



*n*VIDIA™

Texture Space Bump Maps

Sim Dietrich

sim.dietrich@nvidia.com

Bump Mapping Overview

- Bump mapping enables per-pixel detail without needing to use per-pixel-sized triangles
- It is actually a lighting calculation, either diffuse, specular or both
- Because it is a lighting calculation, it is essentially simply a dot product
- Some bump mapping techniques don't use dot products
 - **EMBM** uses lookups into 2D textures based on dU and dV per pixel
 - **Embossing** uses lookups into 2D textures based on dU and dV per vertex
- They are really 2D operations, and can't correctly handle arbitrary geometry very easily



NVIDIA

DOT3 Bump Mapping

- To perform correct bump mapping or per-pixel lighting, we need to perform a true 3 dimensional dot product per-pixel between two 3-dimensional vectors
 - One representing the light vector or half-angle
 - The other representing the per-pixel surface normal
- To generate the correct result, both vectors must be defined in the same coordinate space
 - View space
 - World space
 - Model space
 - Any space, as long as both vectors are expressed in it



NVIDIA

Texture Space

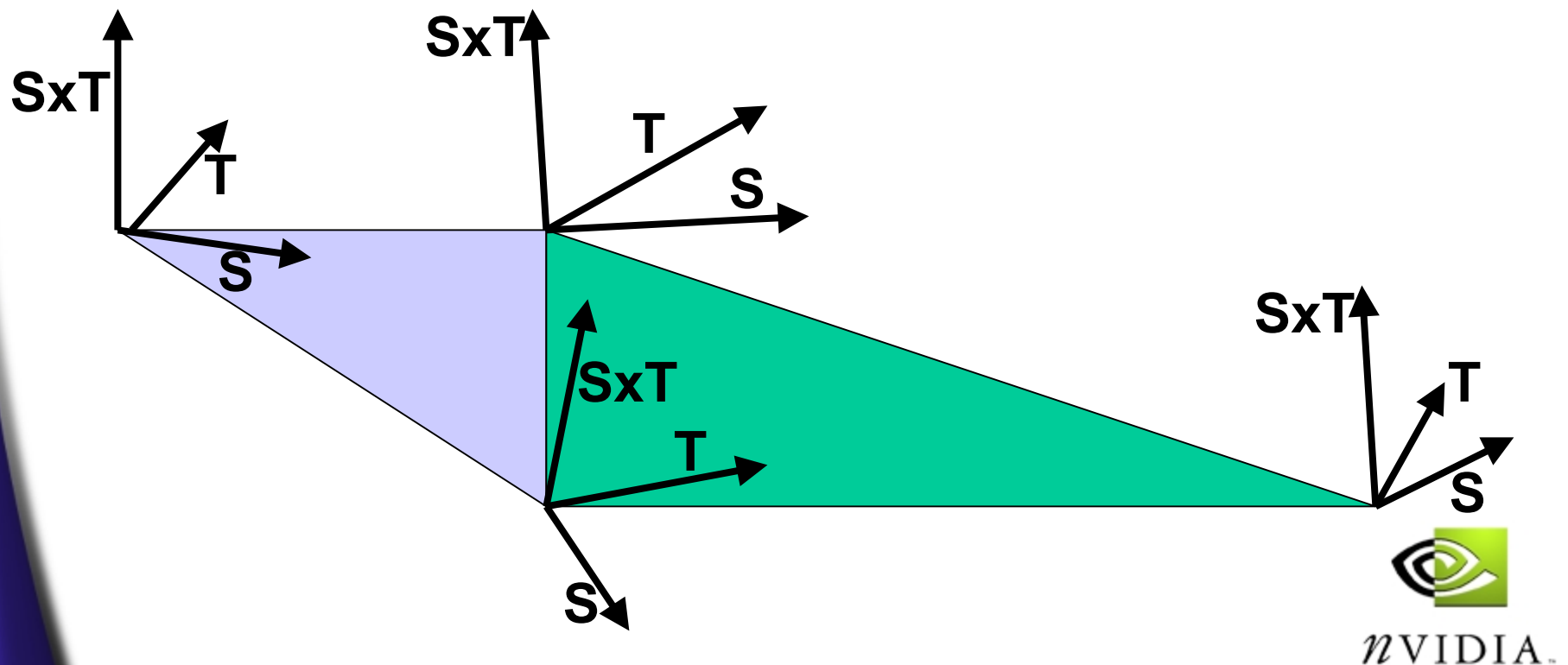
- One convenient space for per-pixel lighting operations is called Texture Space
- It represents a local coordinate system defined at each vertex of a set of geometry
- You can think of the Z axis of Texture Space to be roughly parallel to the vertex normal
- The X and Y axes are perpendicular to the Z axis and can be arbitrarily oriented around the Z axis



