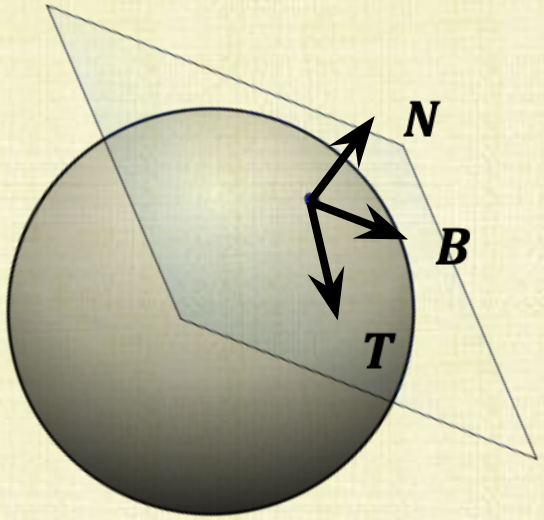
# More Texture Mapping

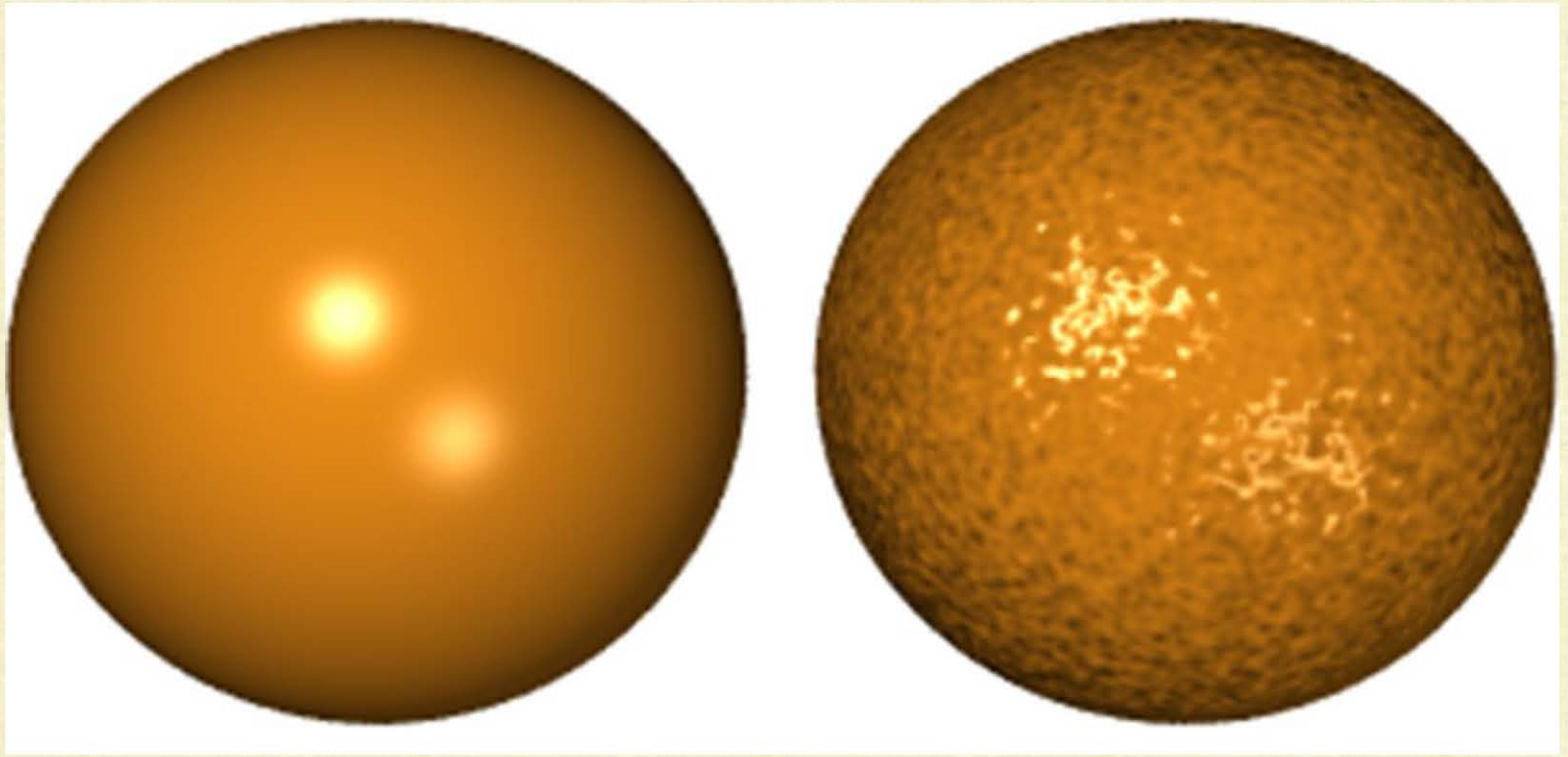
# Parameterizing the Tangent Plane

- Assume that the triangle's $(x, y, z)$ coordinates are linear functions of the $(u, v)$ parameter space on a triangle
  - That is, $x = a_0u + b_0v + c_0$, $y = a_1u + b_1v + c_1$, $z = a_2u + b_2v + c_2$
- Then for each of the 3 triangle vertices, plug in its $(x, y, z)$ and $(u, v)$ values, resulting in 9 equations:
  - a 3x3 system for $a_0$, $b_0$, $c_0$; a 3x3 system for $a_1$, $b_1$, $c_1$; a 3x3 system for $a_2$, $b_2$, $c_2$
- Solve each of the three 3x3 systems separately, and then define the tangent $T = \left(\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}\right) = (a_0, a_1, a_2)$, the binormal $B = \left(\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}\right) = (b_0, b_1, b_2)$, and the normal $N = T \times B$
  - These non-unit directions need to be normalized
- $T$ and $B$ should be in the plane of the triangle, and so $N$ should agree with the triangle normal direction



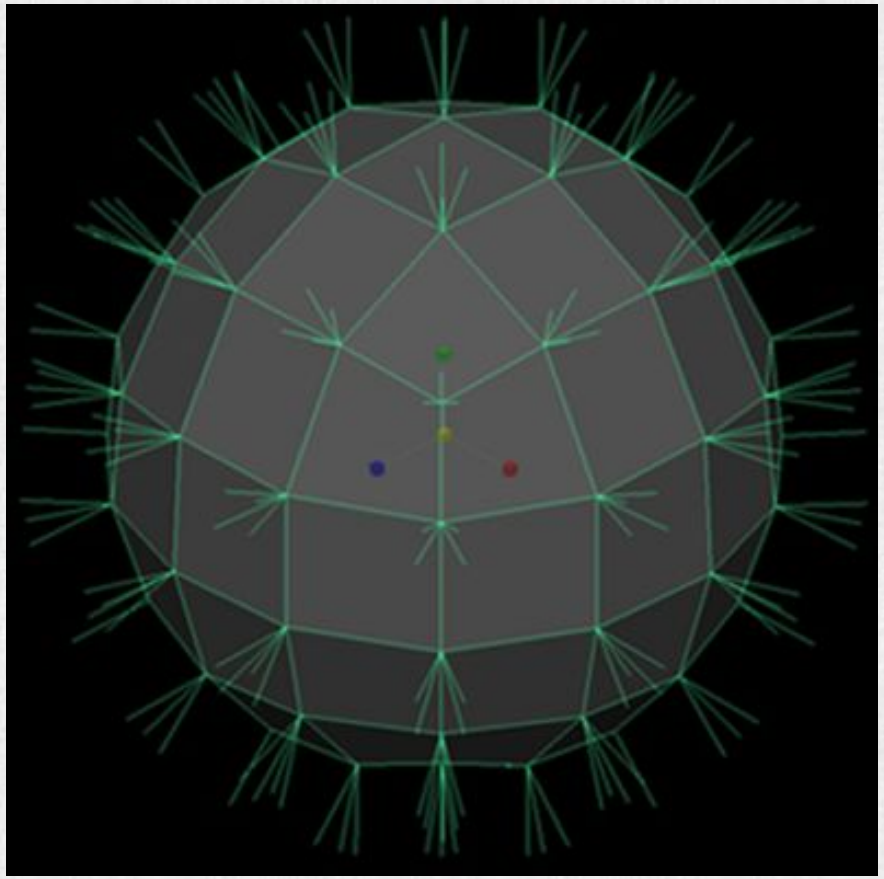
# Perturbing the Normal

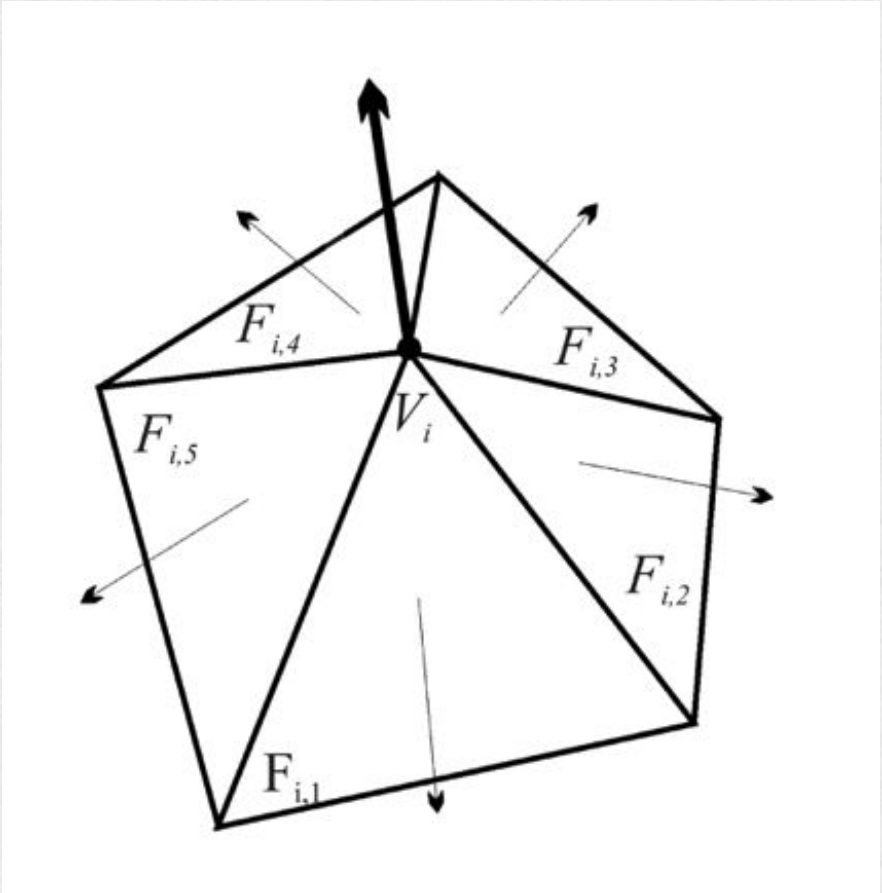
- In the texture, store a new normal $(n_T, n_B, n_N)$, instead of a color value
- Define this new normal relative to the local coordinate system on the triangle, i.e. $N^{new} = n_T\hat{T} + n_B\hat{B} + n_N\hat{N}$
  - $\hat{T}$, $\hat{B}$, and $\hat{N}$ have been normalized to unit length
  - Note that $(n_T, n_B, n_N) = (0,0,1)$ is the unperturbed normal
- This perturbed normal can "fake" geometric details (as we have seen before – see next 2 slides)



# Recall: (Averaged) Vertex Normals

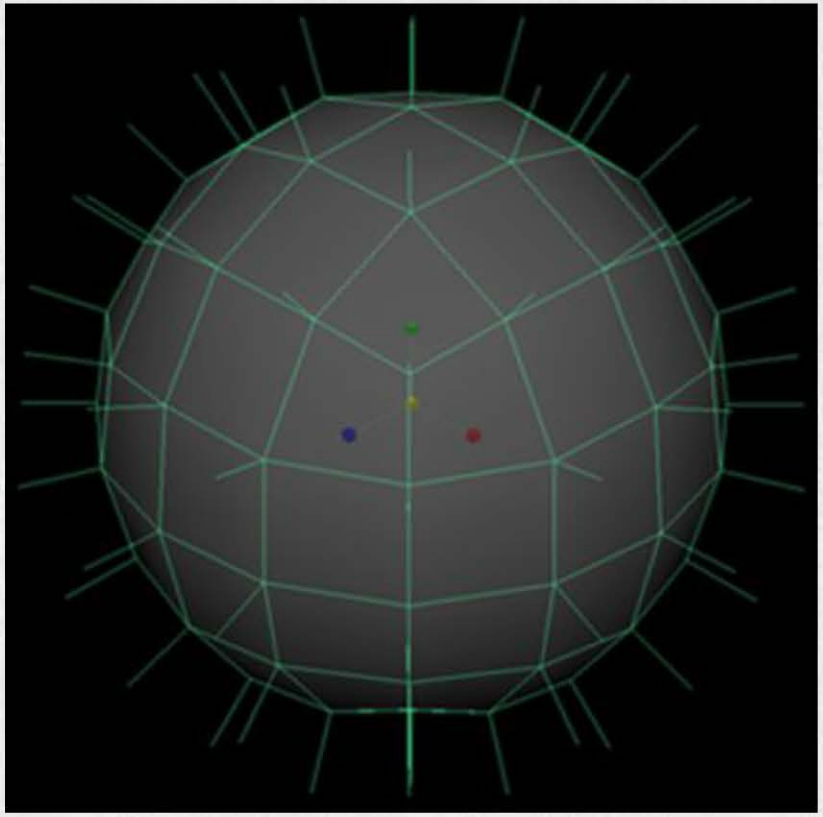
- Each vertex has a number of incident triangles, each with their own normal
- Averaging those face normals (possibly using a weighted average based on area, angle, etc.) yields a unique normal for each vertex

