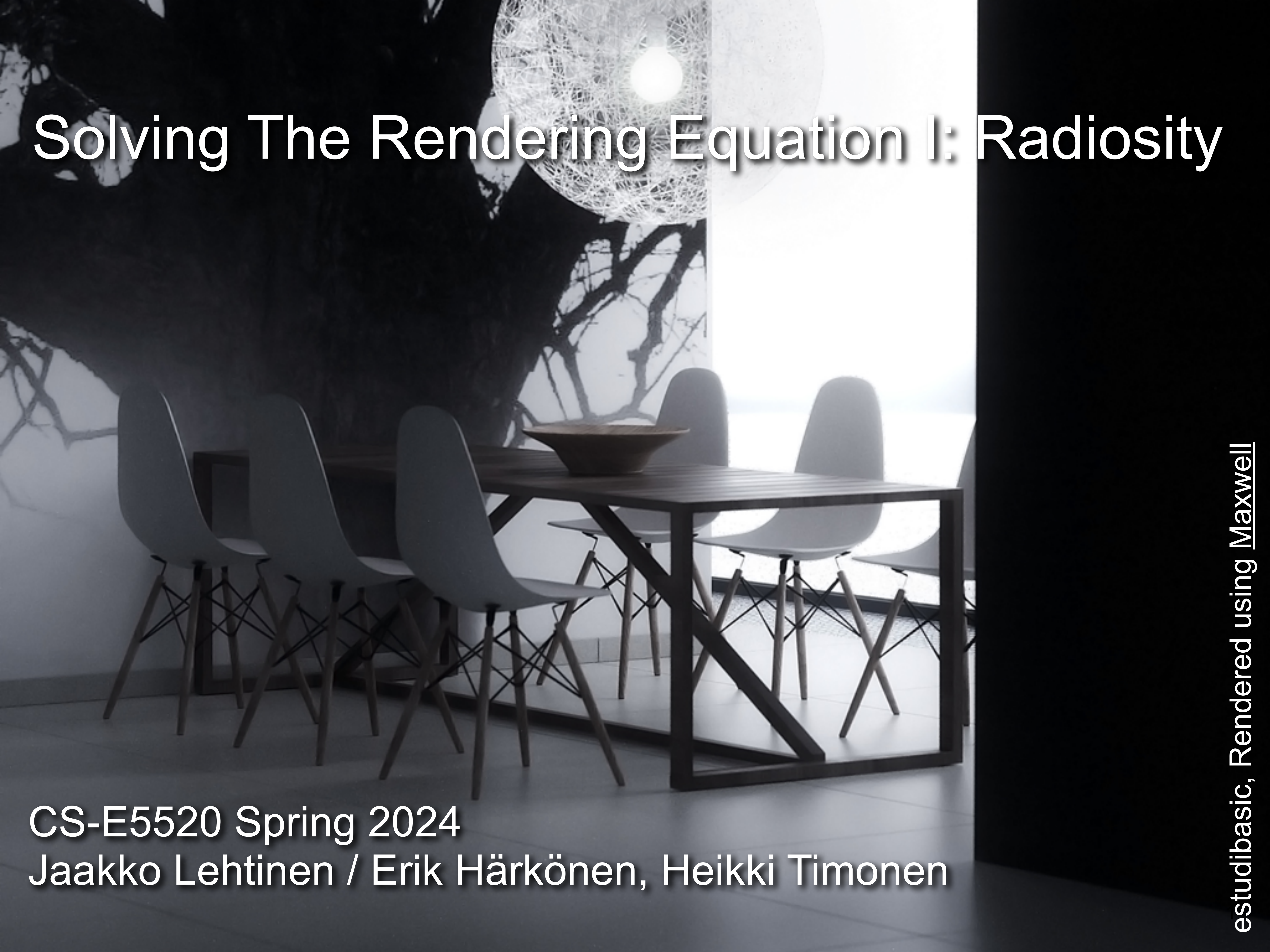
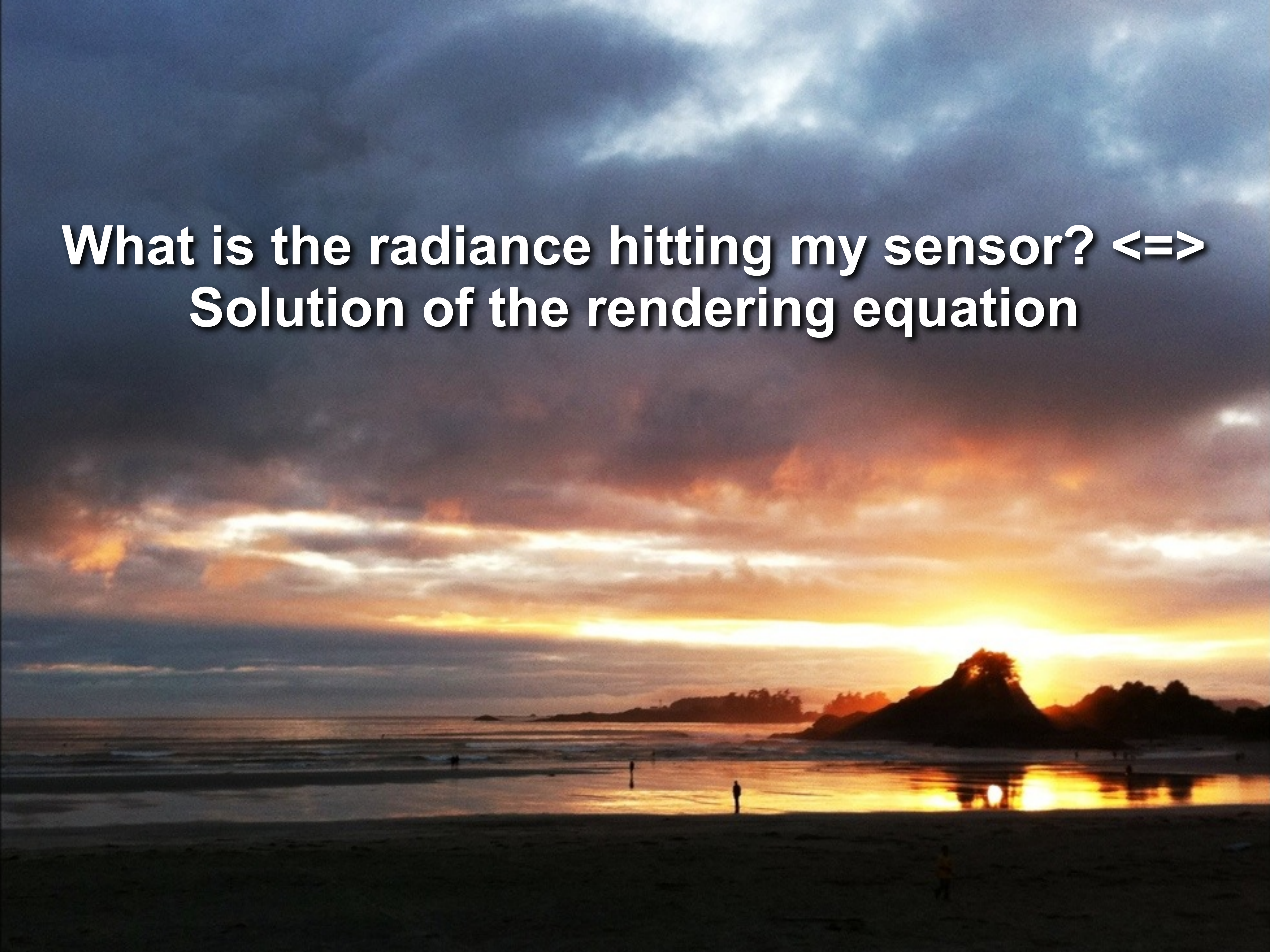
Solving The Rendering Equation I: Radiosity