NVIDIA

Why Texture Space

- Texture Space gives us a way to redefine the lighting coordinate system on a per-vertex basis



Why Texture Space?

- If we used model or world space bump maps, we would have to regenerate the entire bump map every time the object morphed or rotated in any way, because the bump map normals will no longer be pointing in the correct direction in model or world space
- We would have to recompute the bump maps for each instance of each changing object each frame – no thanks



NVIDIA™

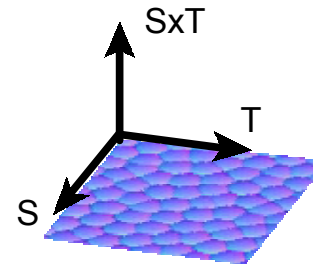
Why Texture Space?

- **Texture Space is a surface-local basis in which we define our normal maps**
- **Therefore, we must rotate the light into this space as well**
- **This means we must move the light vectors into Texture Space before performing the per-pixel dot product**



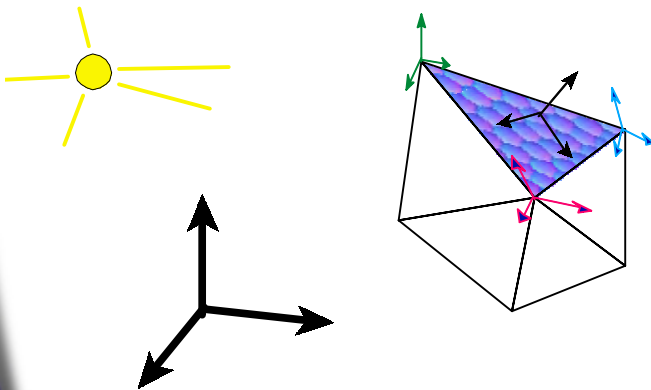
nVidia

Texture Space Diagram



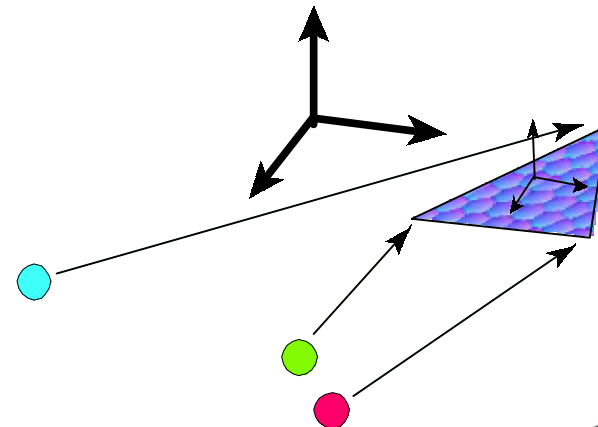
Normal Map – A flat plane in S,T direction

Normal map applied to object
RGB are in the wrong space!!



Solution = Rotate Light position into
S,T,SxT space.