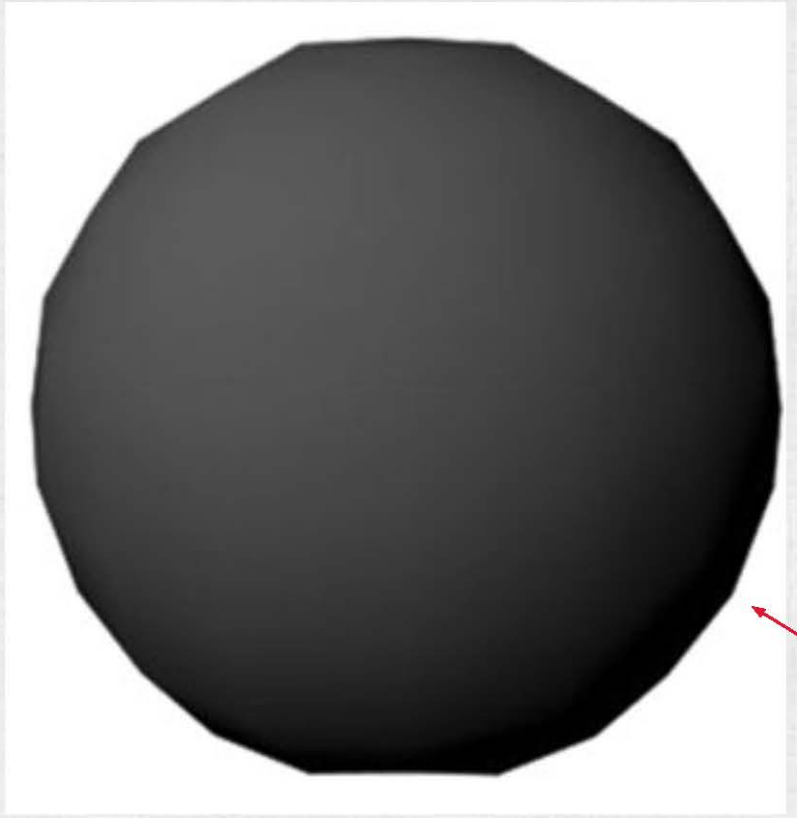




# Recall: Interpolating Vertex Normals

- Given barycentric weights at a point $p$, interpolate a normal at $p$ from the unique (precomputed) vertex normals: $\widehat{N}_p = \frac{\alpha_0 N_0 + \alpha_1 N_1 + \alpha_2 N_2}{\|\alpha_0 N_0 + \alpha_1 N_1 + \alpha_2 N_2\|_2}$
- This is called smooth shading (as opposed to flat shading)





# Bump Map

- Single-channel (grey-scale) height map $\boldsymbol{h(u,v)}$, representing a 3D surface given by $\boldsymbol{(u,v,h(u,v))}$
- The tangent plane at point $\boldsymbol{(u_0,v_0,h(u_0,v_0))}$ is:

$$-\left.\frac{\partial h(u,v)}{\partial u}\right|_{(u_0,v_0)}(u-u_0)-\left.\frac{\partial h(u,v)}{\partial v}\right|_{(u_0,v_0)}(v-v_0)+\big(h-h(u_0,v_0)\big)=0$$

- The outward normal $\left(-\left.\frac{\partial h(u,v)}{\partial u}\right|_{(u_0,v_0)},-\left.\frac{\partial h(u,v)}{\partial v}\right|_{(u_0,v_0)},1\right)$ is normalized to obtain $\boldsymbol{(n_T,n_B,n_N)}$
- Partial derivatives can be computed using finite differences:

$$\left.\frac{\partial h(u,v)}{\partial u}\right|_{(u_0,v_0)}=\frac{h(u_1,v_0)-h(u_{-1},v_0)}{u_1-u_{-1}}\text{ and }\left.\frac{\partial h(u,v)}{\partial v}\right|_{(u_0,v_0)}=\frac{h(u_0,v_1)-h(u_0,v_{-1})}{v_1-v_{-1}}$$

# Normal Map

- A normalized $(\boldsymbol{n_T}, \boldsymbol{n_B}, \boldsymbol{n_N})$ has each component in $[-\mathbf{1}, \mathbf{1}]$, so one can convert back and forth to a color via:

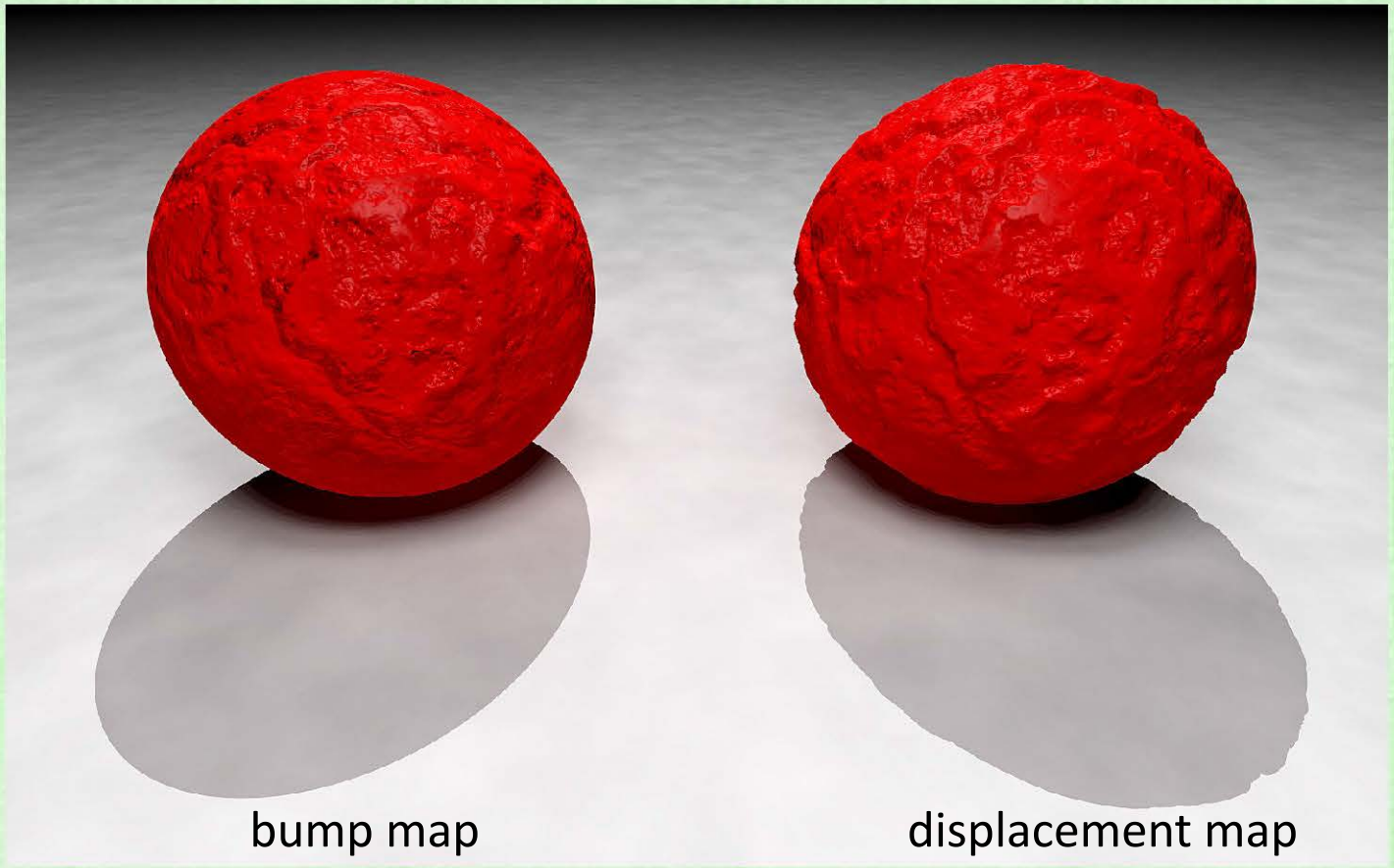
$$(\boldsymbol{R}, \boldsymbol{G}, \boldsymbol{B}) = \mathbf{255} * \frac{(\boldsymbol{n_T}, \boldsymbol{n_B}, \boldsymbol{n_N}) + (\mathbf{1}, \mathbf{1}, \mathbf{1})}{\mathbf{2}} \quad \text{and} \quad (\boldsymbol{n_T}, \boldsymbol{n_B}, \boldsymbol{n_N}) = \mathbf{2} * (\boldsymbol{R}, \boldsymbol{G}, \boldsymbol{B})/\mathbf{255} - (\mathbf{1}, \mathbf{1}, \mathbf{1})$$

- Normal map has larger storage, storing an RGB image as opposed to a single channel for a bump map (only storing height); however, less computation is required (no finite differences) to compute the normal

**normal mapping on a plane (note the variation of specular highlights created by the variation of the normal)**

# Displacement Mapping

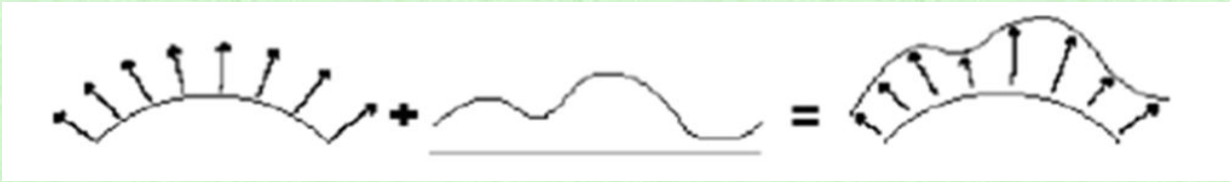
- Perturb the surface geometry (added detail requires making new temporary geometry on-the-fly at render time)
- Store a height map $\boldsymbol{h(u, v)}$, used to (actually) perturb vertices in the normal direction
- Pros: self-occlusion, self-shadowing, correct **silhouettes**
- Cons: expensive, requires adaptive tessellation, still need bump/normal map for sub-triangle (fake) detail



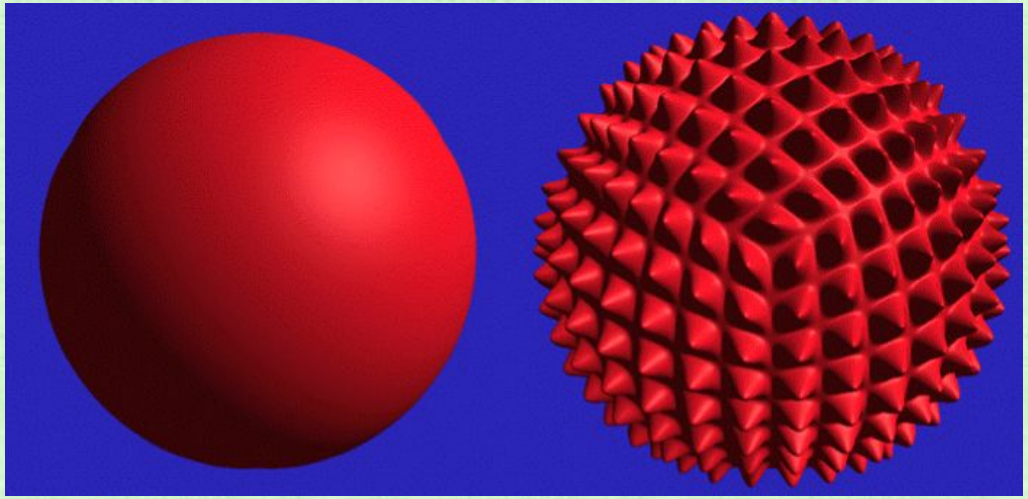
bump map displacement map

# Displacement Mapping

- Compute normals on the vertices of the original mesh (probably also subdivided on-the-fly for more detail)
- Move each vertex in the (original) normal direction a distance $\boldsymbol{h(u,v)}$
- Compute new normals for the now perturbed mesh

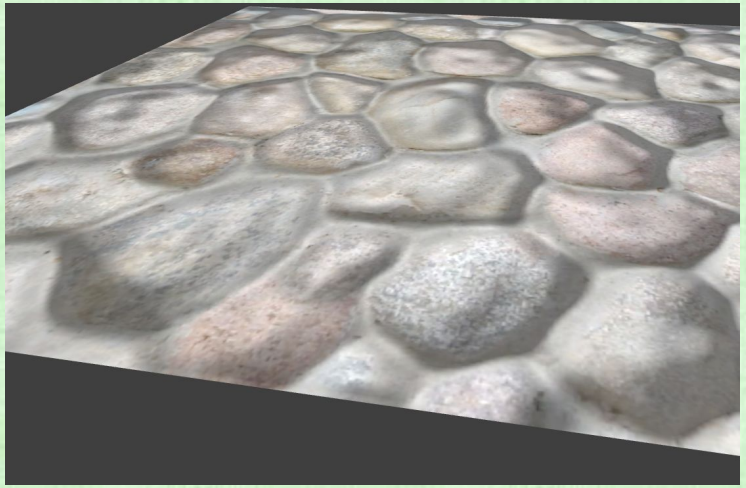


original geometry $\boldsymbol{h(u,v)}$ displacement map

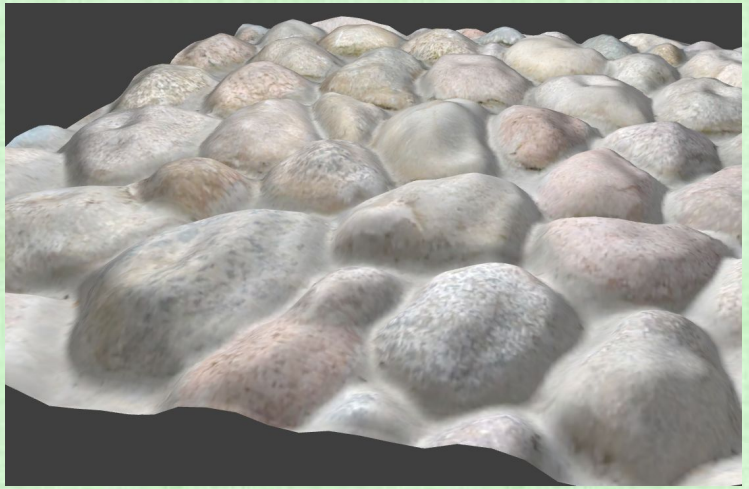


original geometry displacement map

bump map



displacement map



# Recall: Measuring Incoming Light

- Place a small reflective chrome sphere (a light probe) somewhere in the world
- Photograph it, in order to measure/record the intensity of light incoming from all directions