estudibasic, Rendered using Maxwell

CS-E5520 Spring 2024
Jaakko Lehtinen / Erik Härkönen, Heikki Timonen

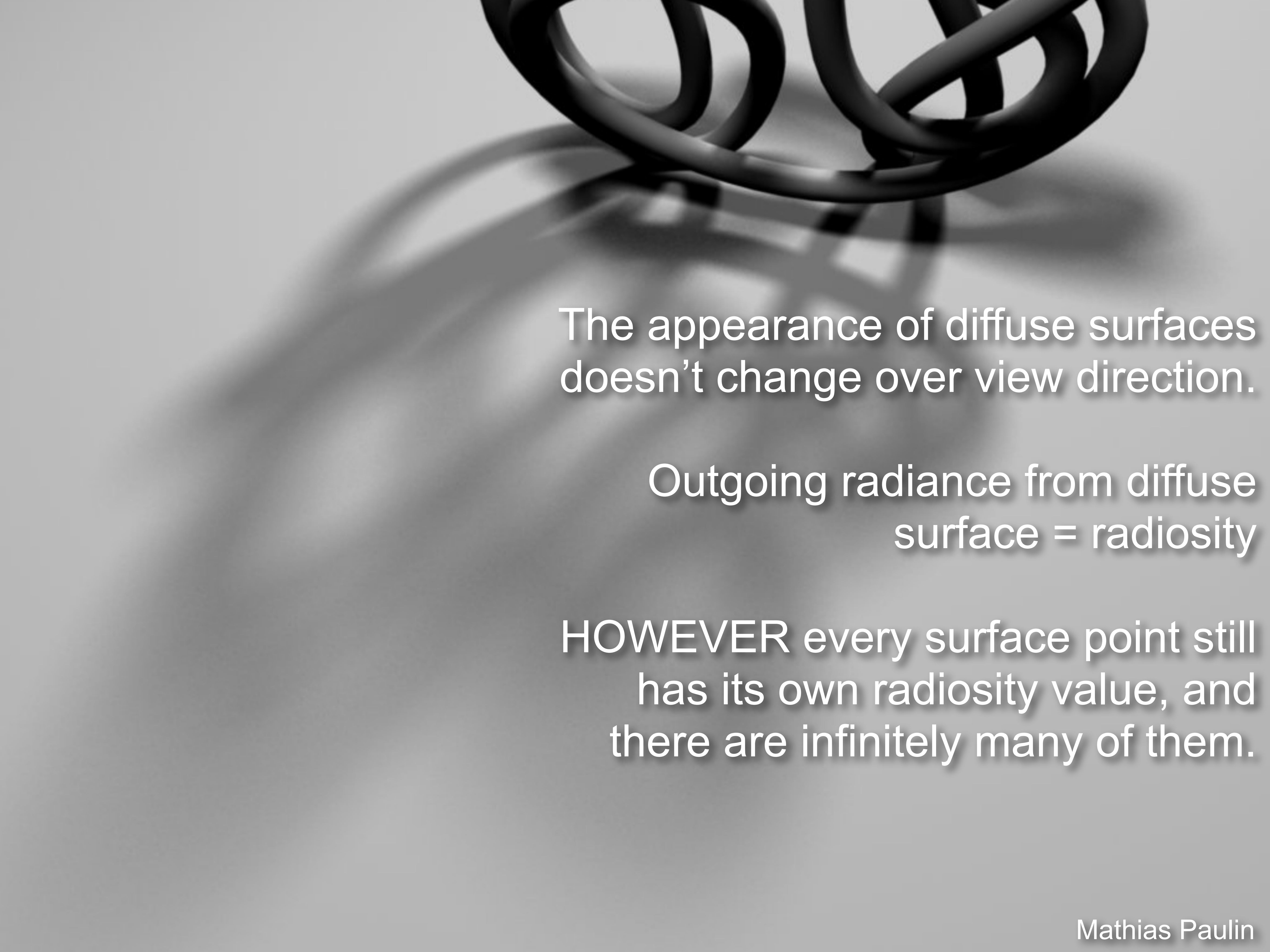


What is the radiance hitting my sensor? <=> Solution of the rendering equation

Today

- Discretizing the rendering equation
  - Radiosity (topic of your assignment!)