Result: New light position for each vertex.



nVIDIA™

How to Author for Texture Space

- **The best method for generating Texture Space for your geometry is as follows :**
 - **Have the artist apply bump maps in their authoring tool – or just use the same mapping as the decal**
 - **Don't let them use texture mirroring**
 - **Don't use degenerate projections (ie stretched textures)**
 - **When loading in a model, create an extra set of 3 3D vectors per vertex**
 - **These will store the axes of the Texture Space basis**
 - **Generate the Texture Space vectors from the vertex positions and bump map texture coordinates**



NVIDIA

How to Generate Texture Space?

- For each triangle in the model :
 - Use the x,y,z position and the s,t bump map texture coordinates
 - Create plane equations of the form :
 - $Ax + Bs + Ct + D = 0$
 - $Ay + Bs + Ct + D = 0$
 - $Az + Bs + Ct + D = 0$
- Solve for the texture gradients dsdx, dsdy, dsdz, etc.



NVIDIA[™]

Generating Texture Space

- Now treat the $dsdx$, $dsdy$, and $dsdz$ as a 3D vector representing the S axis $\langle dsdx, dsdy, dsdz \rangle$
- Do the same to generate the T axis
- Now cross the two to generate the SxT axis – this is the ‘Z’ or up axis of Texture Space, and is typically close to parallel with the triangle’s normal
- If your SxT and the triangle normal point in opposite directions, the artist applied the texture backwards – have the artist fix this, or negate the SxT axis



NVIDIA[™]

Generating Texture Space

- These 3 Axes together make up a 3x3 rotation/scale matrix

dsdx dtdx SxTx

dsdy dtdy SxTy

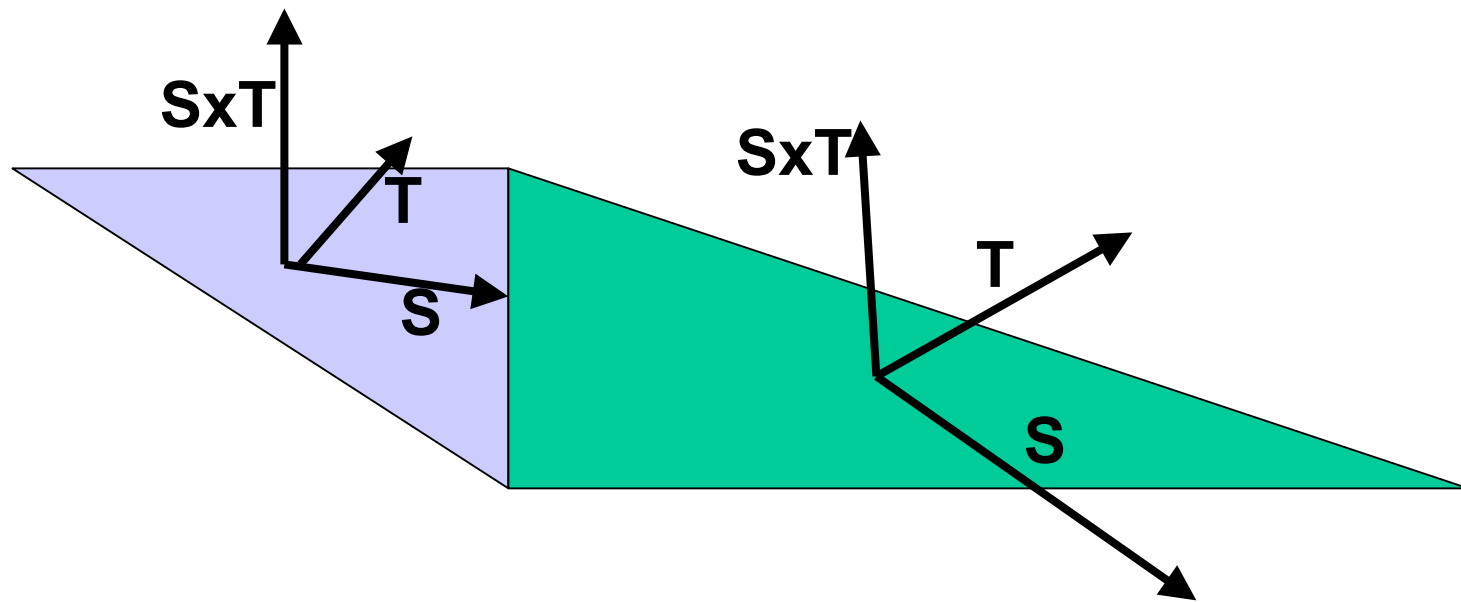
dsdz dtdz SxTz

Putting an XYZ model-space vector through this 3x3 matrix produces a vector expressed in local Texture Space