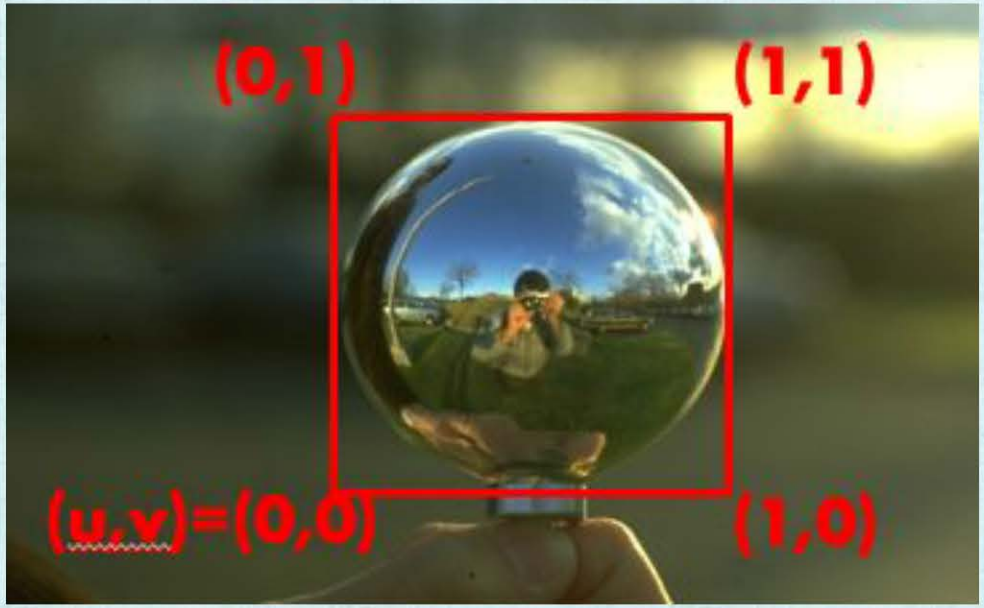
# Recall: Measuring Incoming Light

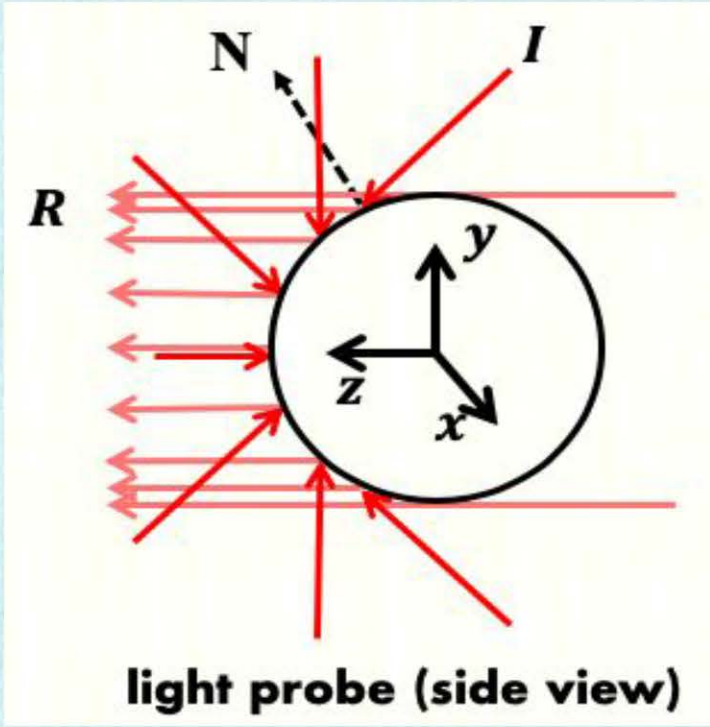
- Understanding/measuring the incoming light allows one to render a new synthetic object in the original scene (with realistic lighting)

# Environment Mapping

••Assume objects in the environment are infinitely far away, so that the intensity of environmental light depends only on the incoming direction $\boldsymbol{I}$ (not on position)

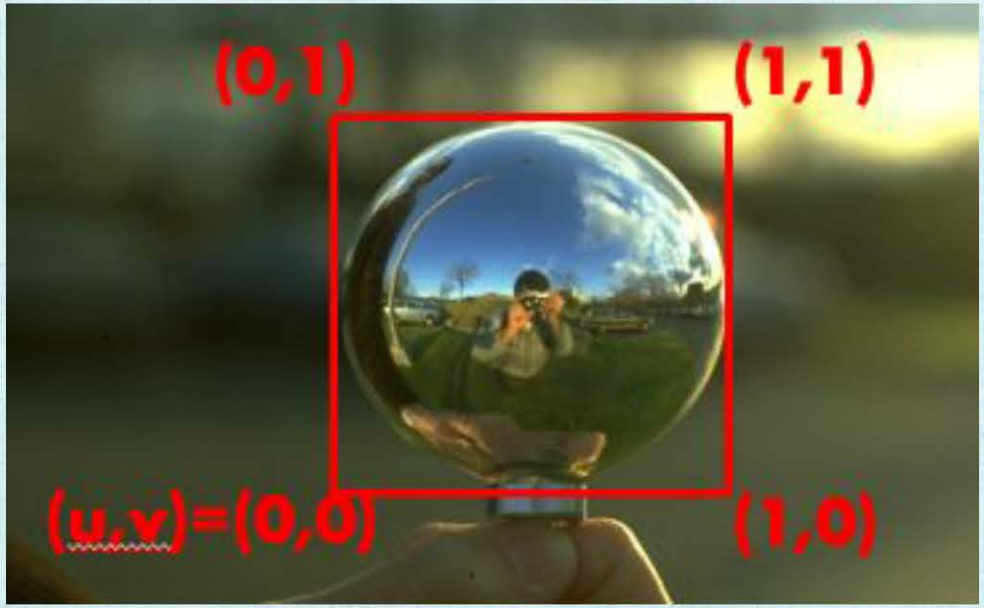
- $\boldsymbol{R}$ is the direction from the light probe to the camera
- $\boldsymbol{I}$ and $\boldsymbol{R}$ are equal-angle from $\boldsymbol{N}$, according to mirrored reflection
- Thus, $\boldsymbol{N}$ has a one-to-one correspondence with $\boldsymbol{I}$
- In the texture (inside the red square below), each texel stores light from one direction $\boldsymbol{I}$ corresponding to one surface normal $\boldsymbol{N}$

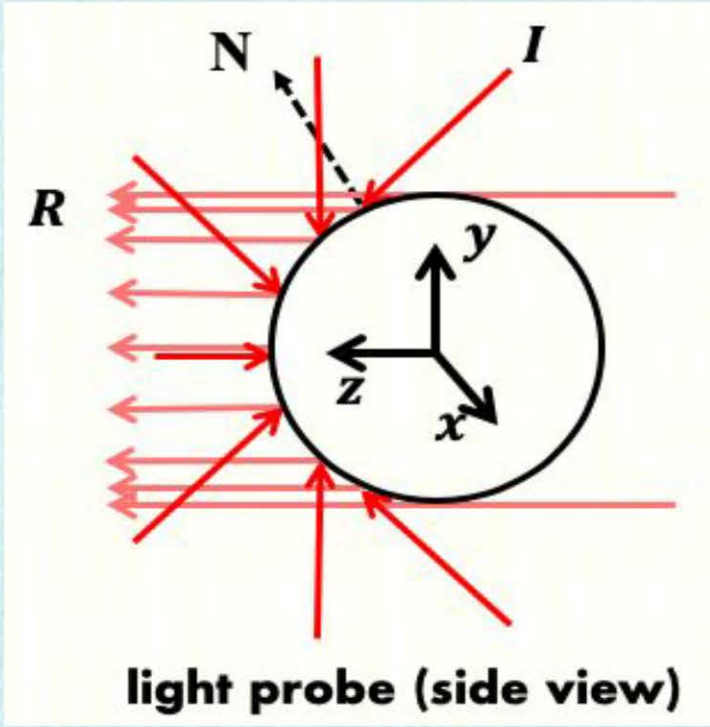


# Environment Mapping

- Coordinate system at the sphere center gives surface normal $\boldsymbol{N} = (\boldsymbol{n_x}, \boldsymbol{n_y}, \boldsymbol{n_z}) = (\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})/\sqrt{\boldsymbol{x^2} + \boldsymbol{y^2} + \boldsymbol{z^2}}$
- $\boldsymbol{n_x}$ and $\boldsymbol{n_y}$ in the range [-1, 1] are used to obtain texture coordinates $(\boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{n_x} + \boldsymbol{1}, \boldsymbol{n_y} + \boldsymbol{1})/\boldsymbol{2}$, which correspond to the locations show below (on the left figure, in red)
- When rendering a CG object, use the local surface normal to compute texture coordinates in order to fetch the color and direction of the incoming light (from the texture map)

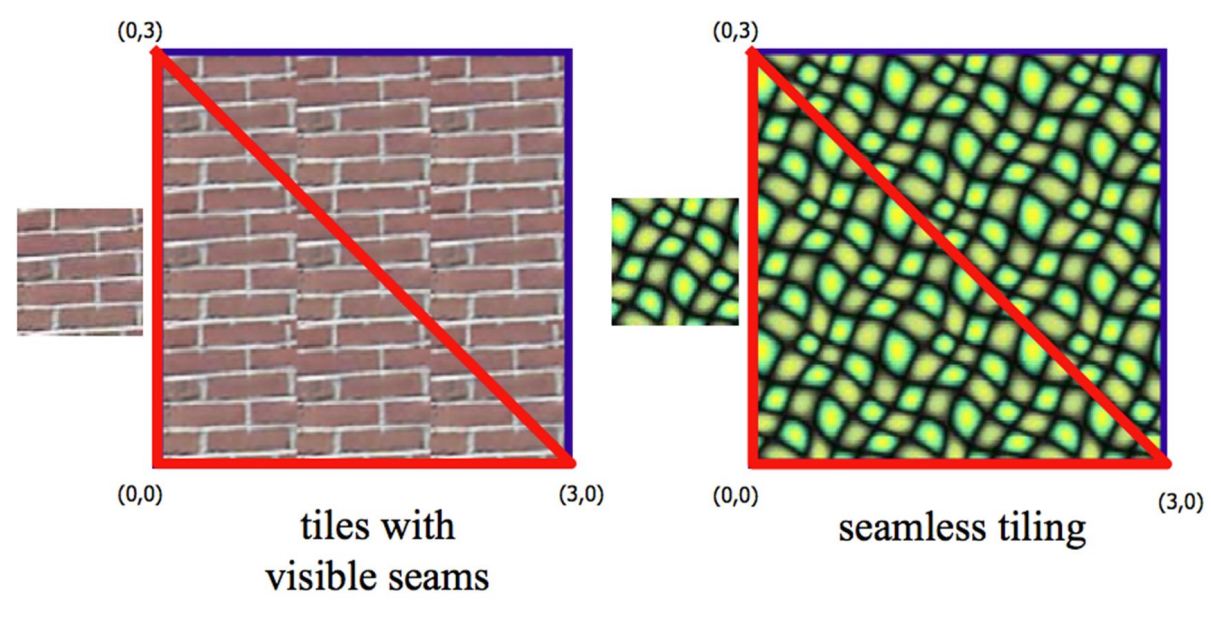


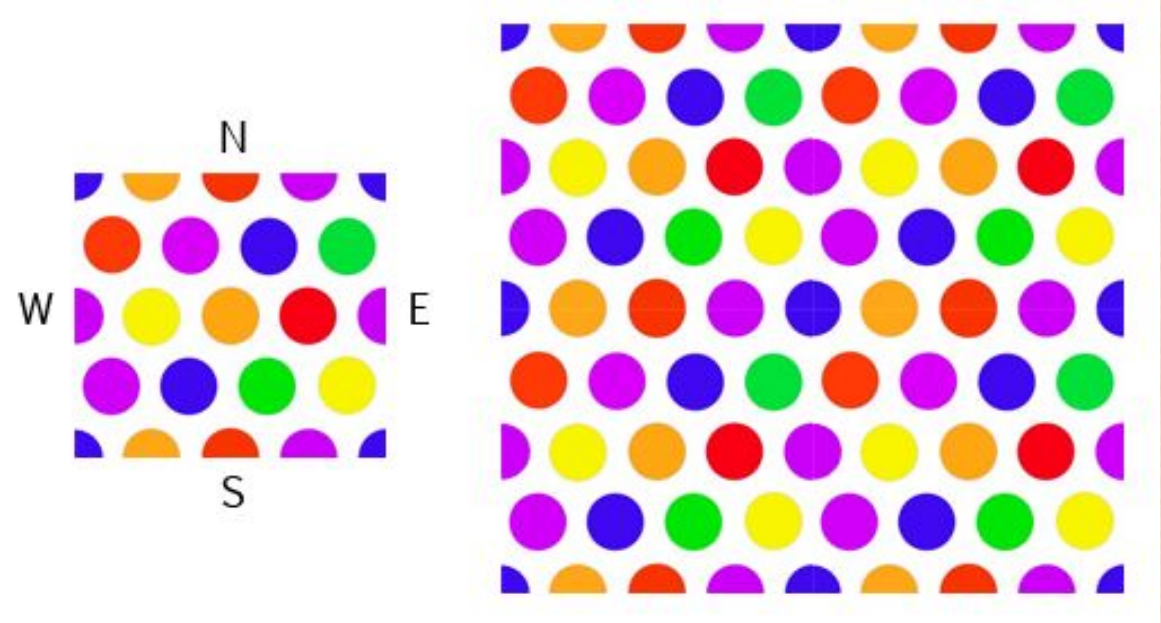



# Environment Mapping

# Texture Tiling

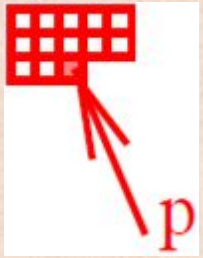
- Stretching textures can look pretty bad, so one often tiles a small texture sample to create a bigger one
  - E.g. Can't stretch 10 bricks to cover a entire wall of a building
- Have to be careful to avoid unwanted visible seams at tile boundaries



