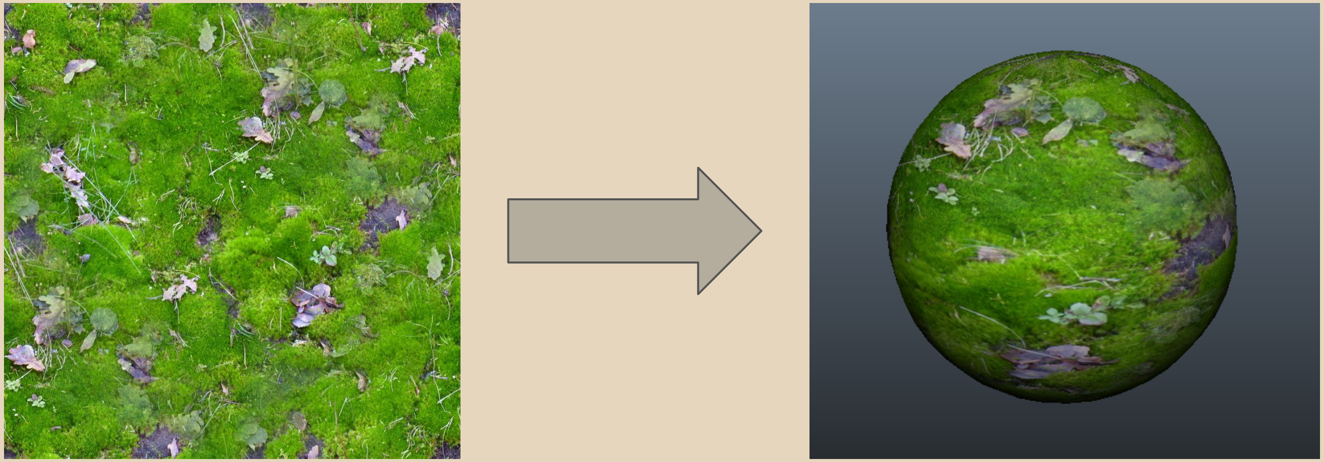
Texture and Normal Mapping

Adam Mally
CIS 560 Spring 2015
University of Pennsylvania

What is a texture?

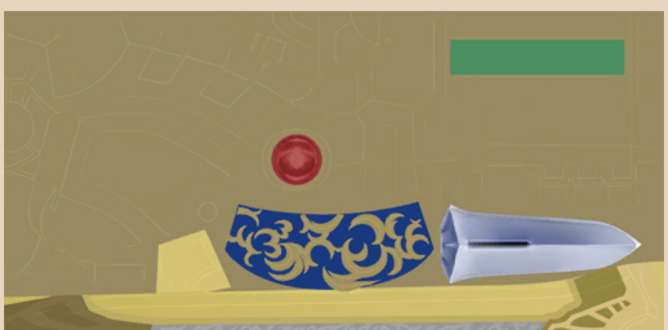
- An image which is sampled to determine the color of an object at a particular point on the object’s surface
  - Most commonly a 2D image, but 1D and 3D are also used on occasion
  - Could even use a “4D” image, which is an animated 3D texture
    - Or Smell-O-Vision
- Problem: The texture’s pixels must be mapped to points on the object’s surface before the texture can be sampled



Texture space

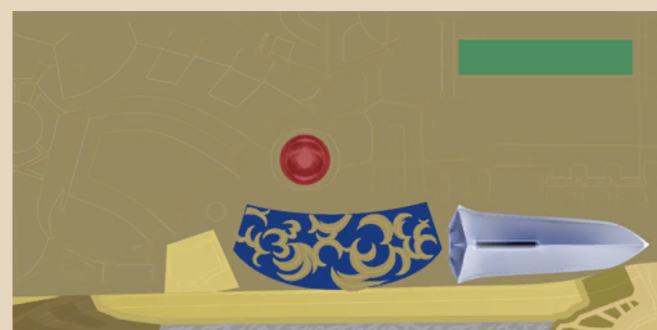
- We’ll be discussing 2D textures unless otherwise specified
- Two forms of parameterization: per-pixel and normalized
  - Per-pixel has, as one would expect, a width and a height equal to the width and height of the image
  - Normalized space has a range of 0 to 1 for both width and height, regardless of the aspect ratio of the texture
- The axes of texture space are commonly referred to as the U (horizontal) axis and the V (vertical) axis
  - W for the third axis if it exists