Mathias Paulin

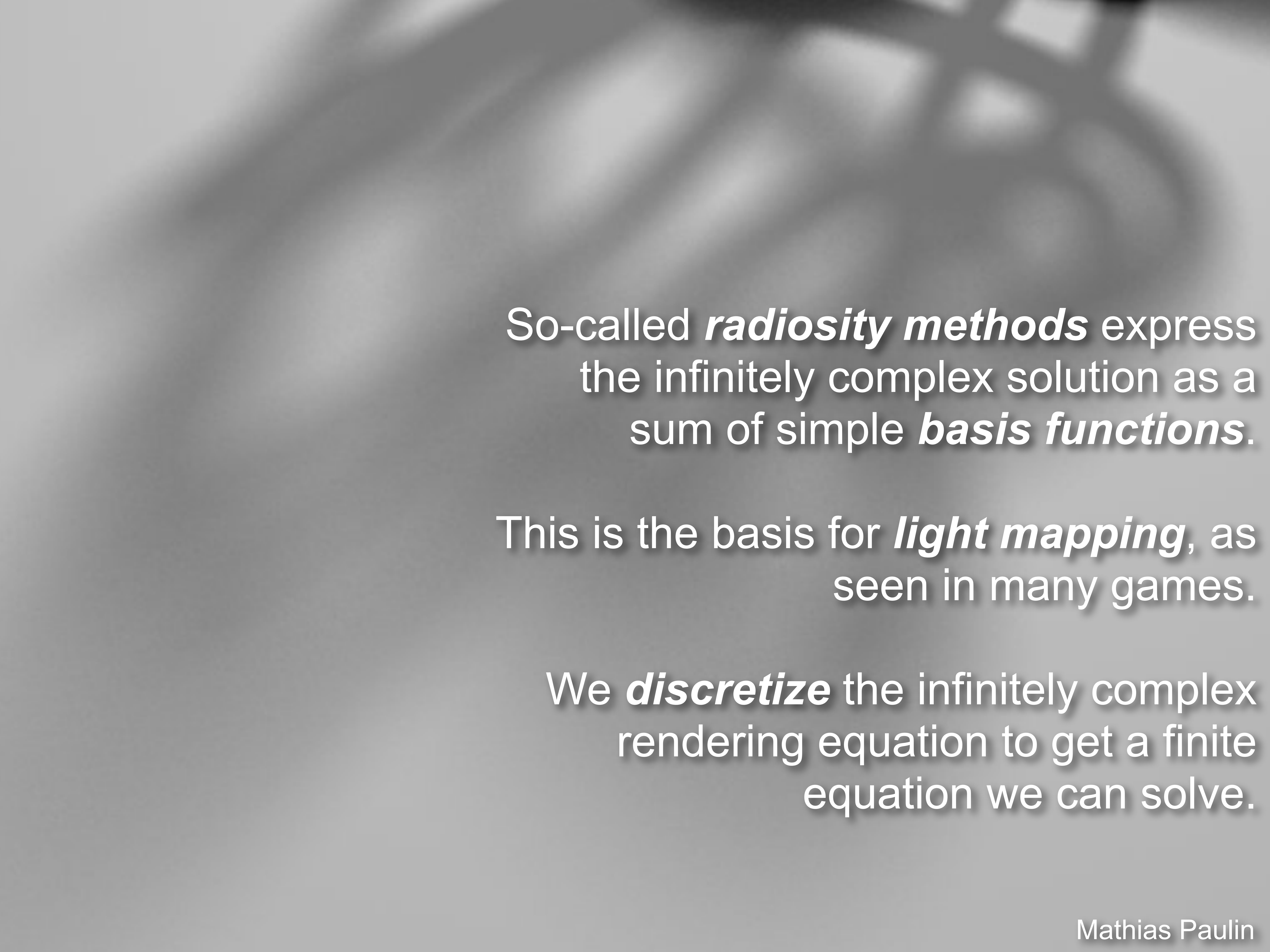


The appearance of diffuse surfaces doesn’t change over view direction.

Outgoing radiance from diffuse surface = radiosity

HOWEVER every surface point still has its own radiosity value, and there are infinitely many of them.

Mathias Paulin



So-called ***radiosity methods*** express the infinitely complex solution as a sum of simple ***basis functions***.

This is the basis for ***light mapping***, as seen in many games.

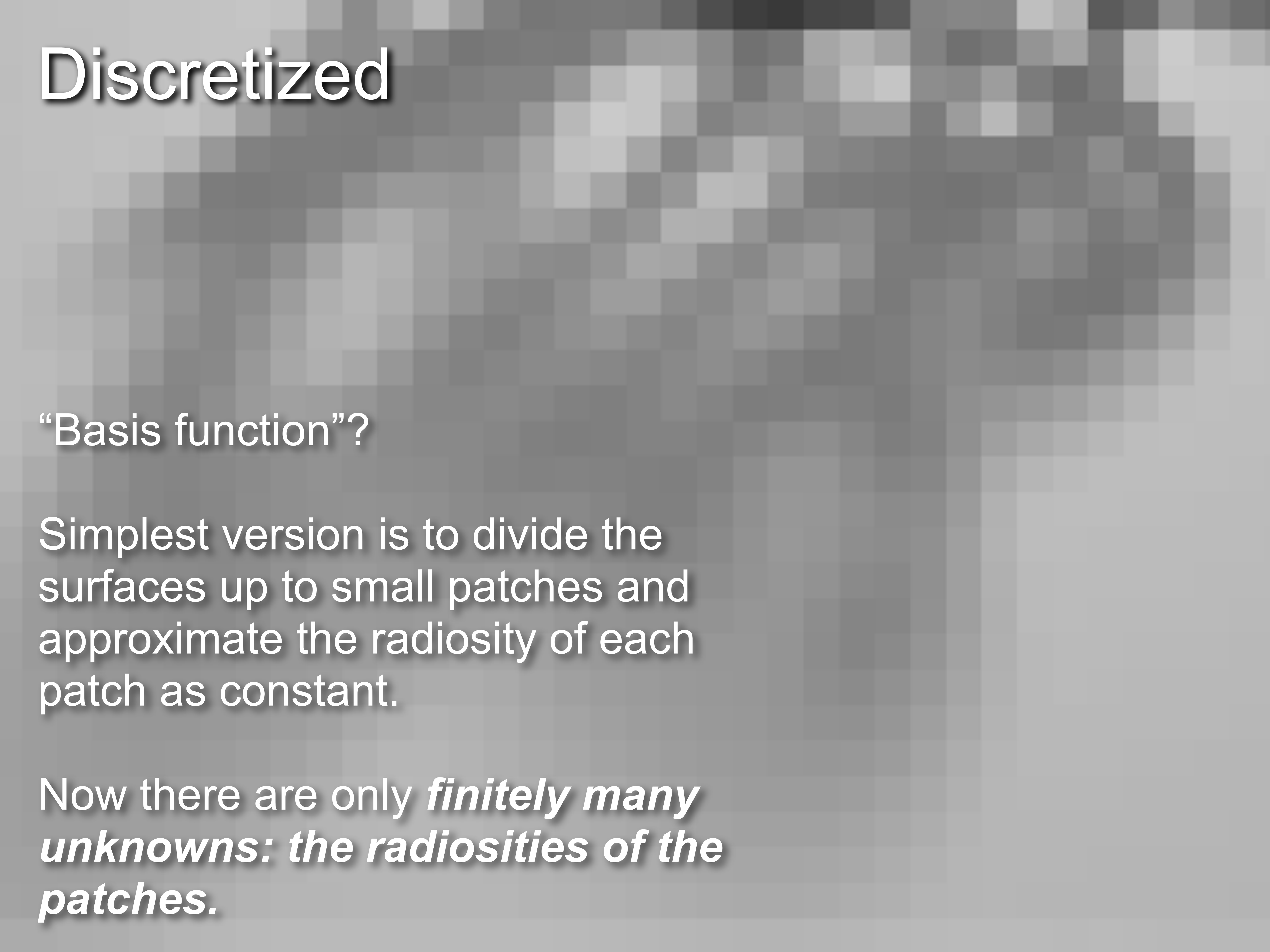
We ***discretize*** the infinitely complex rendering equation to get a finite equation we can solve.

Mathias Paulin

Continuous

Discretized

Discretized

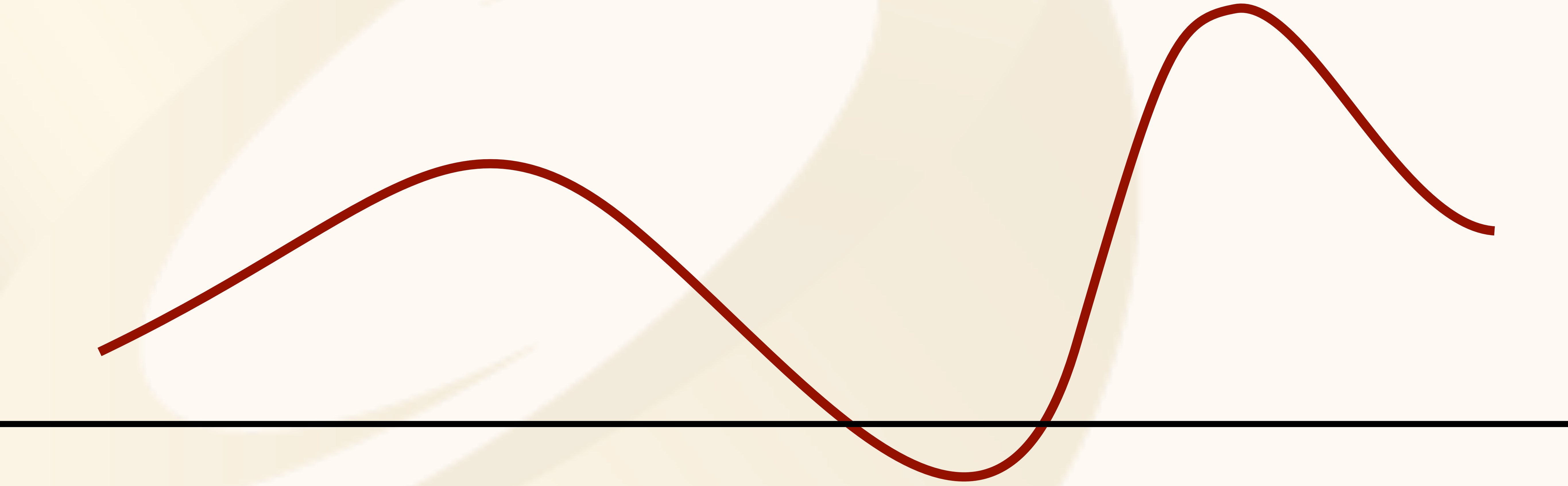


“Basis function”?