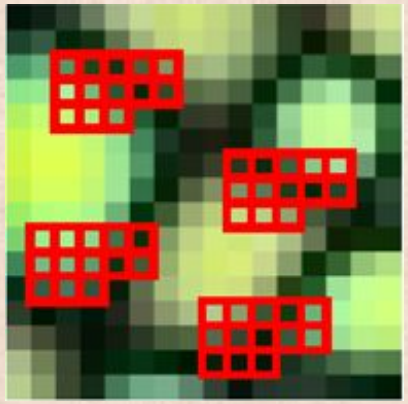



# Texture Synthesis: Pixel Based

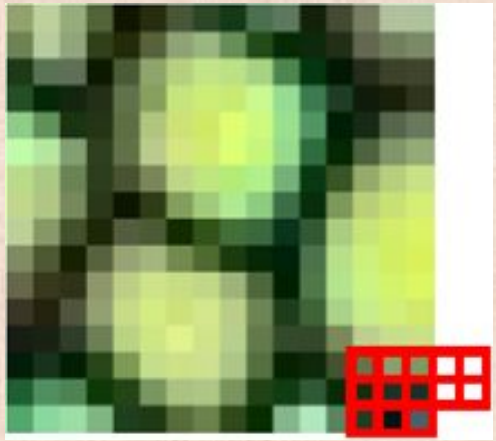
- Create a large non-repetitive texture (one pixel at a time) from a small sample (by using its structural content)
- To generate the texture for pixel *p*
  - compare *p*'s neighboring pixels in the (red) stencil to all potential choices in the sample
  - choose the one with the smallest difference to fill pixel p
- Generate the texture in a raster scan ordering
- When the stencil looks up values outside the domain, periodic boundary conditions are used (so the last few rows and columns of pixels are pre-generated with random values)




stencil



texture sample

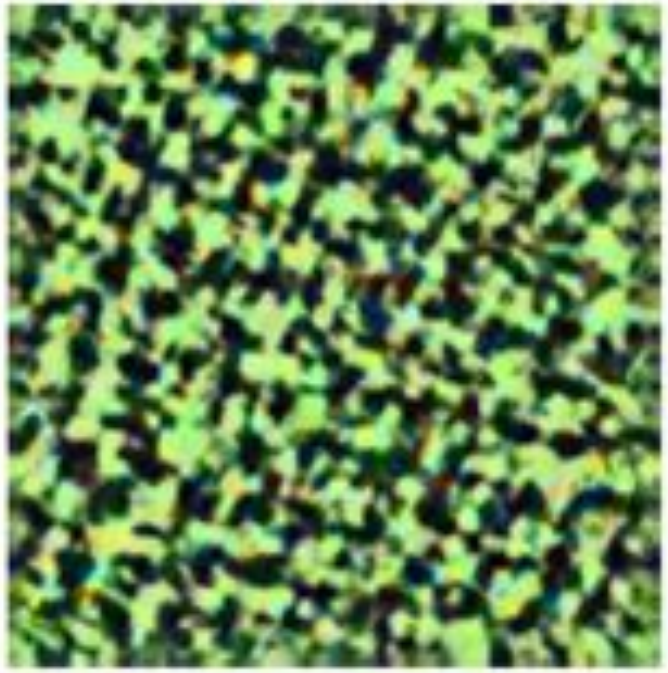


raster scan ordering (with randomly generated periodic boundaries)

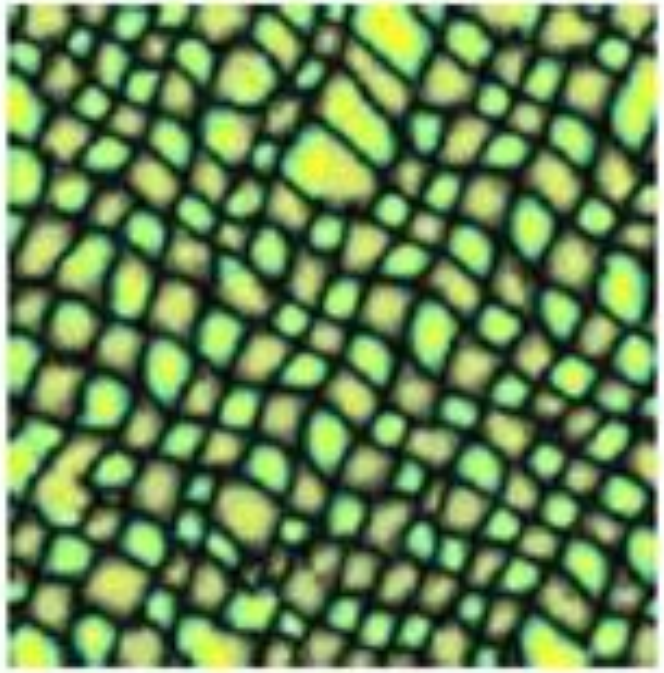
# Texture Synthesis: Pixel Based

- Pro: Reduces repetitive patterns compared to texture tiling
- Pro: Generated texture has similar content to the input sample
- Con: May lose too much structural content and/or create noisy or poorly structured textures

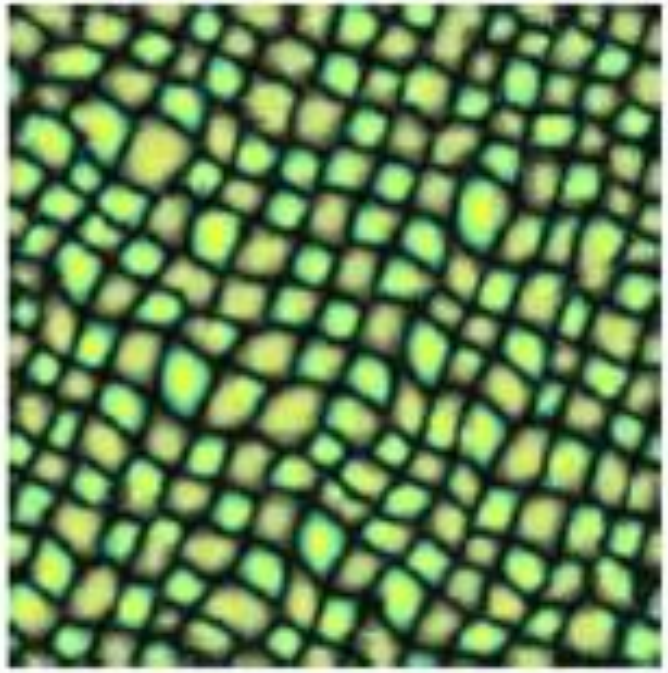
Sample



Heeger and Bergen



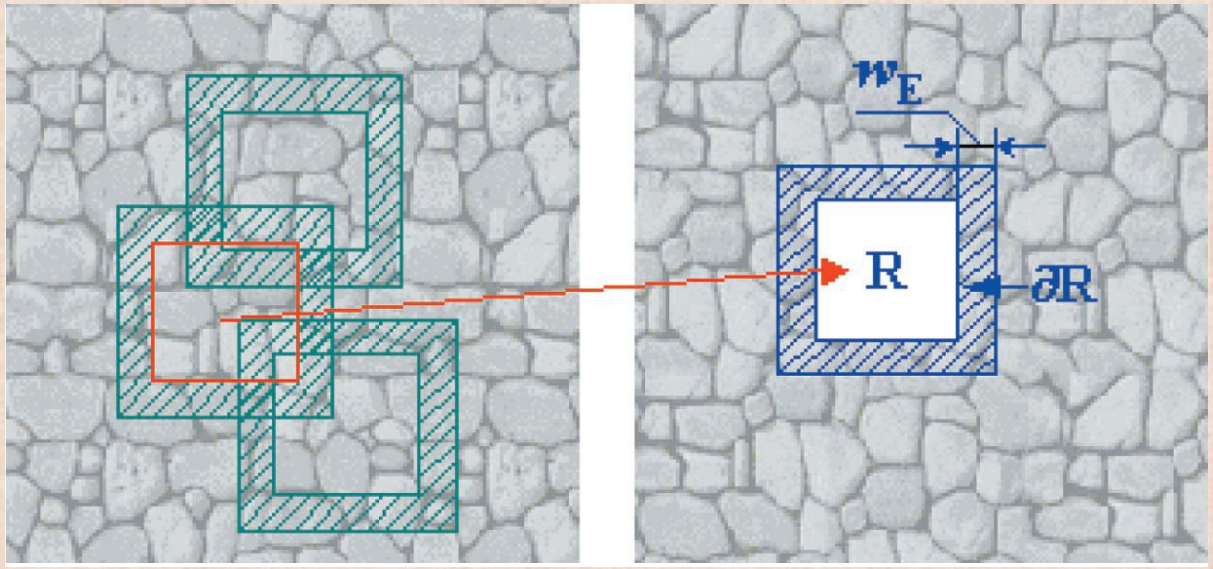
Efros and Leung



Wei and Levoy

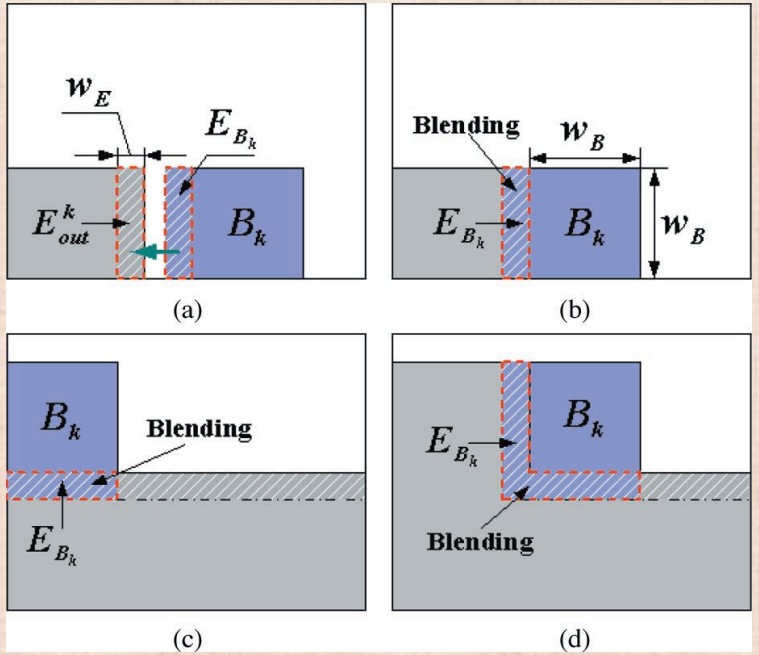
# Texture Synthesis: Patch Based

- Similar to texture tiling, but:
  - only uses a subset of the original texture to avoid repetitive blocks
  - blends overlapped regions to remove “seams”
- For each patch being considered:
  - search the original sample to find candidates which best match the overlap regions (on the boundaries)
  - choose from the good candidates
- Advantages: uses *more* structural content, so noise is less problematic