nVIDIA™

Per-Triangle Bases



nVIDIA™

Per-Triangle Bases

- We now have a coordinate basis for each triangle
- We need them on a per-vertex basis so they can vary smoothly across our geometry
- The solution :
 - For each vertex, sum up the S vectors from each face that shares this vertex.
 - Do the same for all T and SxT vectors
 - Normalize each sum vector
 - Optionally scale by the average original magnitude of S,T or SxT if your texture map is applied anisotropically
 - The result is per-vertex Texture Space
 - This is analogous to calculating vertex normals for lighting



NVIDIA[®]

Per-Vertex Texture Space

- Now we have what we need to move a light into a local space defined at each vertex via the Texture Space Basis Matrix
- For each per-pixel light, we move it's L or H vector into local Texture Space
 - On the CPU with C code
 - Or on the GPU with a vertex program



NVIDIA™

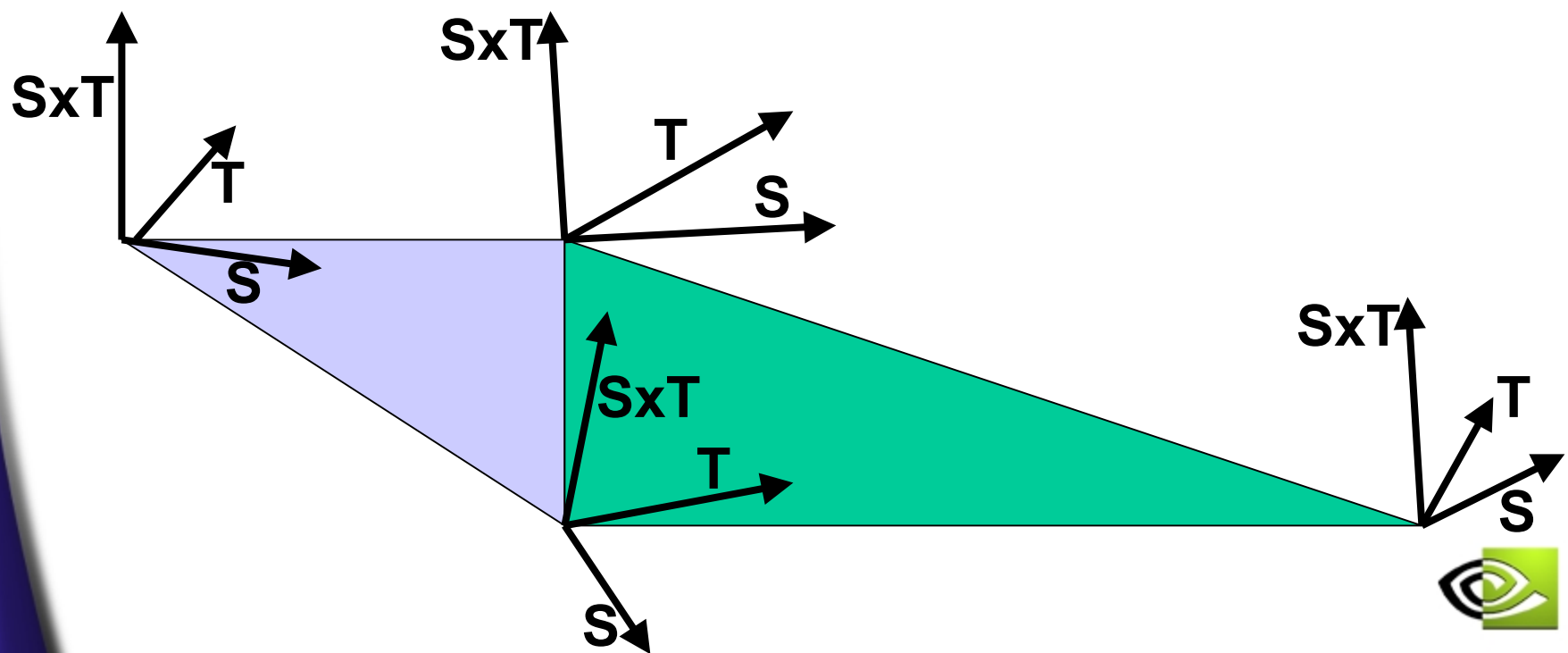
Texture Space In Practice

- The L or H vector is linearly interpolated across the polygon in Texture Space :
- *In the diffuse or specular color*
 - It must be normalized before storing in the color
 - It will get de-normalized across large polygons
 - Doesn't handle anisotropy well
- *Or in a set of 3D texture coordinates*
 - Use a Cube Map to renormalize the vector
 - Able to support scaling on textures
 - Can avoid CPU or GPU work
 - The L or H vector can be renormalized per-pixel via a texture, such as a Cube Map, Volume Map or Projected Texture



NVIDIA

The Resulting Texture Space



nVIDIA™

What about Animation?

- When triangles distort, so do their texture gradients, invalidating the model space->Texture Space matrix
- When triangles rotate, the model space->Texture Space matrix is invalid
- Therefore, the Texture Space will need to be updated or recomputed during animation
- The obvious approach, and one practical for simple models, is to simply go through the previous steps for each animation frame
 - Regenerate Texture Space for each triangle, then each vertex



NVIDIA

A Better Way – Update the Bases