Simplest version is to divide the surfaces up to small patches and approximate the radiosity of each patch as constant.

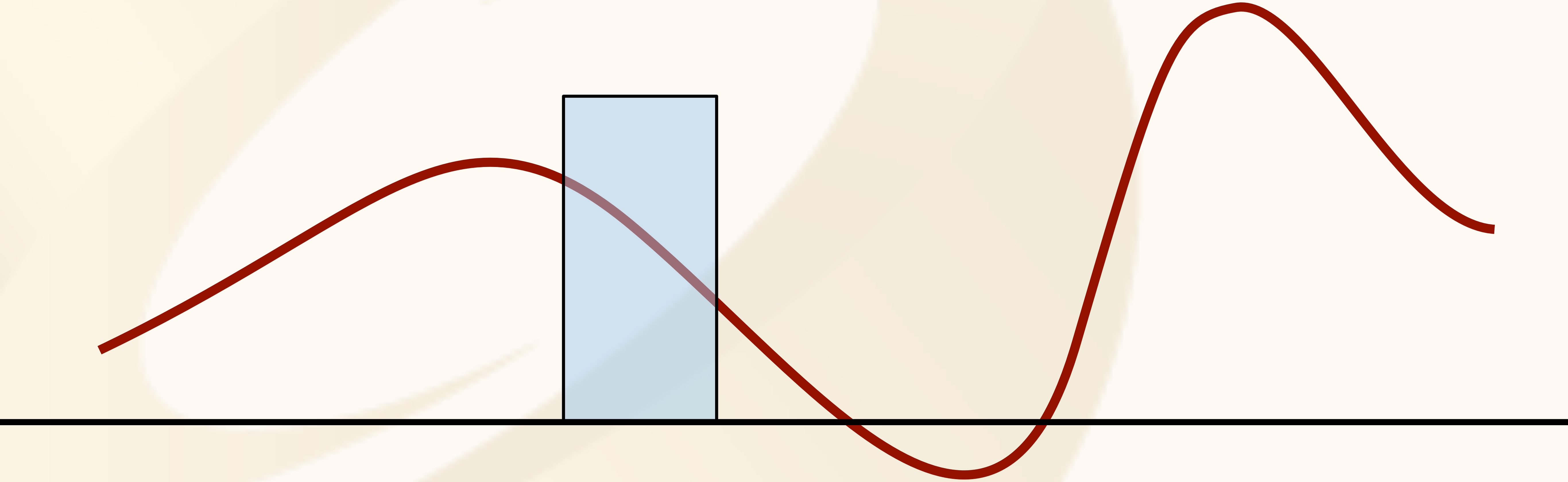
Now there are only ***finitely many unknowns: the radiosities of the patches.***