(0,256) (512,256)



(0,0) (512,0)

(0,1) (1,1)



(0,0) (1,0)

Sampling texture space

- Let's take an example of a unit square that has its vertices mapped to normalized UV space (i.e. its lower-left vertex is mapped to <0,0> and its upper-right vertex is mapped to <1,1>)
- We are given the point <0.4, -0.25> on our square and must find the texture coordinates that correspond to it
  - We know that (-0.5, -0.5) maps to <0,0> and (0.5, 0.5) maps to <1,1> so all we have to do is add 0.5 to each coordinate of our point to convert it from object space to texture space

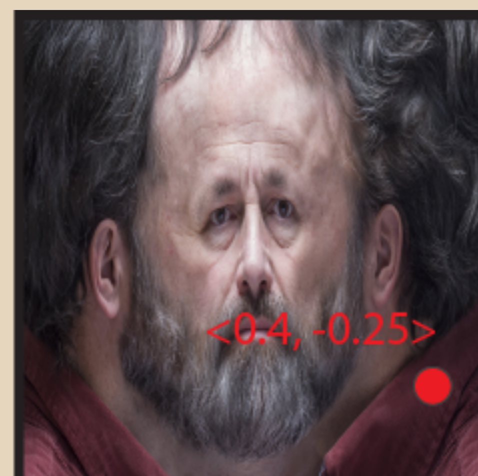
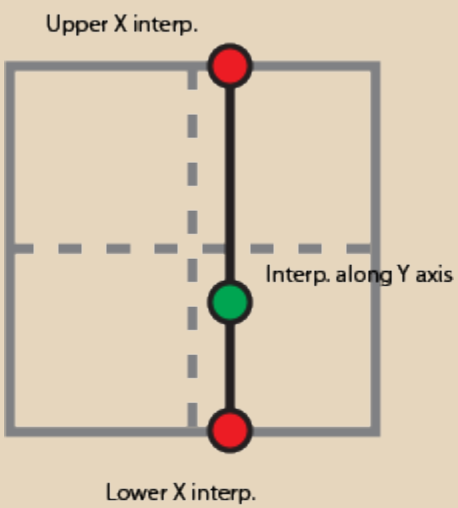