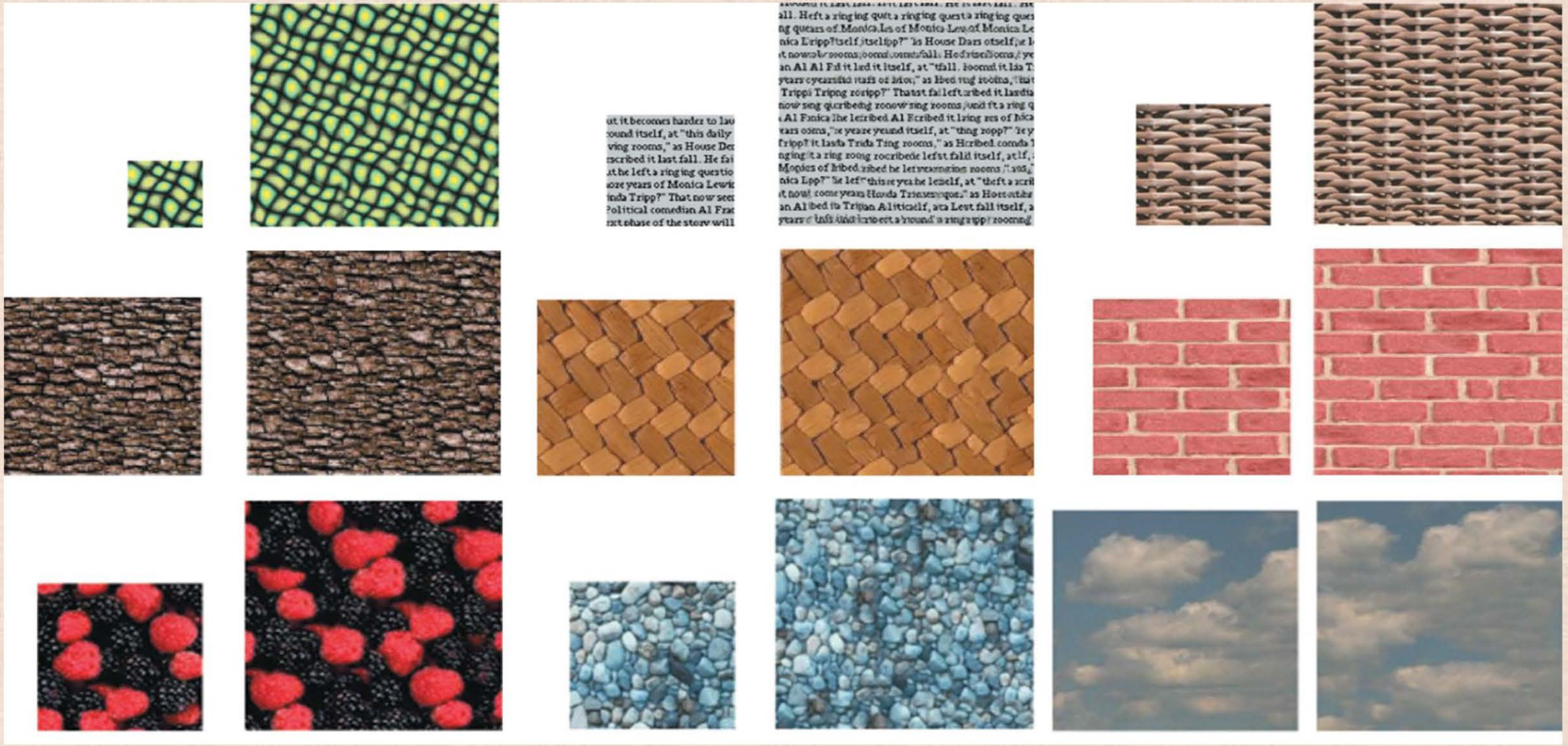



sample texture

matching boundary regions

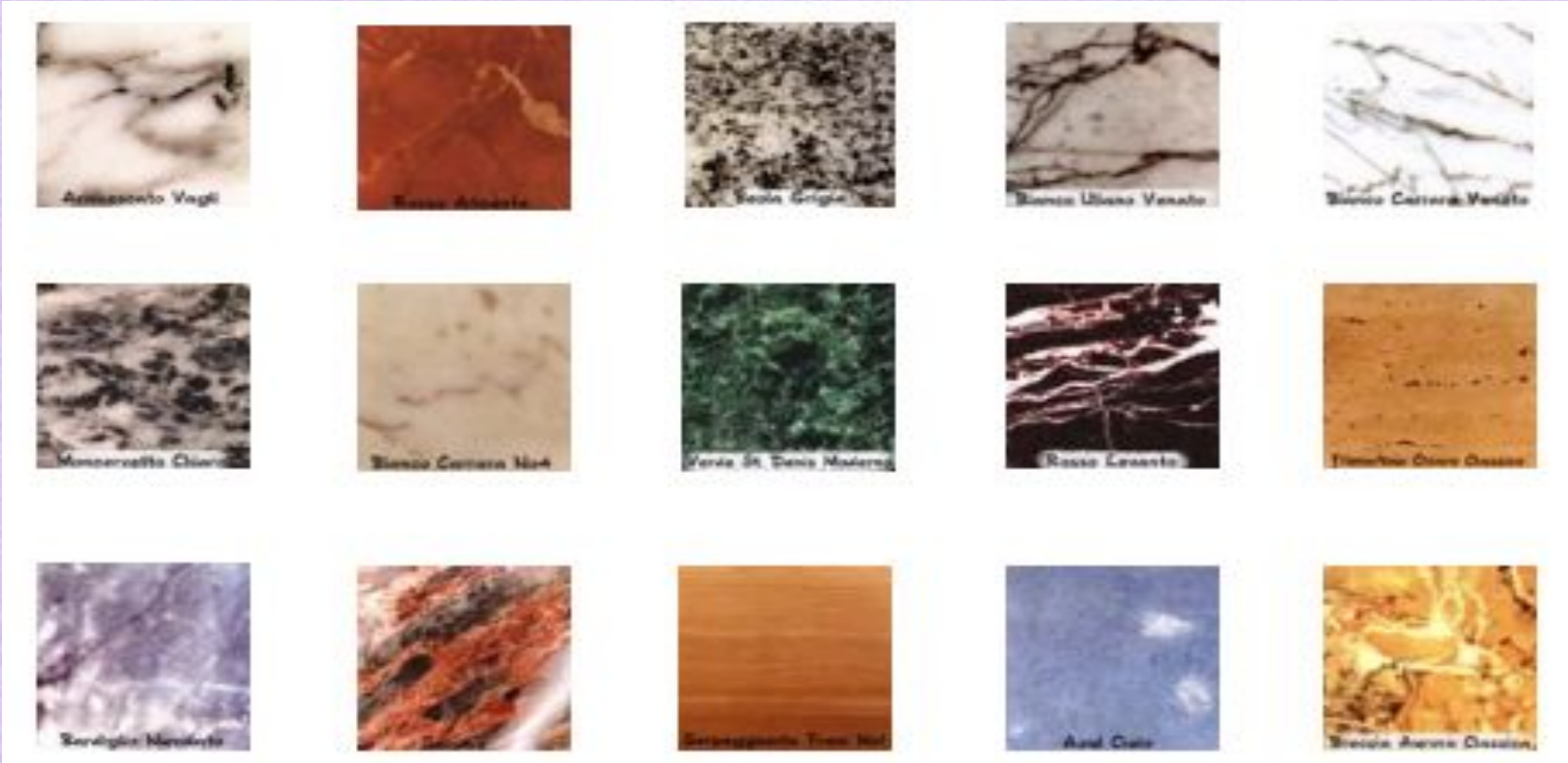


# Texture Synthesis: Patch Based



# Procedural Texture Generation

- Created via mathematical/computational algorithms
- Good for generating natural elements (wood, marble, granite, stone, etc.)
- A natural look is achieved using noise or turbulence functions, which are used as a numerical representation of the "randomness" found in nature
- E.g., marble is metamorphosed limestone, which typically contains a variety of material impurities (chaotically distributed during metamorphosis)

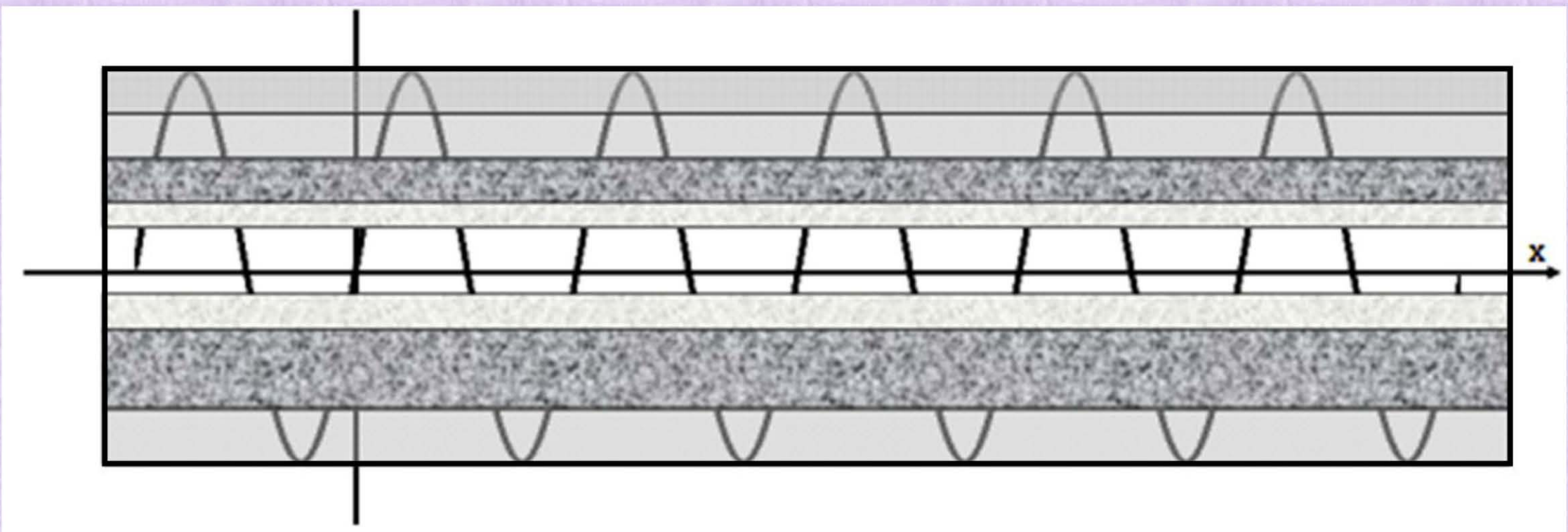


# Marble Texture

- Predefine layers of different colors
- Use a function to map $(u, v)$ texture locations to layers

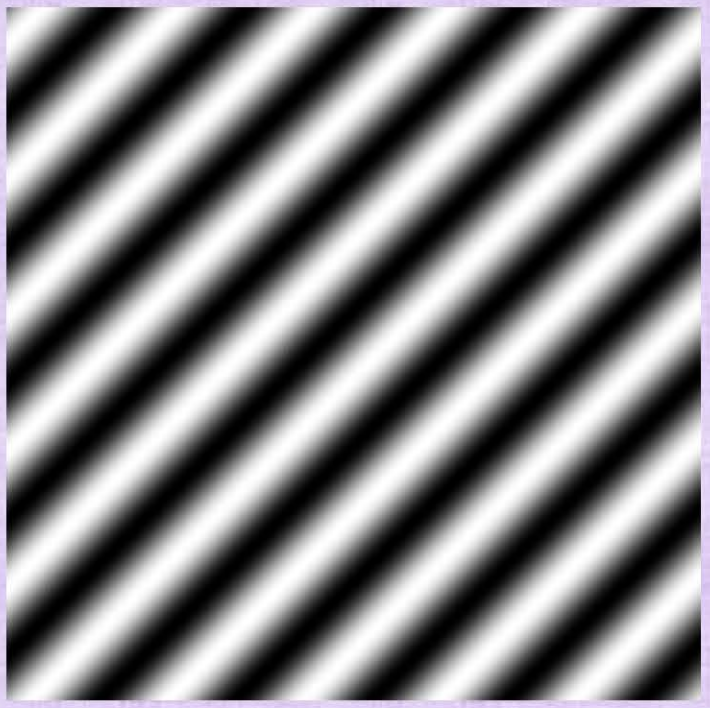
- For example:

$$marbleColor(u,v) = LayerColor\big(\sin(k_u u + k_v v)\big)$$

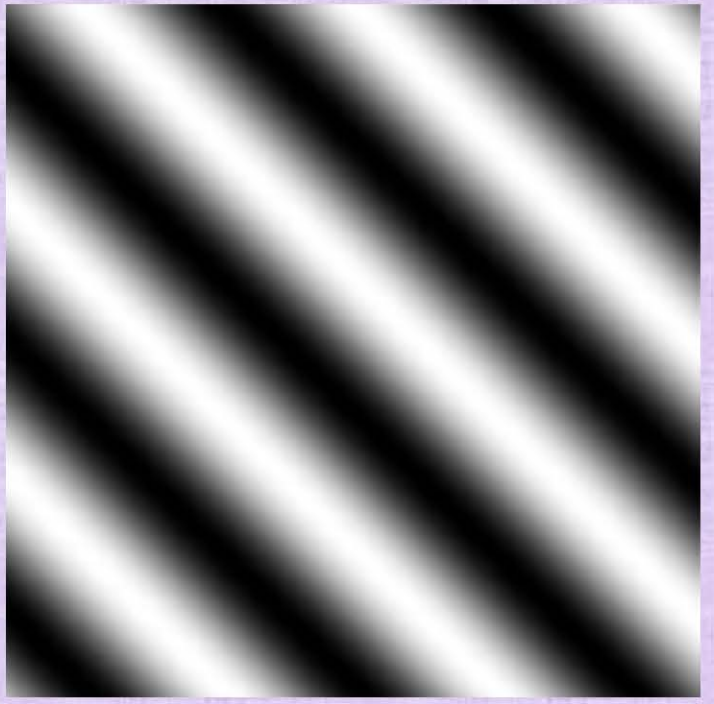


# Marble Texture

- $k_u$ and $k_v$ are spatial frequencies set by the user
- $(k_u, k_v)$ determines the direction, and $\frac{2\pi}{\sqrt{k_u^2+k_v^2}}$ determines the spatial periodicity
- Problem: too regular, need more randomness
- Solution: add noise



higher frequency



lower frequency

# Perlin Noise

- One could use a random number generator to add noise at every point of the texture
- But this is “white noise” and has no structure
- A smoother and more structured noise is preferable

- So, make a large grid with random numbers on each grid node
- Then, interpolate the randomness to the points inside the gird cells (to get spatial coherency, and structue)