Some Function on a Continuous Domain



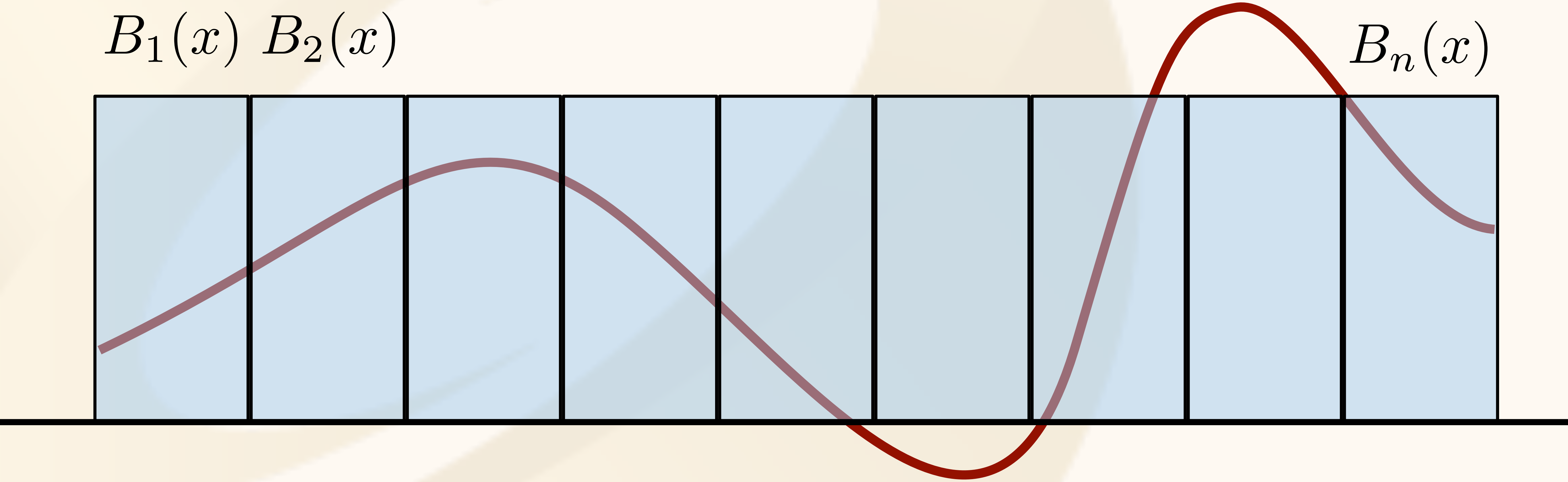
Unweighted Basis Functions

- Here each basis function is a box, translated so that they don't overlap



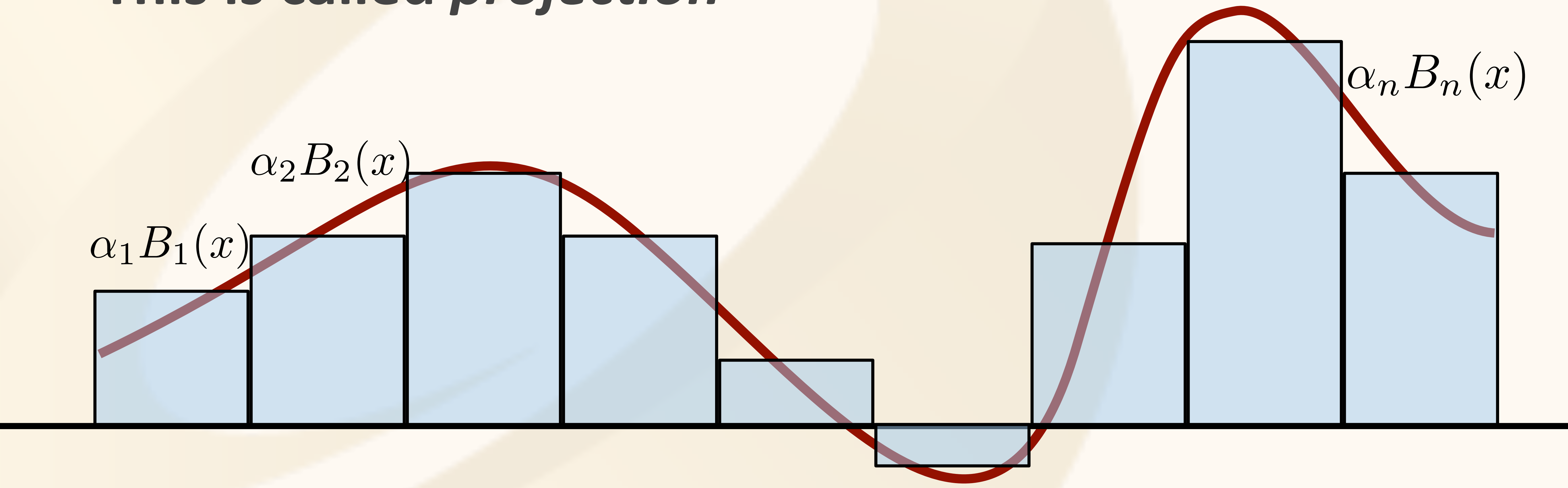
Unweighted Basis Functions

- Here each basis function is a box, translated so that they don't overlap

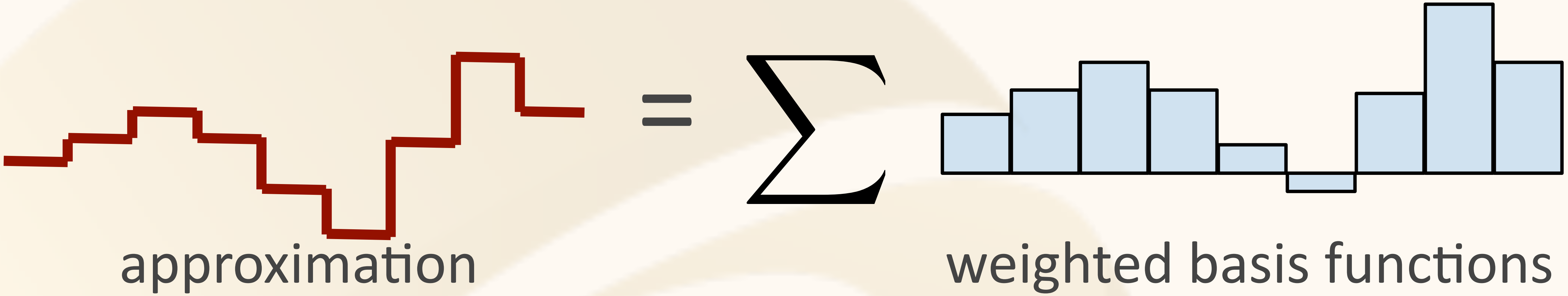


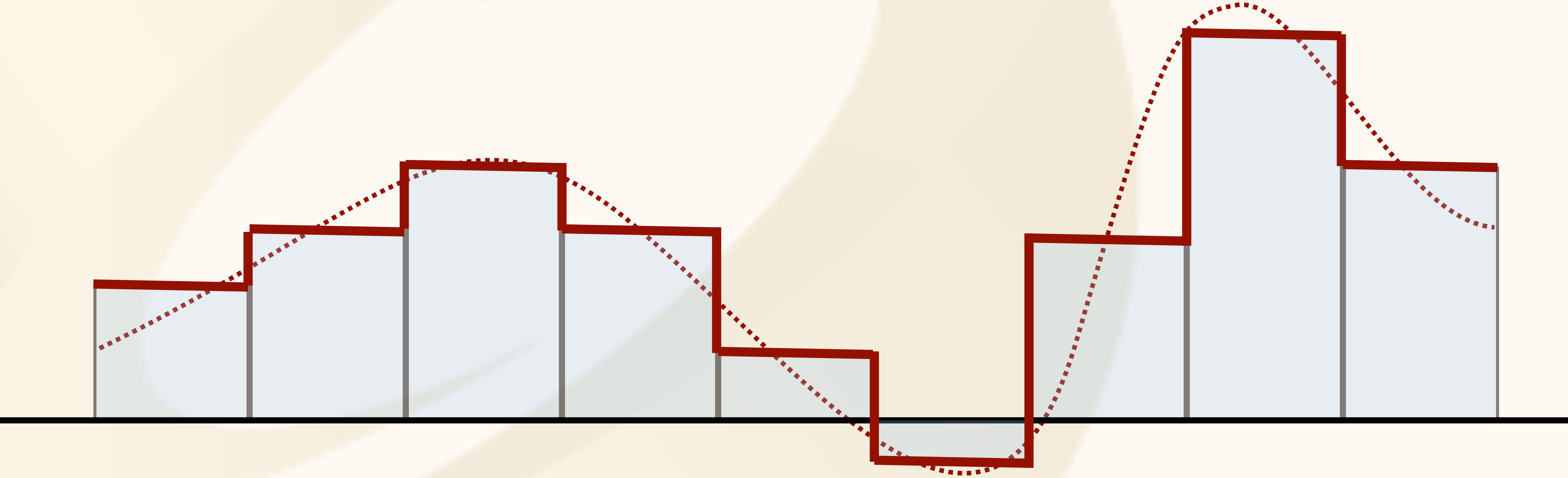
Approximation by Basis Functions

- We can try to choose weights for the basis functions such that together the boxes approximate the input function well
- This is called *projection*

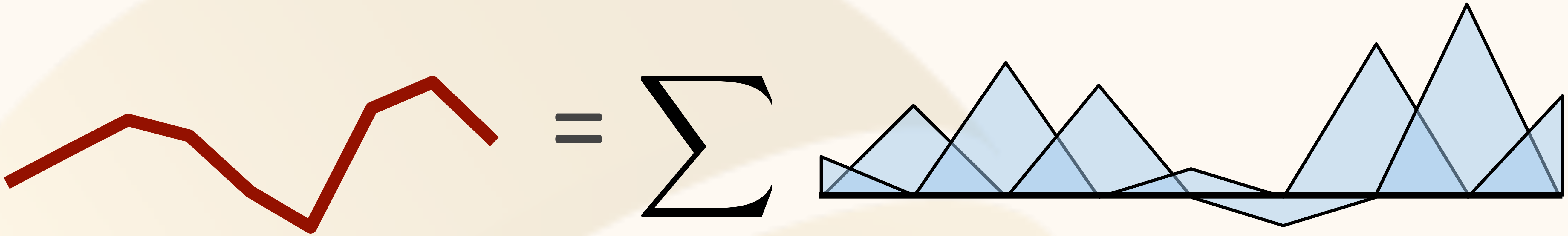


"Projection onto Finite Basis"



