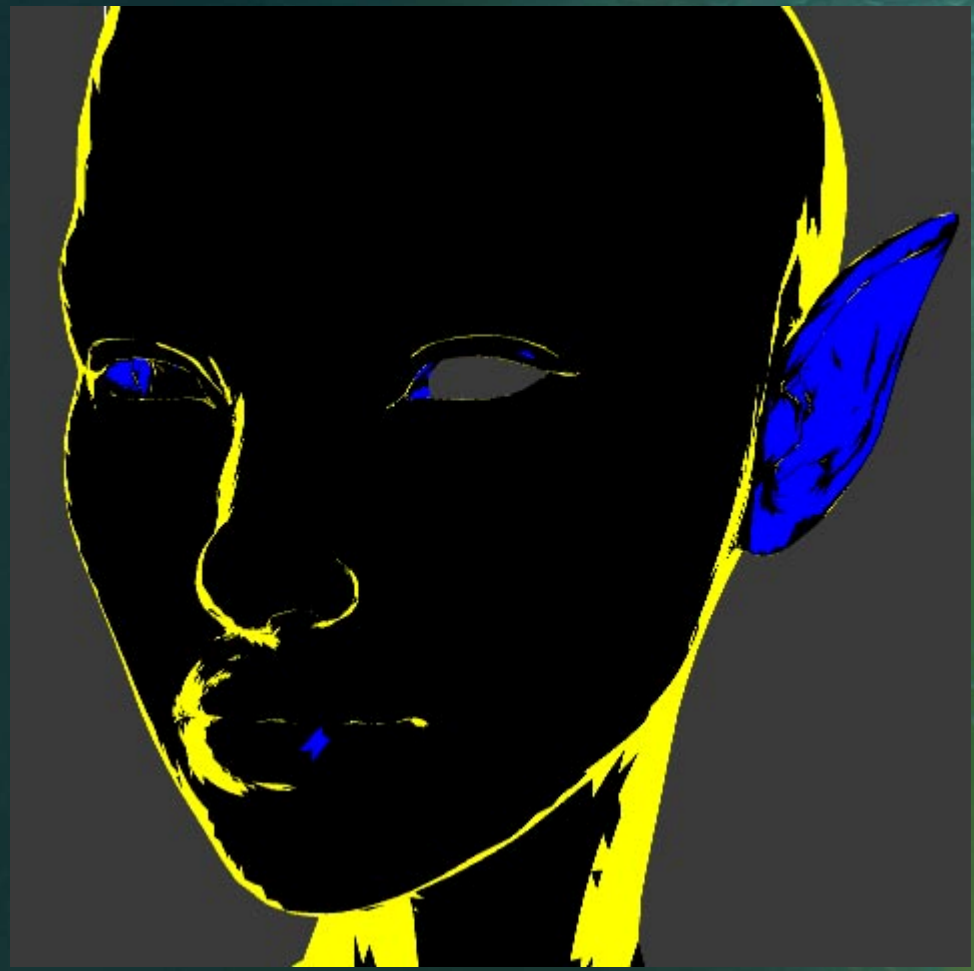
How Big Does it *Really* Need to Be?

- Sample images at varying reso's. How big will players see this model 90% of the time? 100 pixels? 12? 500?
- The problem of varying player screen sizes (including handheld/SD/HD consoles)
- Programmers: See Iain Cantley's chapter on dynamic MIP mgmt in *GPU Gems 2*

Makeup example courtesy Tara Maginnis, http://www.costumes.org

Knowing What Resolution to Use

- “uvDetective.fx” in NVIDIA SDK and FX Composer
- Black = selected reso
- Blue = *could* use higher reso
- Red, Green, Yellow = lower-res okay