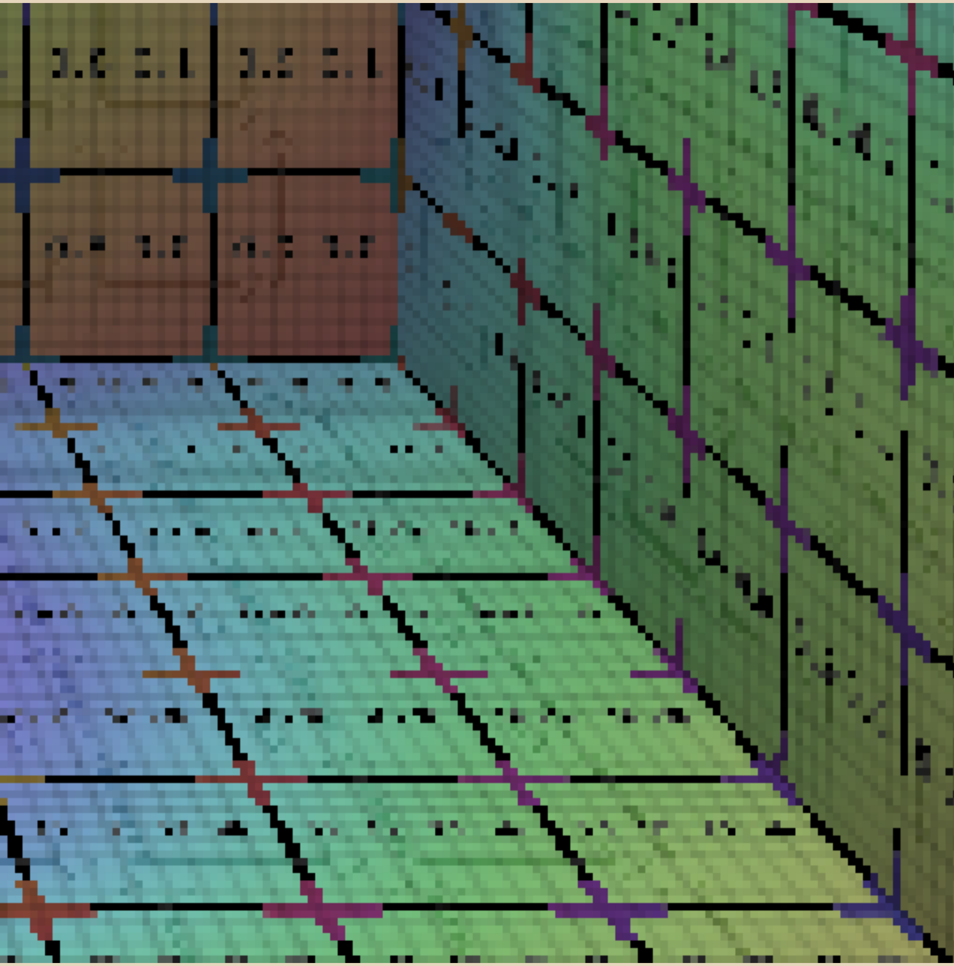
Image source: http://www.qwertzus.com/wordpress/wp-content/uploads/schikora-web.jpg

Sampling texture space

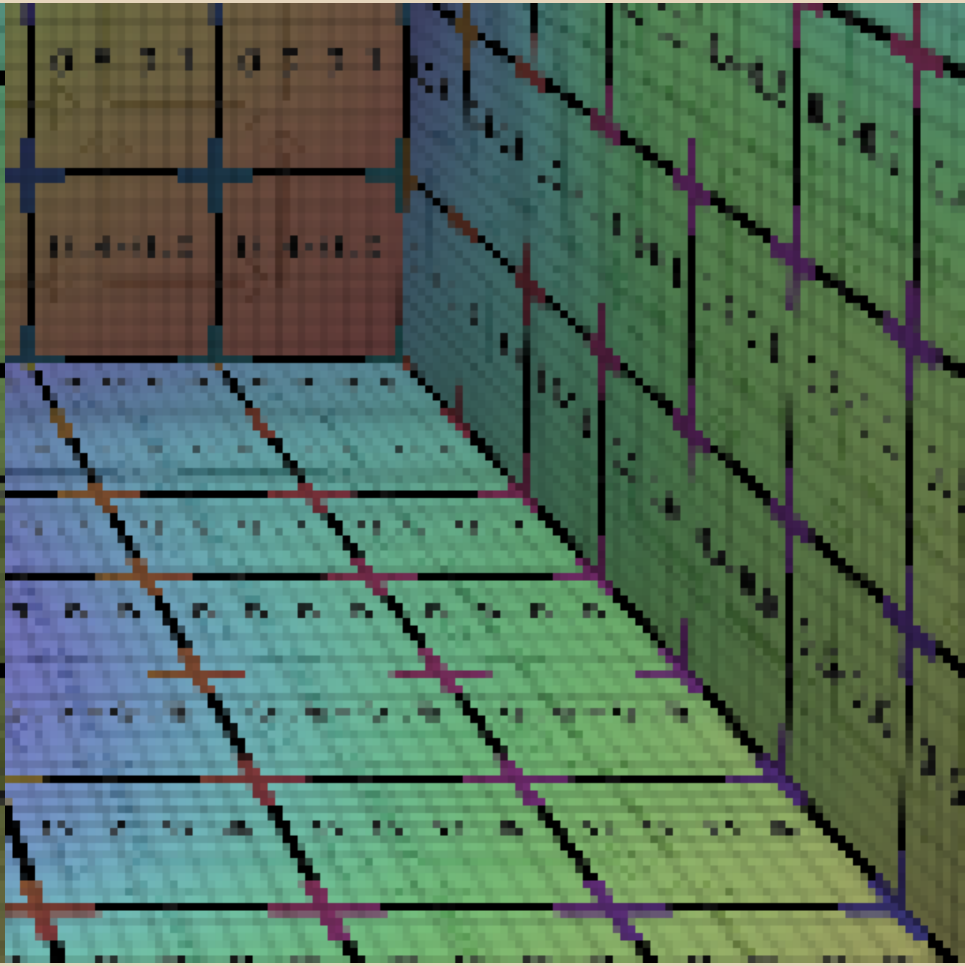
- Now we have our UV coordinates of <0.9, 0.25> in normalized space, and we need to convert them to pixel space in order to sample our image
- Simplest solution: multiply U by the image width and V by the image height, then truncate to an integer
  - Problem: Aliasing comes into play again!
  - Solution: Super-sample the texture using bilinear interpolation of pixels
- Bilinear interpolation: Interpolate the X axis of pixel space for both the upper and lower bounding Y pixel value, then interpolate those two values along the Y axis
  - Produces a weighted average of the four pixels surrounding the pixel-space texture coordinate