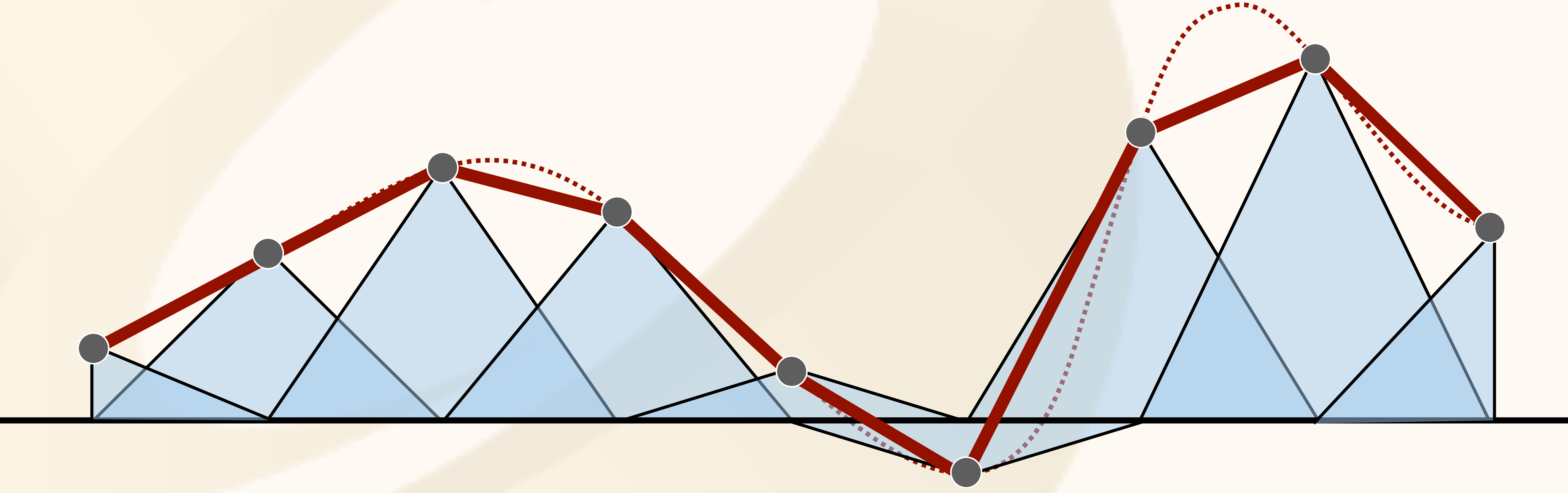


Projection onto Finite Basis, Piecewise Linear



approximation

weighted basis functions



Fourier Series is the Same Thing

(Formula for interval of length 2pi;
Weights are the A_0, A_i, B_i)

$$s_N = A_0 + \sum_{i=1}^{N} \left(A_i \cos(nx) \; + \; B_i \sin(nx) \right)$$

weighted basis functions=scaled sines and cosines

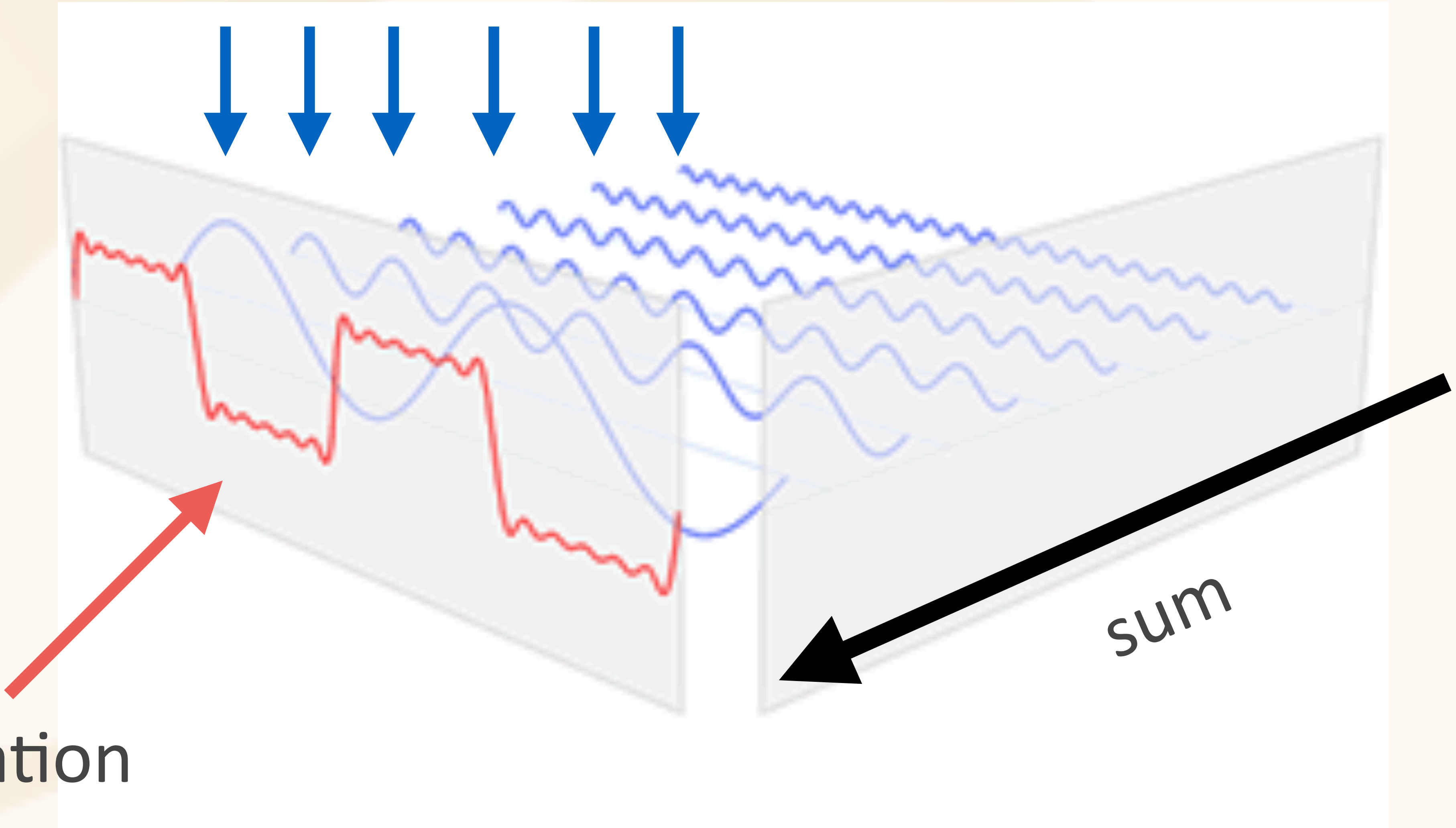
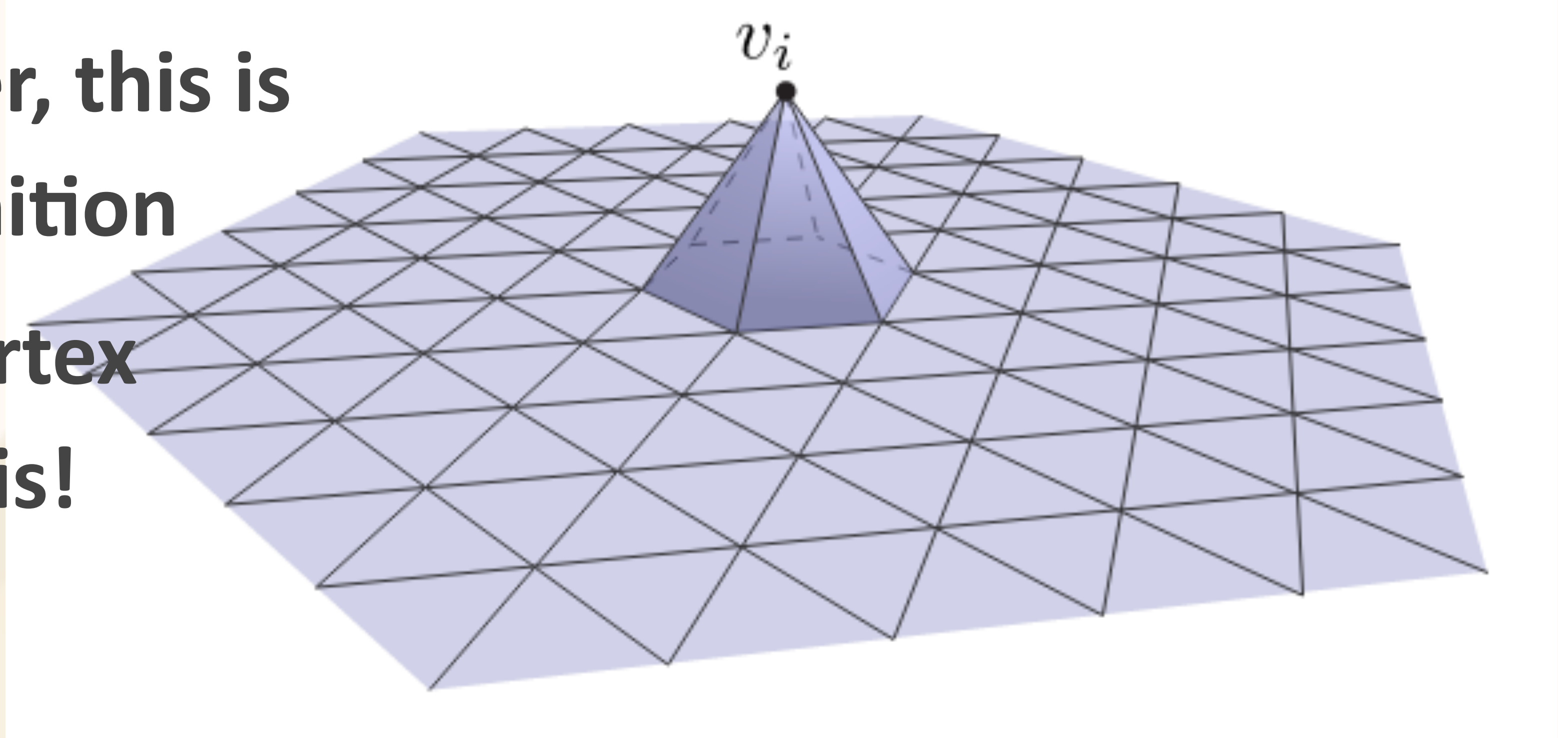