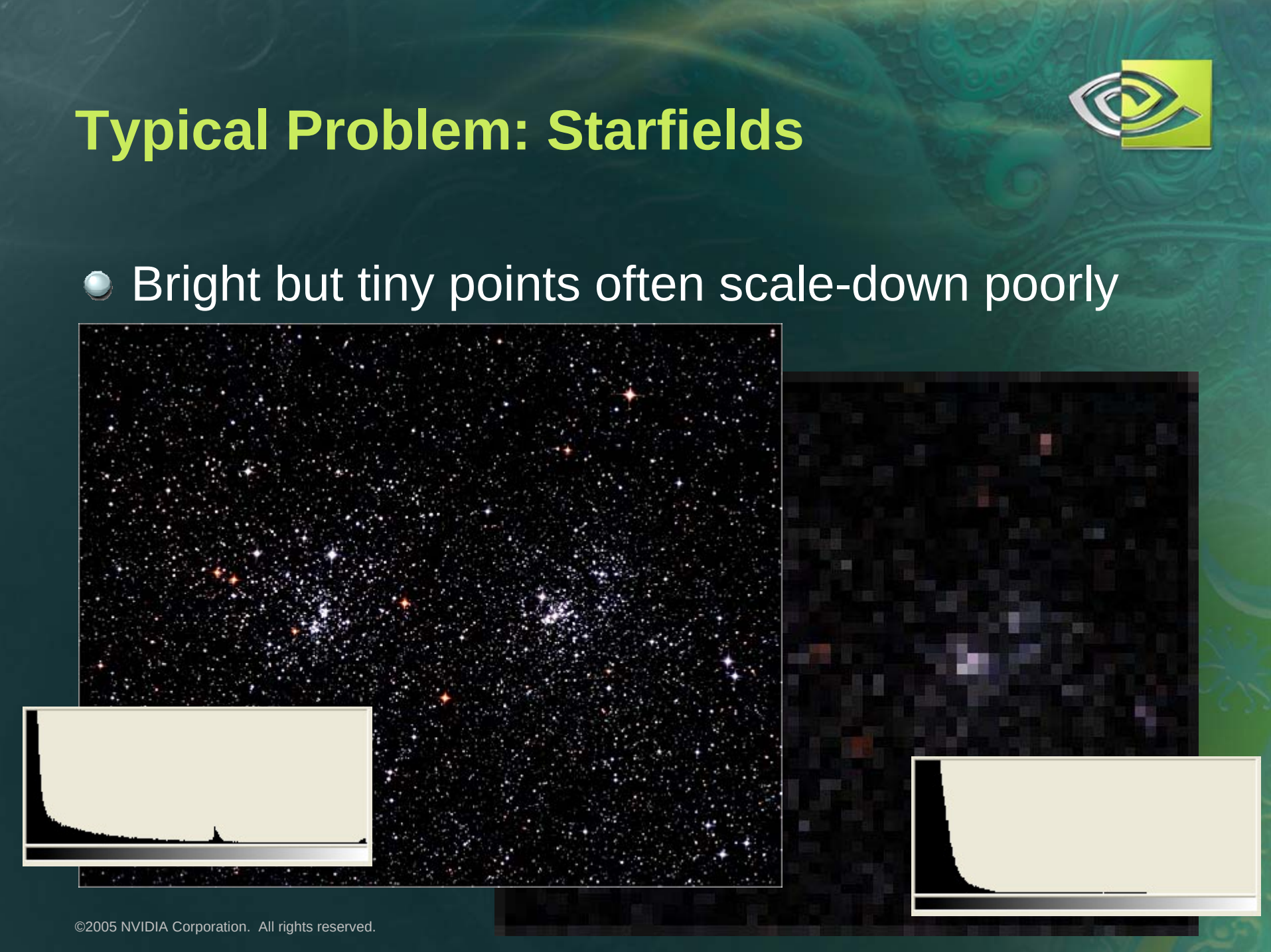


Running Demo

Dynamic Range Problems

- Point light, e.g.:
  - Stars
  - Office buildings
  - Landing lights

Typical Problem: Starfields



- Bright but tiny points often scale-down poorly

Why Starfields Can Look Bad

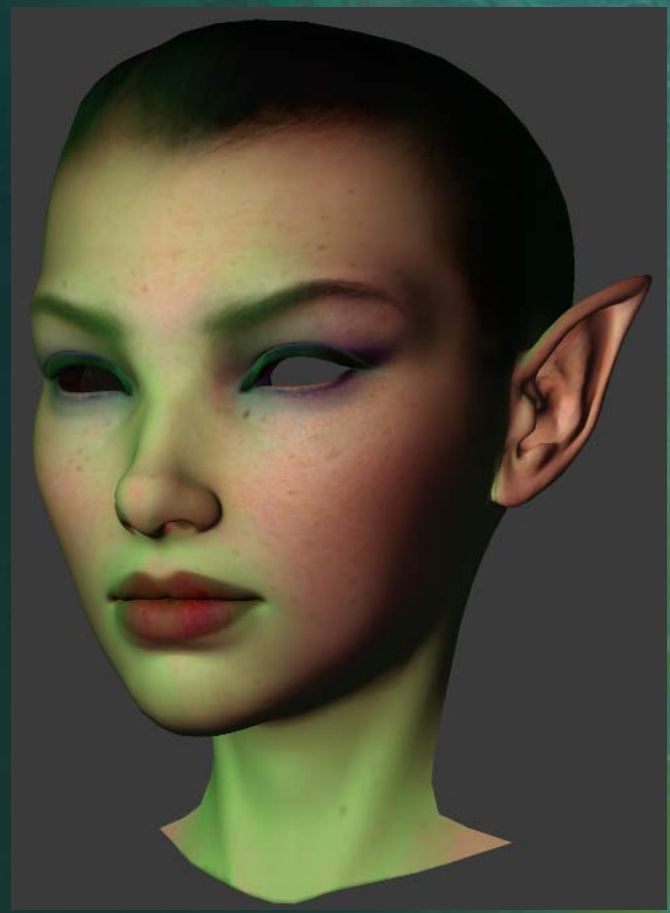
- Stars (and other points) are very tiny but brighter than a “white” pixel
- Filtering them after being clipped to 8-bit reduces their genuine contribution
- HDR corrects for this, but not always available
- Using non-linear MIP filters can also help
- Common example: Hi-res Sci-fi movies look rotten on low-res TV, the stars disppear!

Original points

Down-sampling

Result

Questions????



(“Poor Man’s Fresnel” – fade low mip levels to green)

- kbjorke@nvidia.com
- http://developer.nvidia.com/