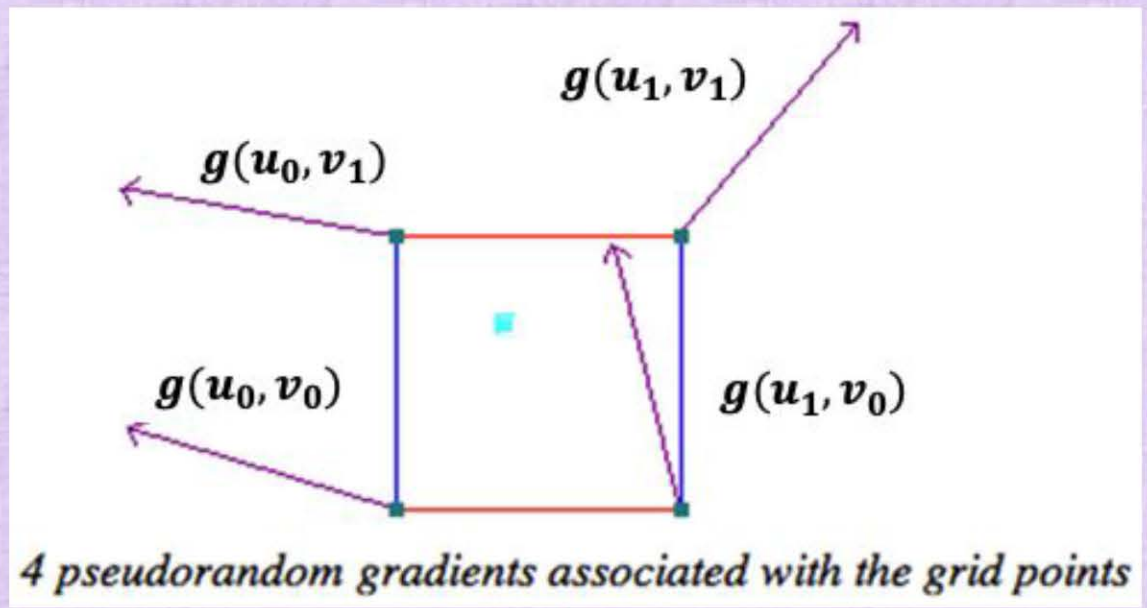
- Ken Perlin proposed a specific (and amazing!) method for doing this

# Perlin Noise

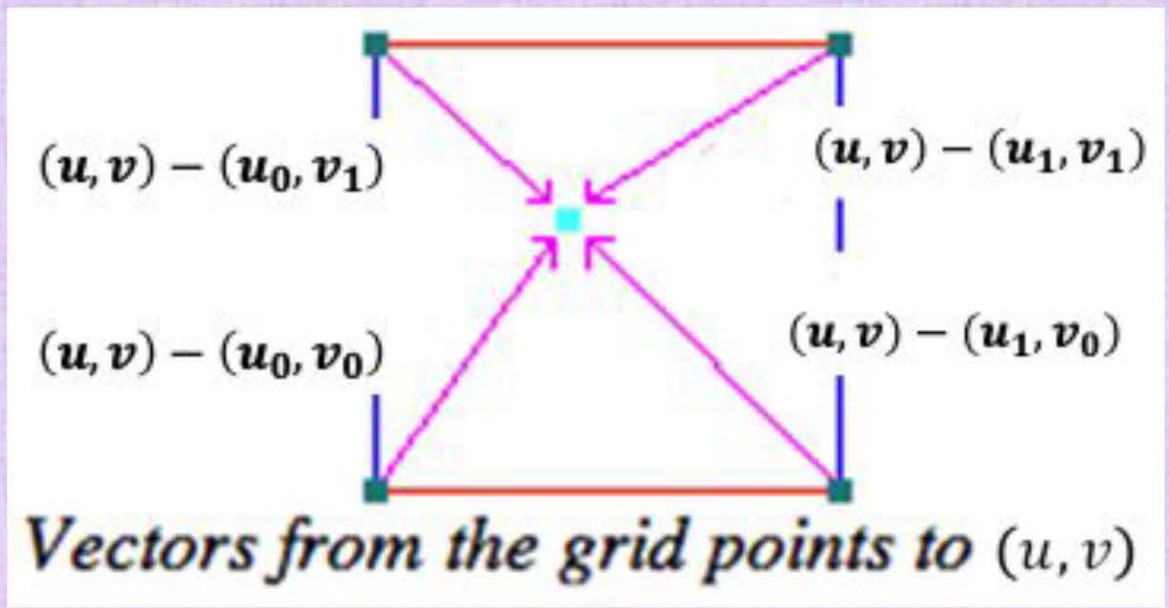
- Lay a 2D grid over the image, and assign a random (unit) gradient to each grid point
- For each pixel, with texture coordinates $(u, v)$, find the grid cell containing it
- Compute a weighted average of the gradients dot-producted with a vector from the corresponding corner
- The noise value is:

$$noise(u,v) = \sum_{i=0,1;j=0,1} w\left(\frac{u-u_i}{\Delta u}\right) w\left(\frac{v-v_j}{\Delta v}\right)\Big(g(u_i,v_j)\cdot\big((u,v)-(u_i,v_j)\big)\Big)$$

- Cubic weighting function: $w(t) = 2|t|^3 - 3|t|^2 + 1$ for $-1 < t < 1$



*4 pseudorandom gradients associated with the grid points*



***Vectors from the grid points to*** $(u, v)$

# Summing Multiple Scales

- Many natural textures contain a variety of feature sizes in the same texture
- Perlin Noise recreates this by adding together noises with different frequencies and amplitudes

$$perlin(u,v) = \sum_k noise\big(frequency(k) * (u,v)\big) * amplitude(k)$$

- Each successive noise function added is called an octave, because it is twice the frequency of the previous one
- I.e., frequencies are chosen via:

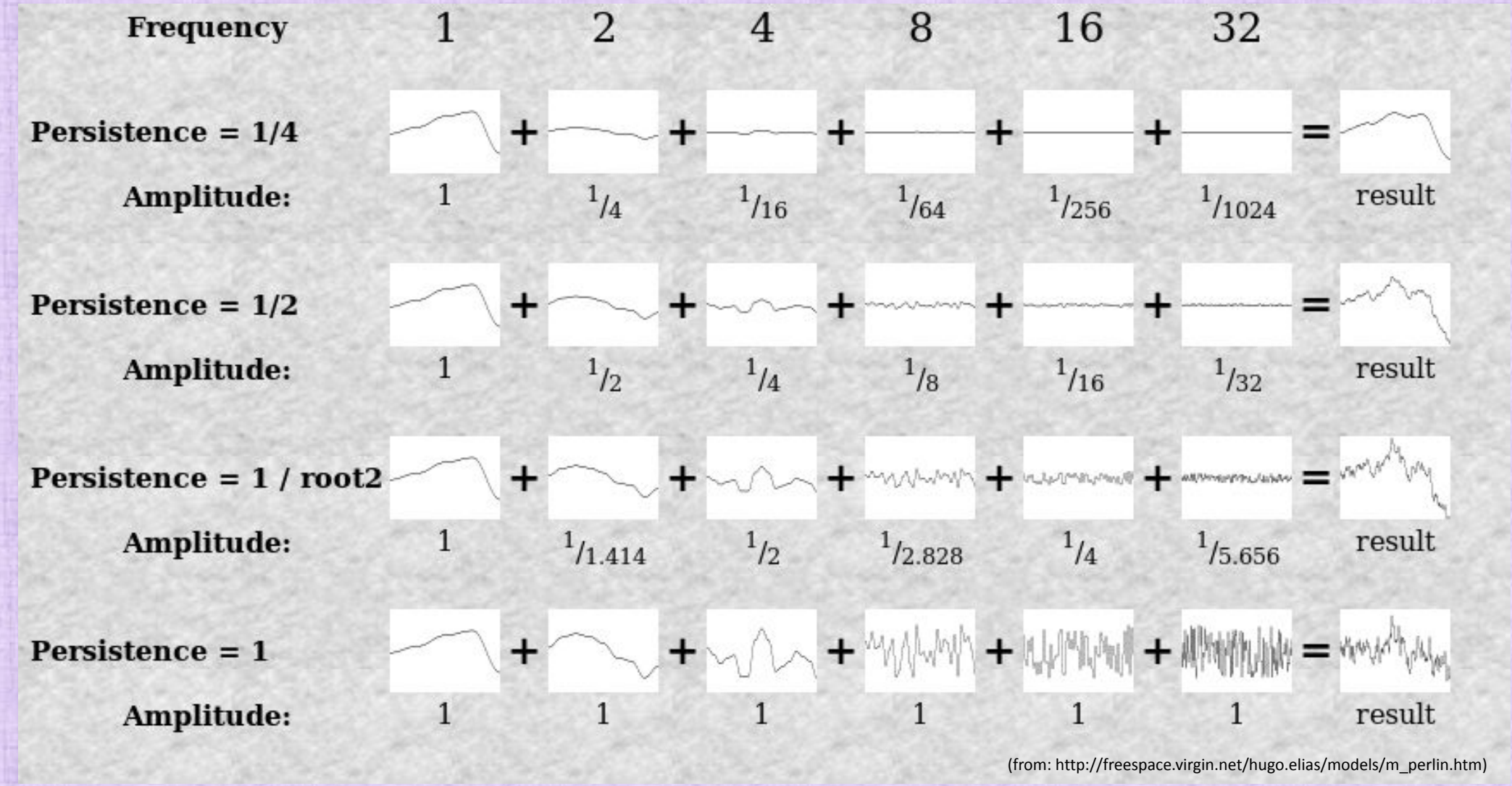
$$frequency(k) = 2^k$$

- The amplitude of higher frequencies is measured by a persistence parameter ($\leq 1$)
- Higher frequencies are given a diminishing contribution:

$$amplitude(k) = persistence^k$$

# Perlin Noise: 1D Example

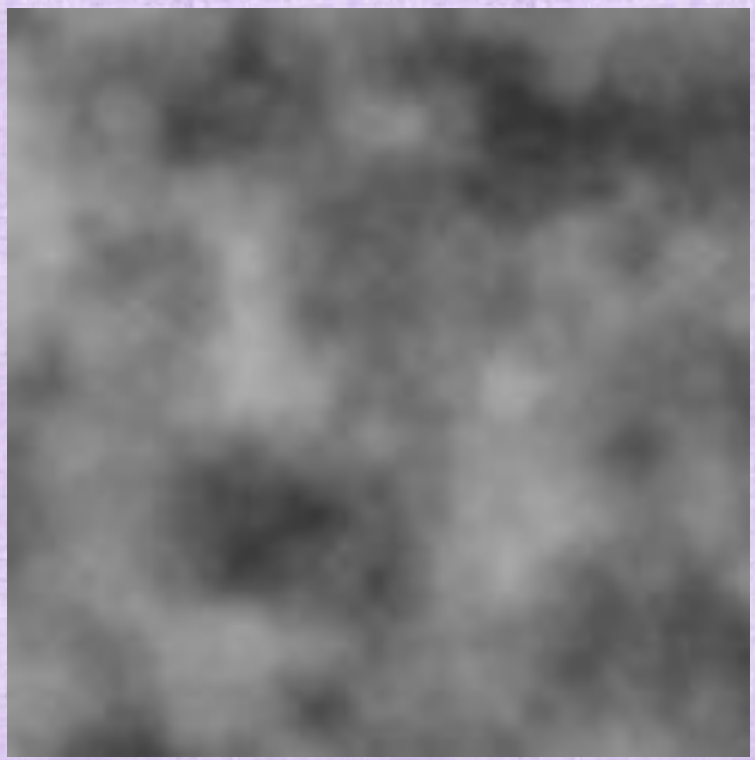
- Smaller persistence = smaller higher frequency noise = smoother result

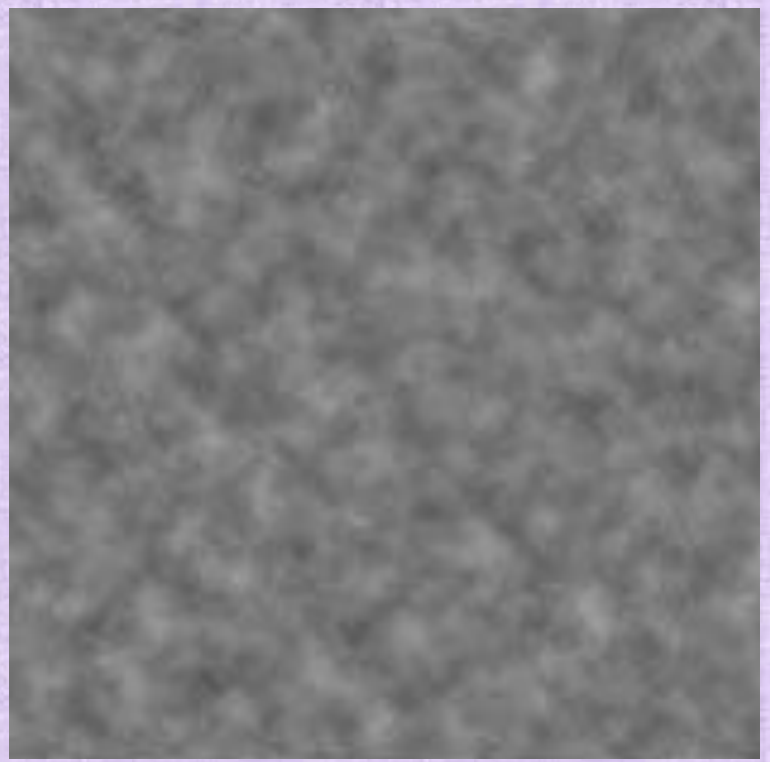


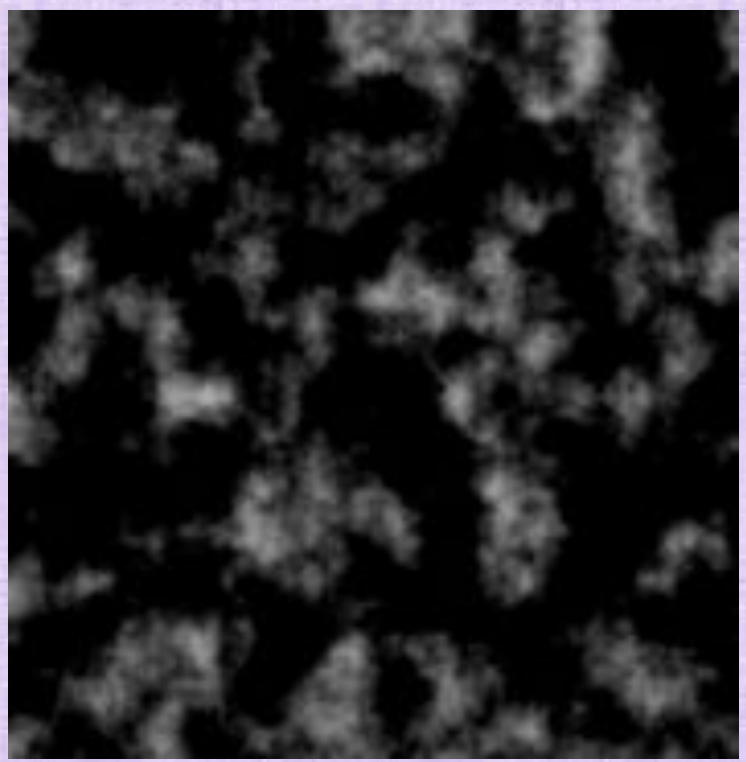

(from: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm)