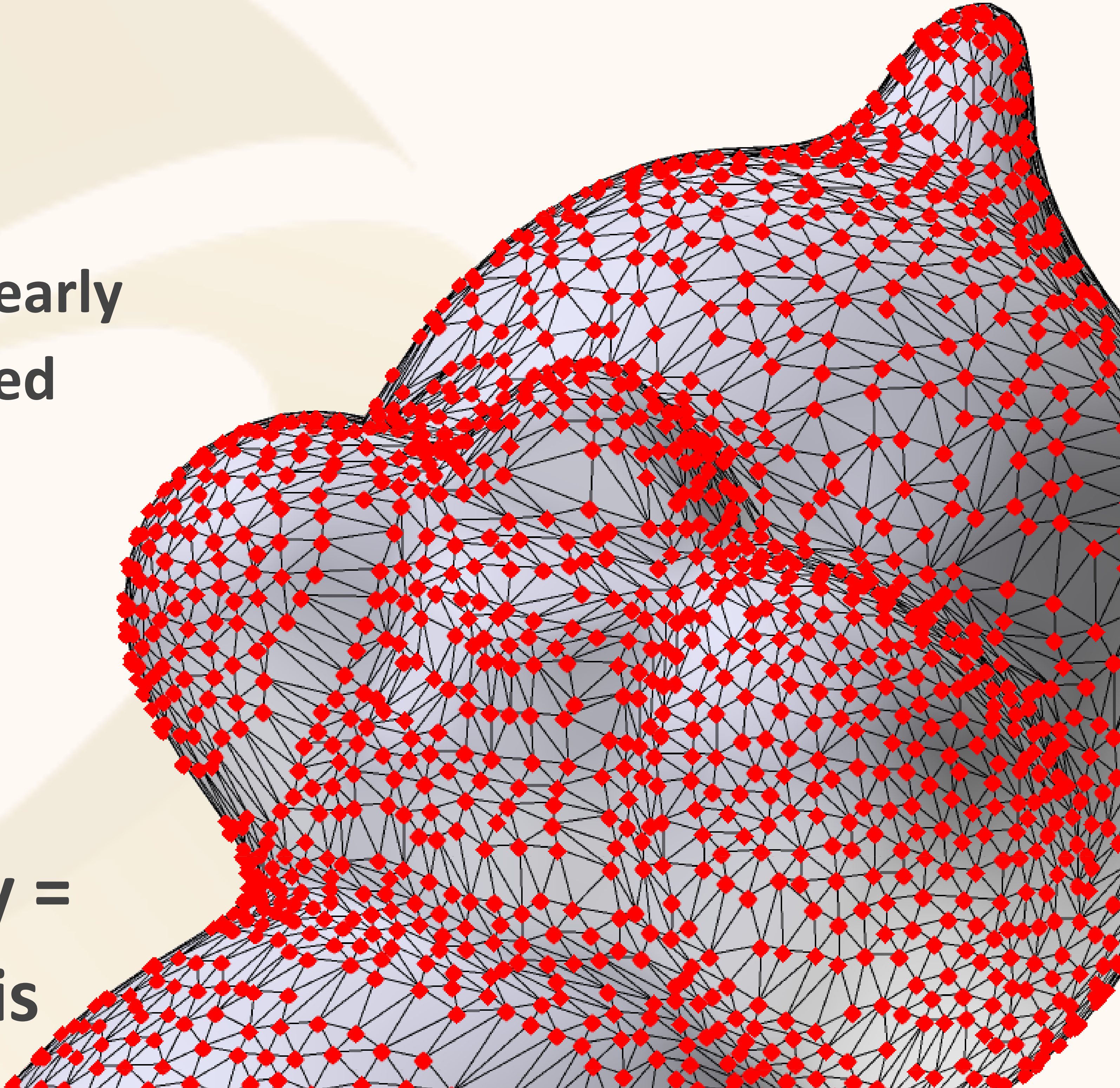
Figure adapted from wikipedia

Piecewise Linear Basis Functions

- Each vertex has one basis function
  - 1 at the vertex, falls linearly to 0 inside the connected triangles
  - Easy to evaluate using barycentrics: remember, this is pretty much their definition
  - But remember each vertex affects all connected tris!