Bilinear interpolation of a texture



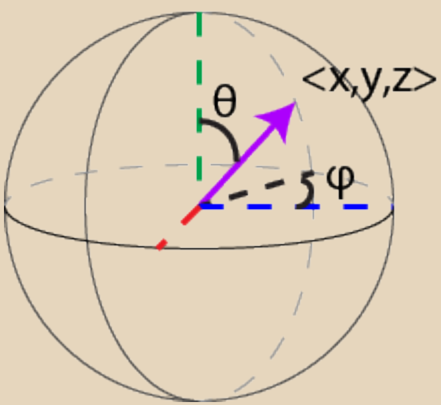
No bicubic interpolation



Bicubic interpolation

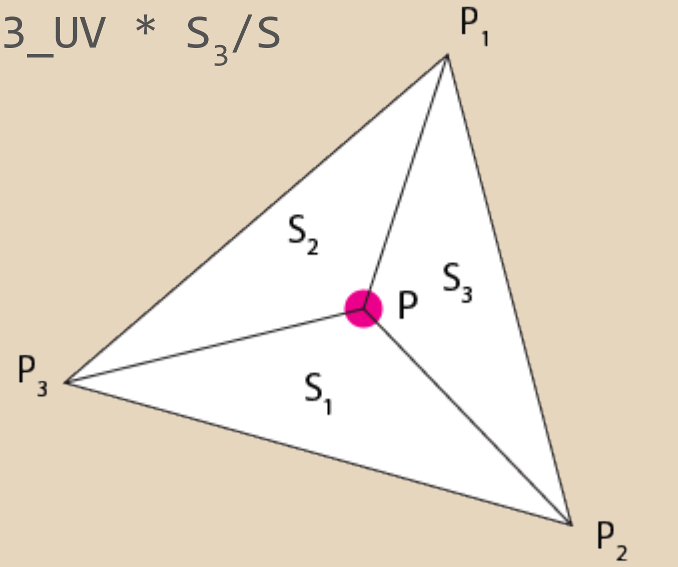
Spherical UV mapping

- We need to map a 3D coordinate to a 2D coordinate while trying to keep the 2D coordinates as undistorted as possible
- When working with spheres, we can use a spherical polar coordinate system for our UV coordinates
- Given a vector from the sphere's center to the point of intersection (i.e. the surface normal) we can compute an angle φ and an angle θ to represent our U and V coordinates (note that the vector must be normalized)
    - $\varphi = \text{atan2f}(z, x)$, add 2π to φ if the function returns a negative value
    - $\theta = \cos^{-1}(y)$
- Now we just have to convert our angles into normalized UV space
    - $U = 1 - \varphi/2\pi$
    - $V = 1 - \theta/\pi$