# Perlin Noise: Examples

- Used to generate height-fields, marble textures, etc.
- Can rescale it and add to itself to get a variety of natural looking maps



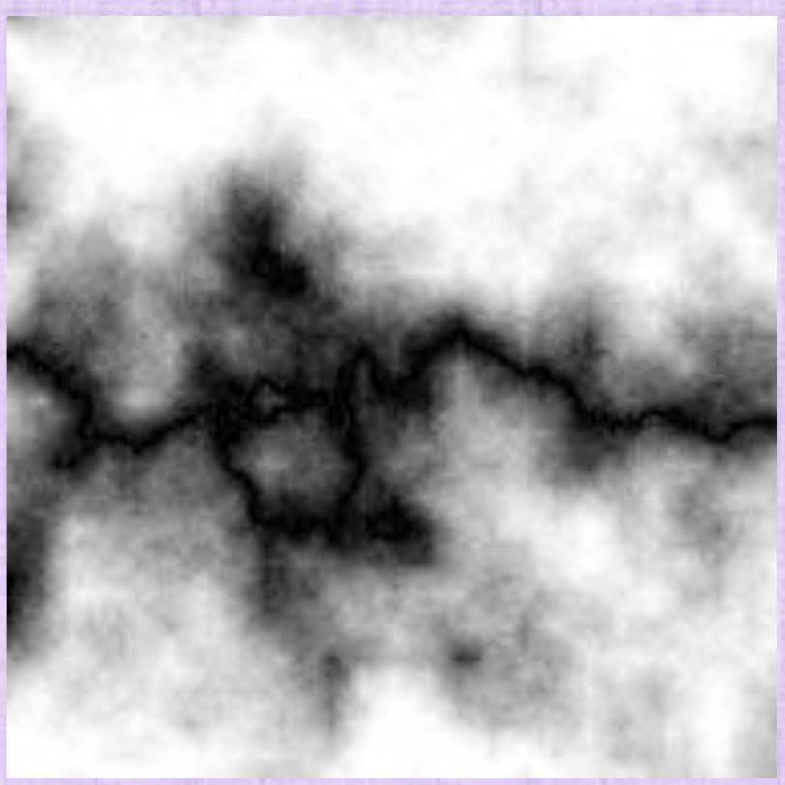


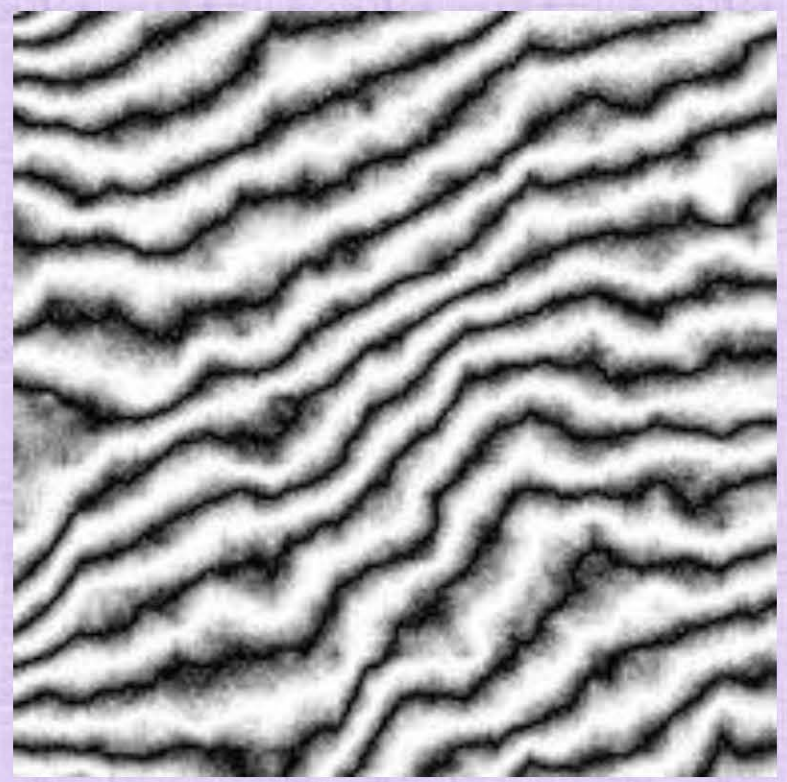


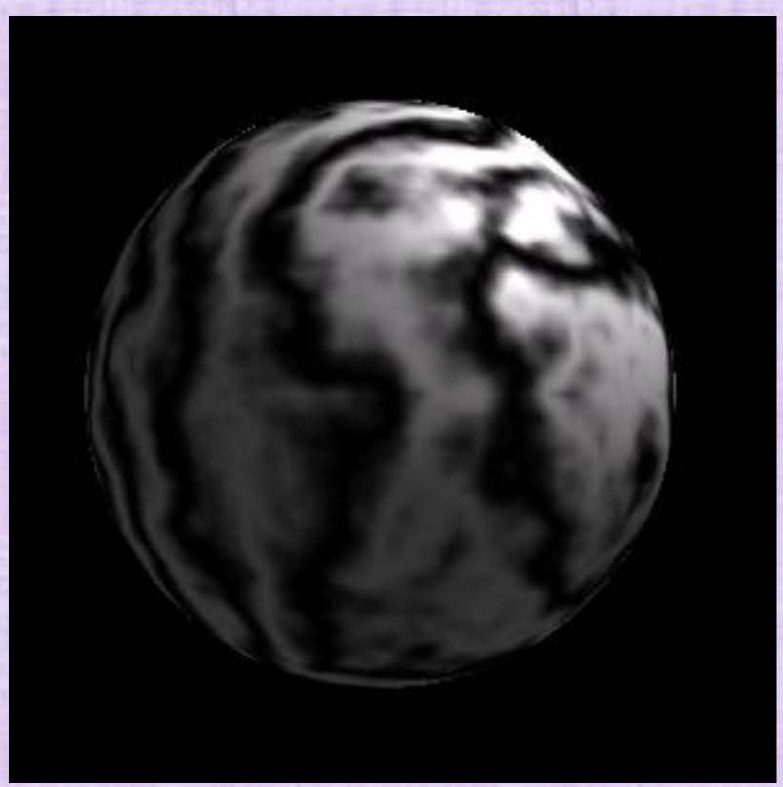
# Marble + Perlin Noise

- Can be scaled and added to the marble texture (set the value of $A$ to scale the noise):

$$marbleColor(u,v) = LayerColor\left(\sin\left(k_u u + k_v v + A * perlin(u,v)\right)\right)$$

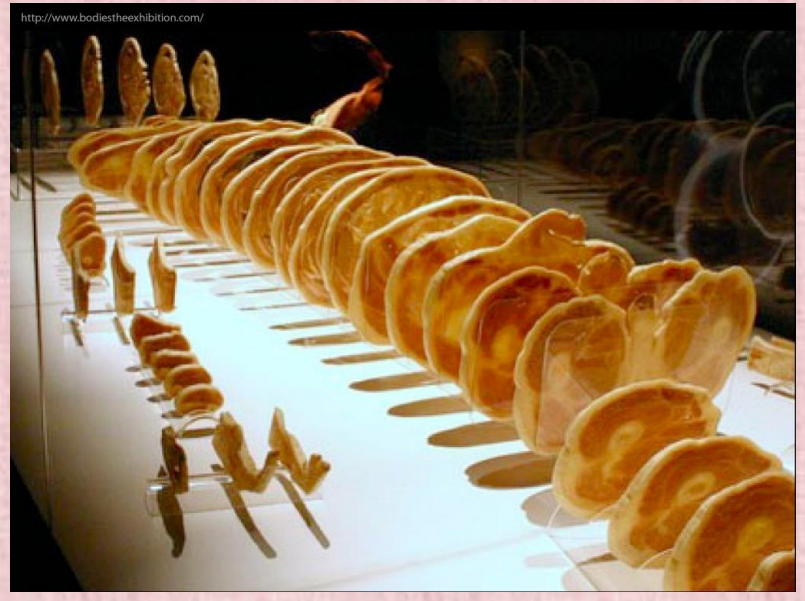


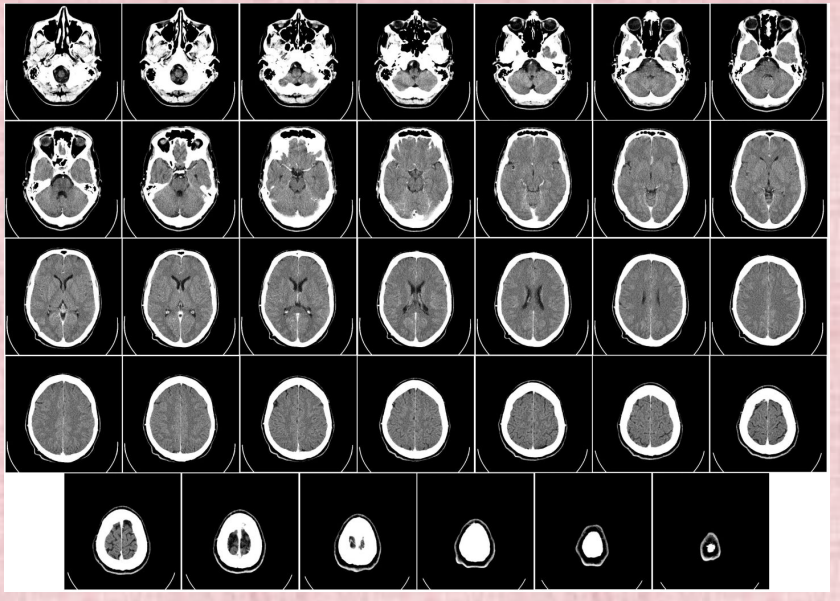




# 3D Textures

- Typically generated procedurally, since 3D images are rare
- Although, one could slice up a 3D object and take a bunch of 2D pictures to make a 3D texture
- Or use some sort of 3D imaging technology

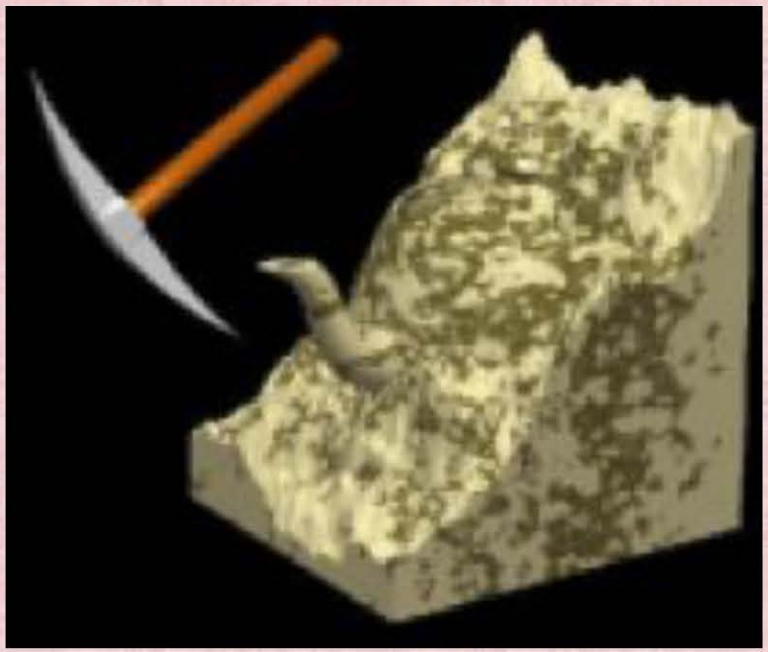


# 3D Marble Texture

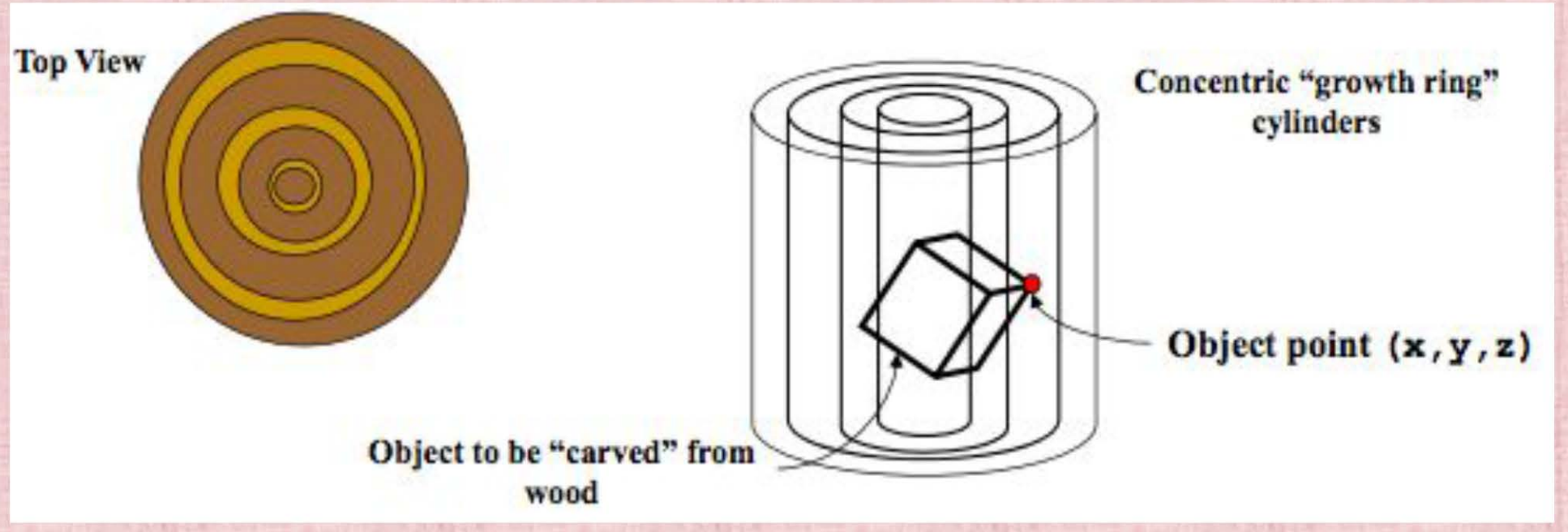
- Generate a 3D texture representing the material, and "carve" the object out of this 3D texture
- Eliminates the difficulty of wrapping a 2D texture over a complex 3D object
- Marble texture function with Perlin noise for a 3D texture:

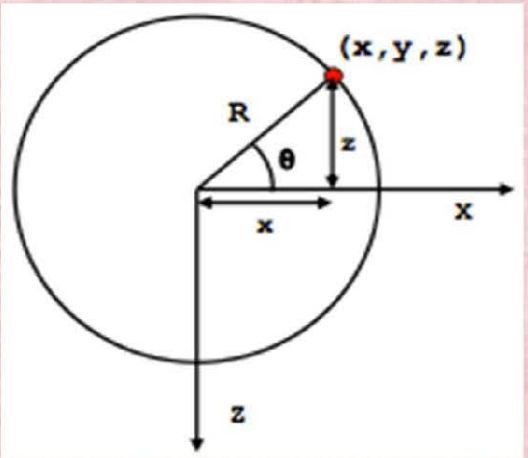
$$marbleColor(u,v,w) = LayerColor\left(\sin\left(k_u u + k_v v + k_w w + A * perlin(u,v,w)\right)\right)$$



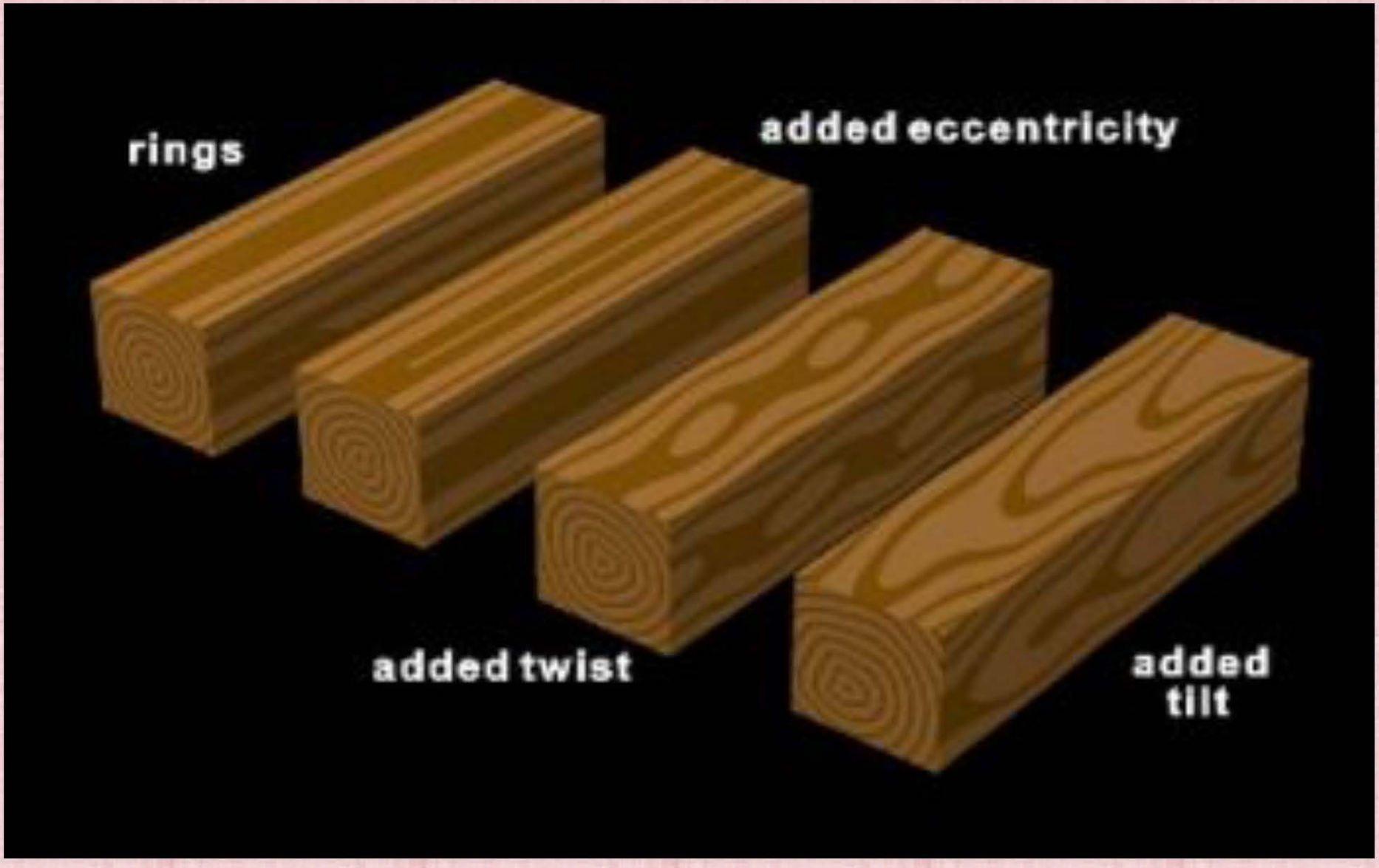
# 3D Wood Texture

- Generated by tree rings
- Compute cylindrical coordinates for $(x, y, z)$ object points: $H = y$, $R = \sqrt{x^2 + z^2}$, $\theta = \tan^{-1}\left(\frac{z}{x}\right)$



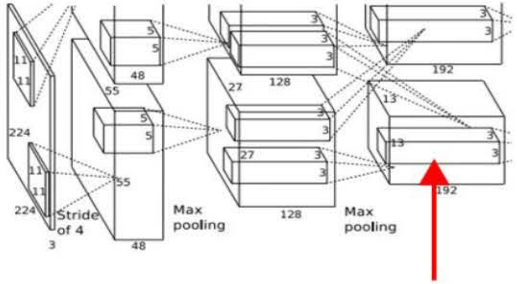
# 3D Wood Texture

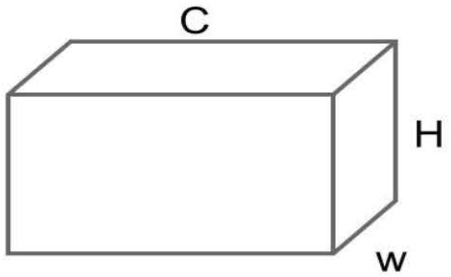


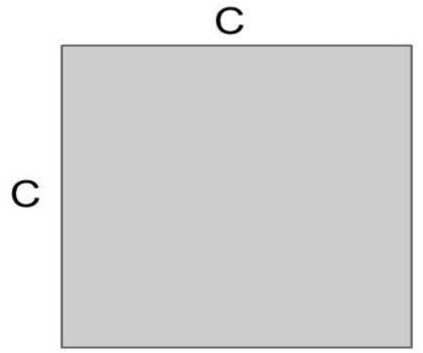
# Machine Learning Approaches

## Neural Texture Synthesis: Gram Matrix

This image is in the public domain.




Each layer of CNN gives C x H x W tensor of features; H x W grid of C-dimensional vectors

Outer product of two C-dimensional vectors gives C x C matrix measuring co-occurrence

Average over all HW pairs of vectors, giving **Gram matrix** of shape C x C

Efficient to compute; reshape features from

C x H x W to =C x HW

then compute $G = FF^T$

# Machine Learning Approaches

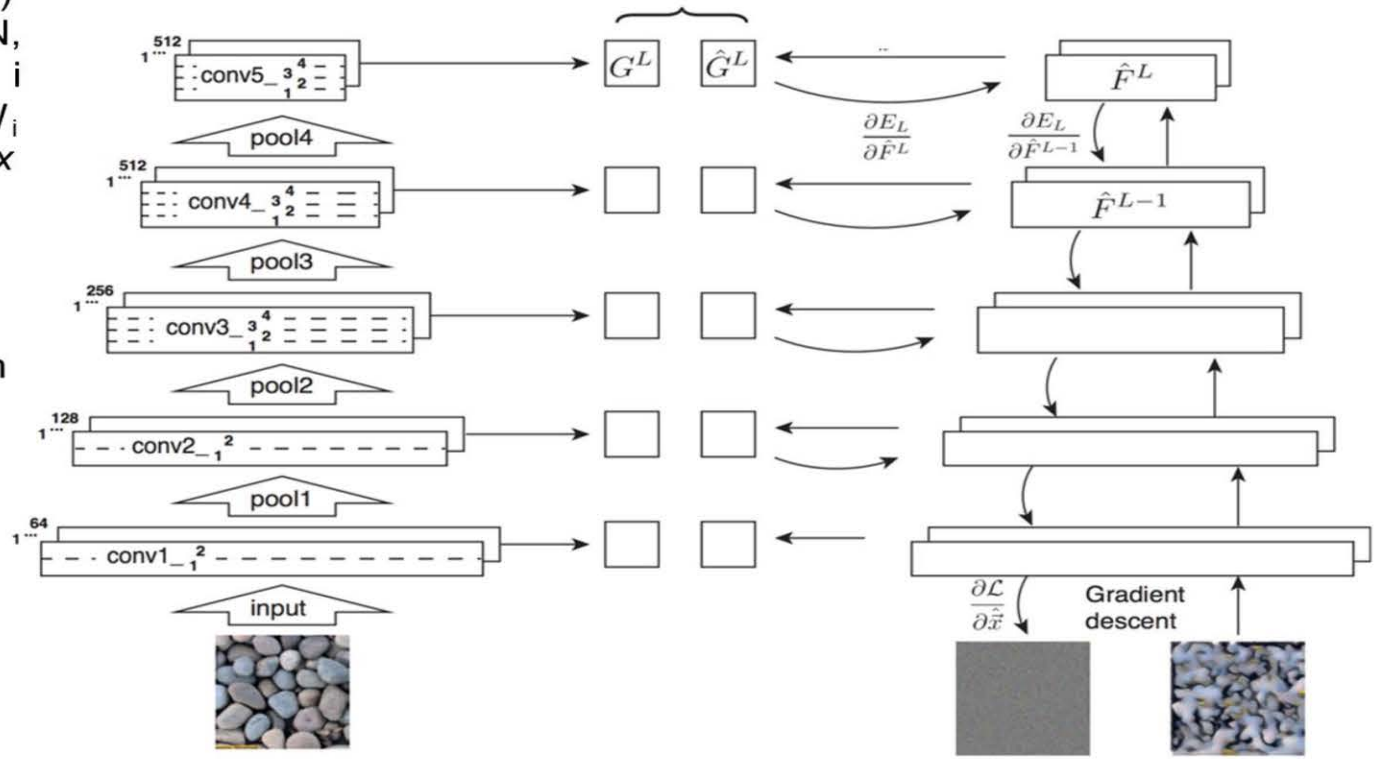
## Neural Texture Synthesis

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^l - \hat{G}_{ij}^l\right)^2 \qquad \mathcal{L}(\vec{x}, \hat{\vec{x}}) = \sum_{l=0}^{L} w_l E_l$$

1. Pretrain a CNN on ImageNet (VGG-19)
2. Run input texture forward through CNN, record activations on every layer; layer i gives feature map of shape $C_i \times H_i \times W_i$
3. At each layer compute the *Gram matrix* giving outer product of features:

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l \text{ (shape } C_i \times C_i)$$

4. Initialize generated image from random noise
5. Pass generated image through CNN, compute Gram matrix on each layer
6. Compute loss: weighted sum of L2 distance between Gram matrices
7. Backprop to get gradient on image
8. Make gradient step on image
9. GOTO 5




Gatys, Ecker, and Bethge, "Texture Synthesis Using Convolutional Neural Networks", NIPS 2015
Figure copyright Leon Gatys, Alexander S. Ecker, and Matthias Bethge, 2015. Reproduced with permission.

# Machine Learning Approaches

## Neural Texture Synthesis

Reconstructing texture from higher layers recovers larger features from the input texture


Gatys, Ecker, and Bethge, "Texture Synthesis Using Convolutional Neural Networks", NIPS 2015
Figure copyright Leon Gatys, Alexander S. Ecker, and Matthias Bethge, 2015. Reproduced with permission.