- The two most popular animation techniques both work with Texture Space bump mapping **WITHOUT** requiring recalculating the entire basis
- Bone-Based Skinning (Indexed or Not)
- Keyframe Interpolation



nVIDIA™

Bone-Based Skinning

- For each axis of the Texture Space – S, T and SxT, “skin” the axis by putting it through the same matrix as the vertex normals
- Alternatively, skip the SxT axis and perform S cross T instead – can be cheaper if you have many bones



NVIDIA™

Keyframe Interpolation

- Create keyframes for the S, T and SxT axes as well
- Linearly interpolate between the S(0) and S(1) using the keyframe weight from 0 to 1
$$(1 - \text{Weight}) S0 + (\text{Weight}) * S1$$
- Now Normalize the result
- To handle scaled or stretched textures
 - Rescale by the linearly interpolated length of the two keyframe vectors
 - NormalizedVector *=
$$(1 - \text{Weight}) \text{LengthOf}(S0) + (\text{Weight}) * \text{LengthOf}(S1)$$



Keyframe Interpolation

- The normalizing of the vector approximates a SLERP
- The rescaling ensures that any stretching or scaling in the textures is preserved
 - especially important if morphing



nVIDIA™

Texture Space Calculations

- The cost of computing and updating Texture Space for moving models can seem large
- Keep it in perspective :
 - For a certain amount of per-vertex work, you are getting tremendous per-pixel detail
- All of the previous techniques for moving lights into Texture Space and updating the Texture Space vectors for moving objects can be handled with vertex shaders



nVIDIA™

Finally, the Dot Product

- Now we have a per-pixel representation of the L or H vector expressed in smoothly varying local coordinates
- Now, we can perform the DOT3 operation in the pixel shader or register combiners
- Next, apply the light color
- If computing H, we can raise the dot product to a power via self-multiplication
 - Watch for banding with high exponents
- We can apply attenuation effects either per-vertex or per-pixel with texture techniques
- Finally, combine with a decal texture or gloss map for the final result



NVIDIA[™]