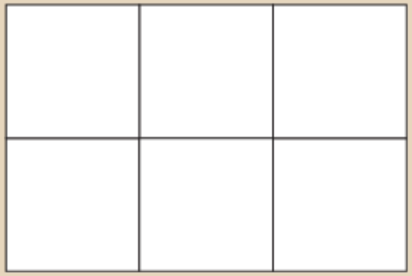
Interpolating triangle UVs

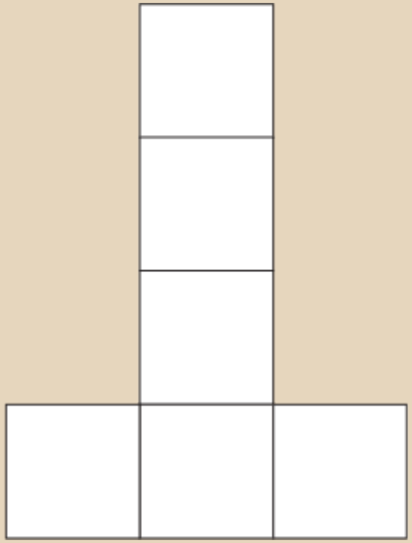
- Use **barycentric coordinates** again!
  - This assumes we already know the UV coordinates associated with P_1 , P_2 , and P_3
- S = area(P_1, P_2, P_3)
  - Area of a triangle = length(cross(P1-P2, P3-P2))/2
- S_1 = area(P, P_2, P_3)
- S_2 = area(P, P_3, P_1)
- S_3 = area(P, P_1, P_2)
- P_UV = P1_UV * S_1/S + P2_UV * S_2/S + P3_UV * S_3/S

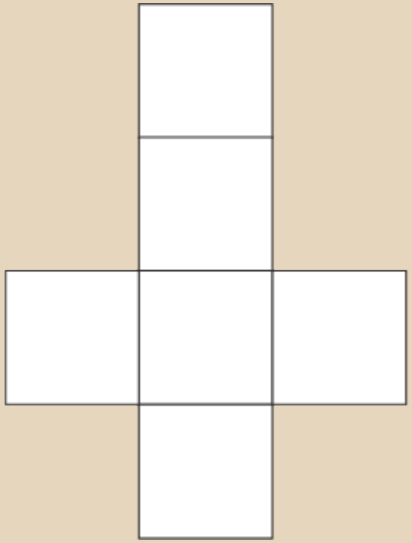


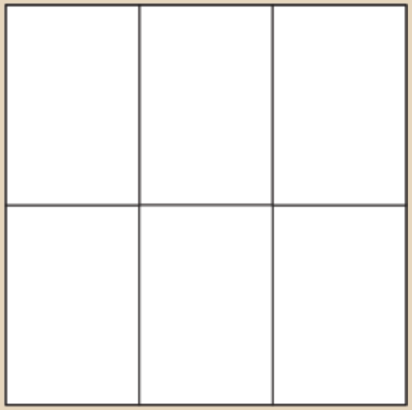
UV mapping a cube

- Usually want to give each face of the cube a unique portion of the texture to cover
- Usually want to prevent the texture distortion caused by giving the UVs a non-1:1 aspect ratio
- Can make an “unfolded cube” formation for ease of seam hiding
- No “best” solution, just pick one you prefer



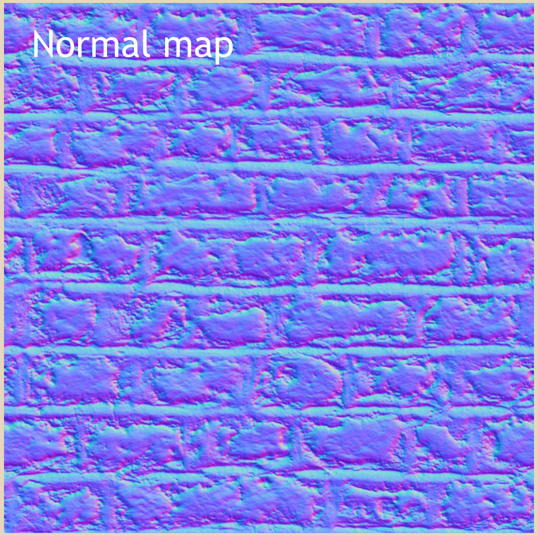






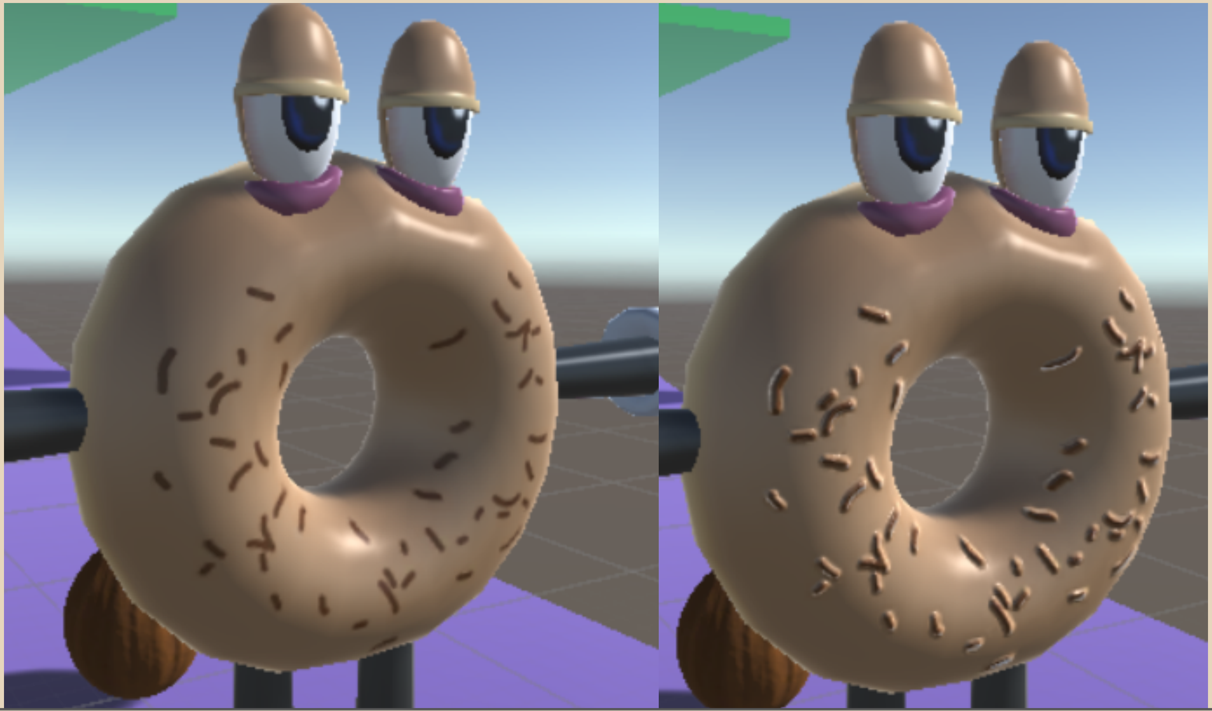
Normal maps

- An RGB texture used to alter the local-space normal found at a given point on an object
- Works on the assumption that the untransformed normal at a given point in object space is <0,0,1>, which is obviously untrue most of the time
  - In order to use a normal map properly, a matrix that transforms vectors from texture space to object space must be computed
- The “identity” color of an object-space normal map is <128, 128, 255>, which corresponds to a texture-space normal of <0,0,1>
  - In a normal map, 0 corresponds to -1 and 255 corresponds to 1 for any color channel



Normal maps

- Commonly used in applications that have to run in real time
    - Adds an extra level of detail for minimal computation cost



- Gives the illusion of 3D sprinkles when they’re actually just a texture applied to the torus

Normal maps

- To compute the matrix that transforms from texture space to object space, we need three vectors: a normal, a tangent, and a bitangent
  - These form a local orthonormal space
  - The same concept as creating the orientation matrix for a camera
- We already have our normal, so we must compute the other two vectors
- The tangent corresponds to our local X axis, so it should align with the U axis of our texture
  - Spherical example: For all points except the very poles of our sphere, we can get the tangent by crossing <0,1,0> with our normal (remember, order matters in cross products)
- Likewise, the bitangent corresponds to our local Y axis, so it should align with the V axis of our texture
  - Sphere: Get bitangent by crossing our normal with our tangent
- Our transformation matrix can now be created:

$$\begin{vmatrix} T.x & B.x & N.x & 0 \\ T.y & B.y & N.y & 0 \\ T.z & B.z & N.z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

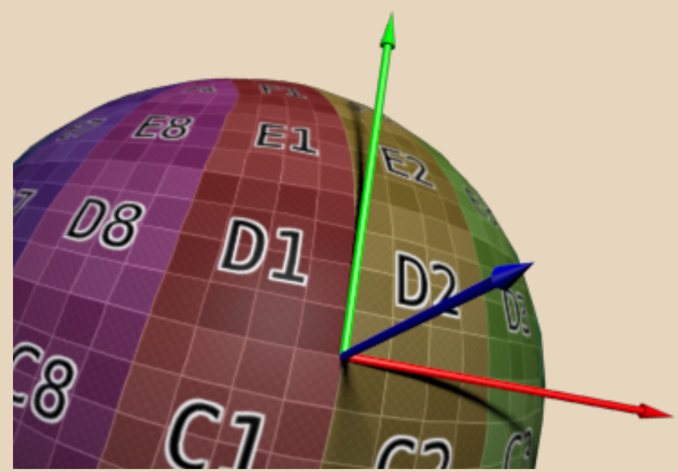
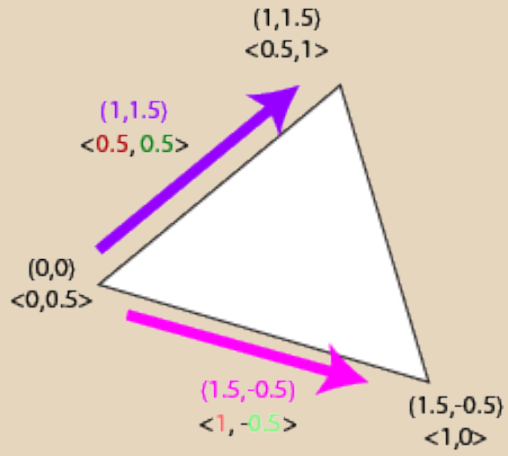


Image source: http://www.opengl-tutorial.org/wp-content/uploads/2011/05/NTBFromUVs.png

Tangents and bitangents based on UVs

- In a general case, such as a triangle, we want to be able to compute our tangent and bitangent regardless of UV mapping technique
- If we know three points **of a plane** on our object and the UVs at those points (which gives us two changes in positions and UVs), we can solve a system of linear equations to compute our tangent and bitangent:
  - `ΔPos1 = ΔUV1.x * T + ΔUV1.y * B`
  - `ΔPos2 = ΔUV2.x * T + ΔUV2.y * B`

- `B = (ΔPos2 - ΔUV2.x * T)/ΔUV2.y`
- `T = (ΔUV2.yΔPos1 - ΔUV1.yΔPos2) / (ΔUV2.yΔUV1.x - ΔUV1.yΔUV2.x)`



Normal map summary

- Need to compute an orientation matrix to convert the map's normal into local object space
- In each color channel, normal maps use 0 to represent -1, 128 to represent 0, and 255 to represent 1
- Sample the normal map like a texture, then multiply the acquired normal by the orientation matrix to get an object-space normal
- Use this object-space normal in place of the object's default normal for a more detailed look to your model