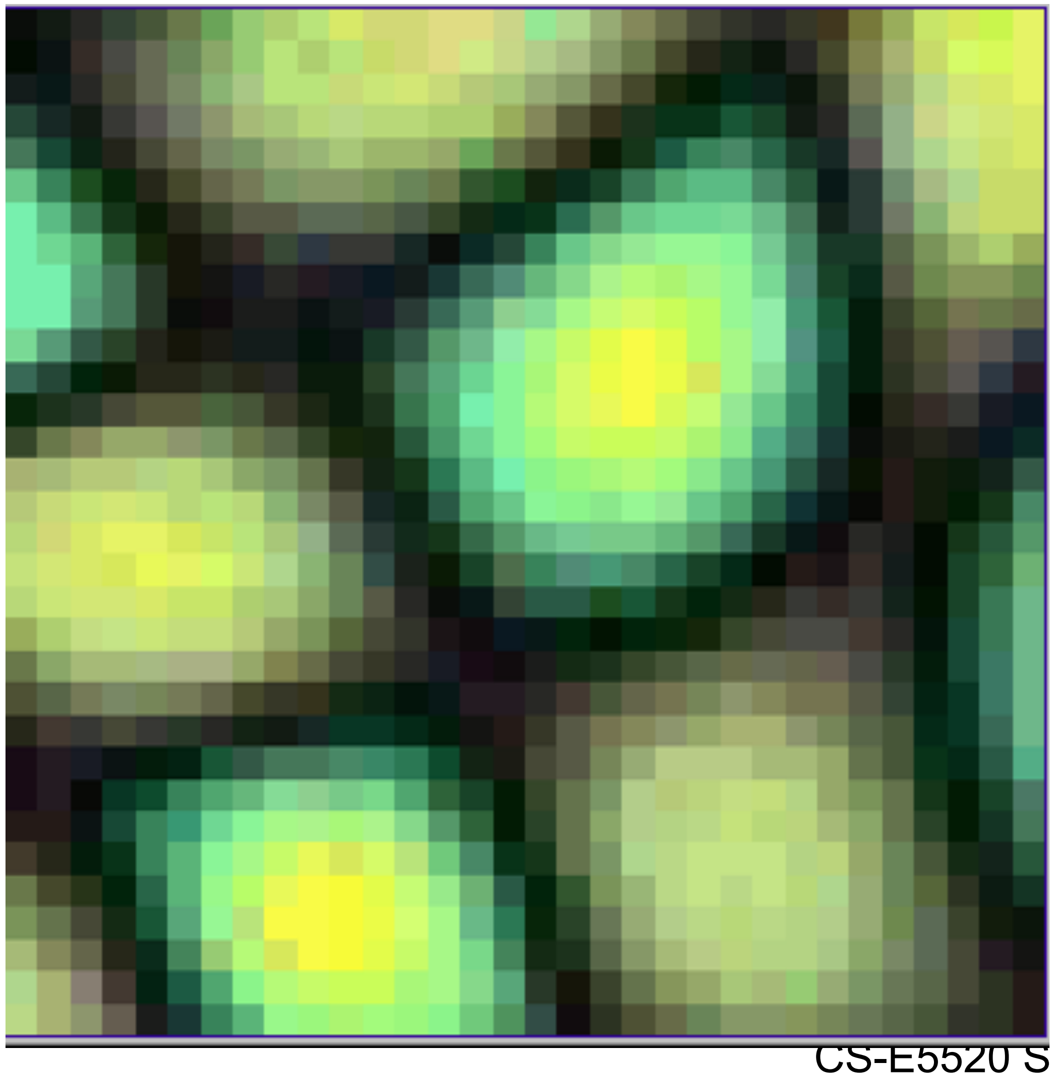
Piecewise Linear Basis Functions

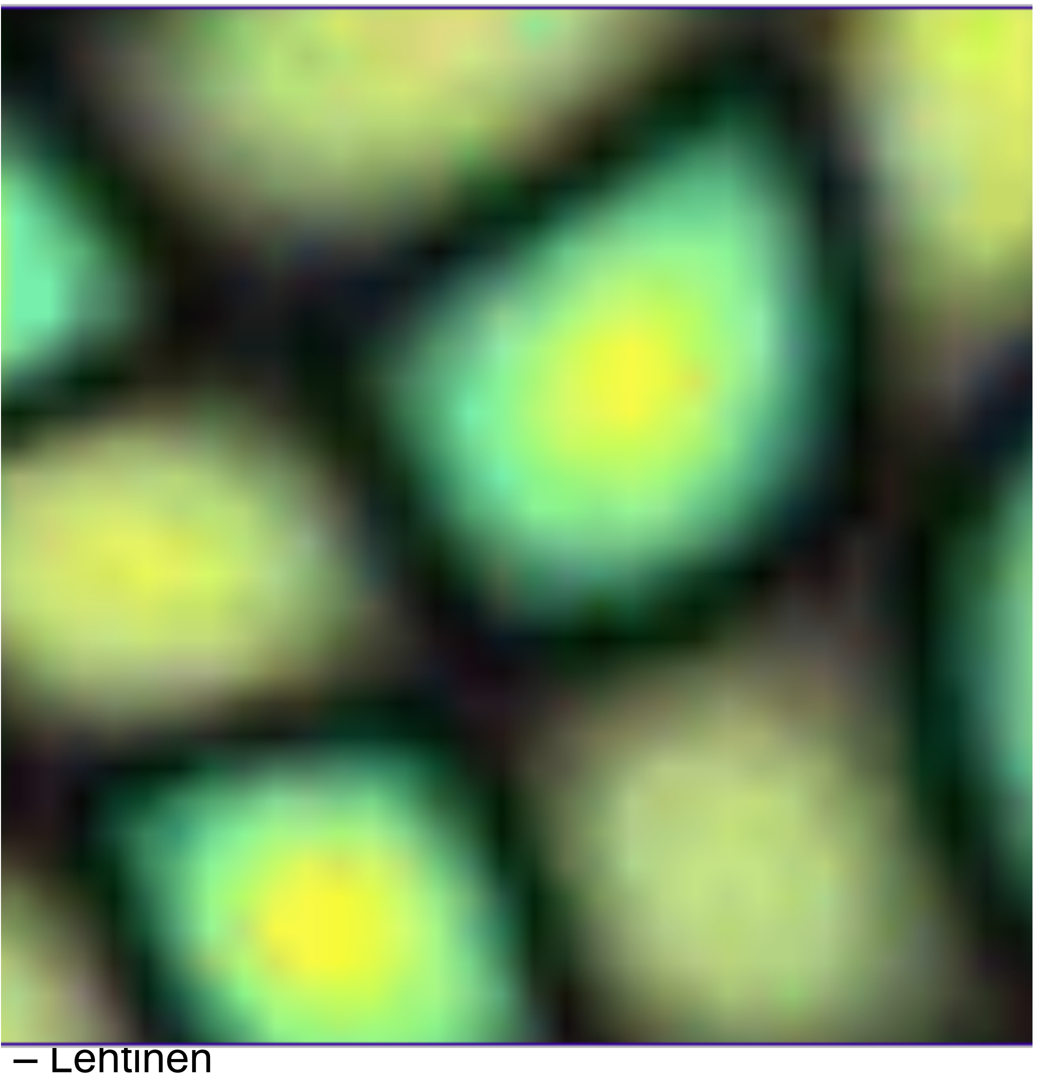


- Each vertex has one basis function
  - 1 at the vertex, falls linearly to 0 inside the connected triangles
  - Barycentrics!
- Sampling values at vertices and interpolating linearly = piecewise linear basis

Flashback: Bilinear Texture Filtering

- Tell OpenGL to use a tent filter instead of a box filter
- Magnification looks better, but blurry
  - (texture is under-sampled for this resolution)
  - Oh well...





Texture Maps

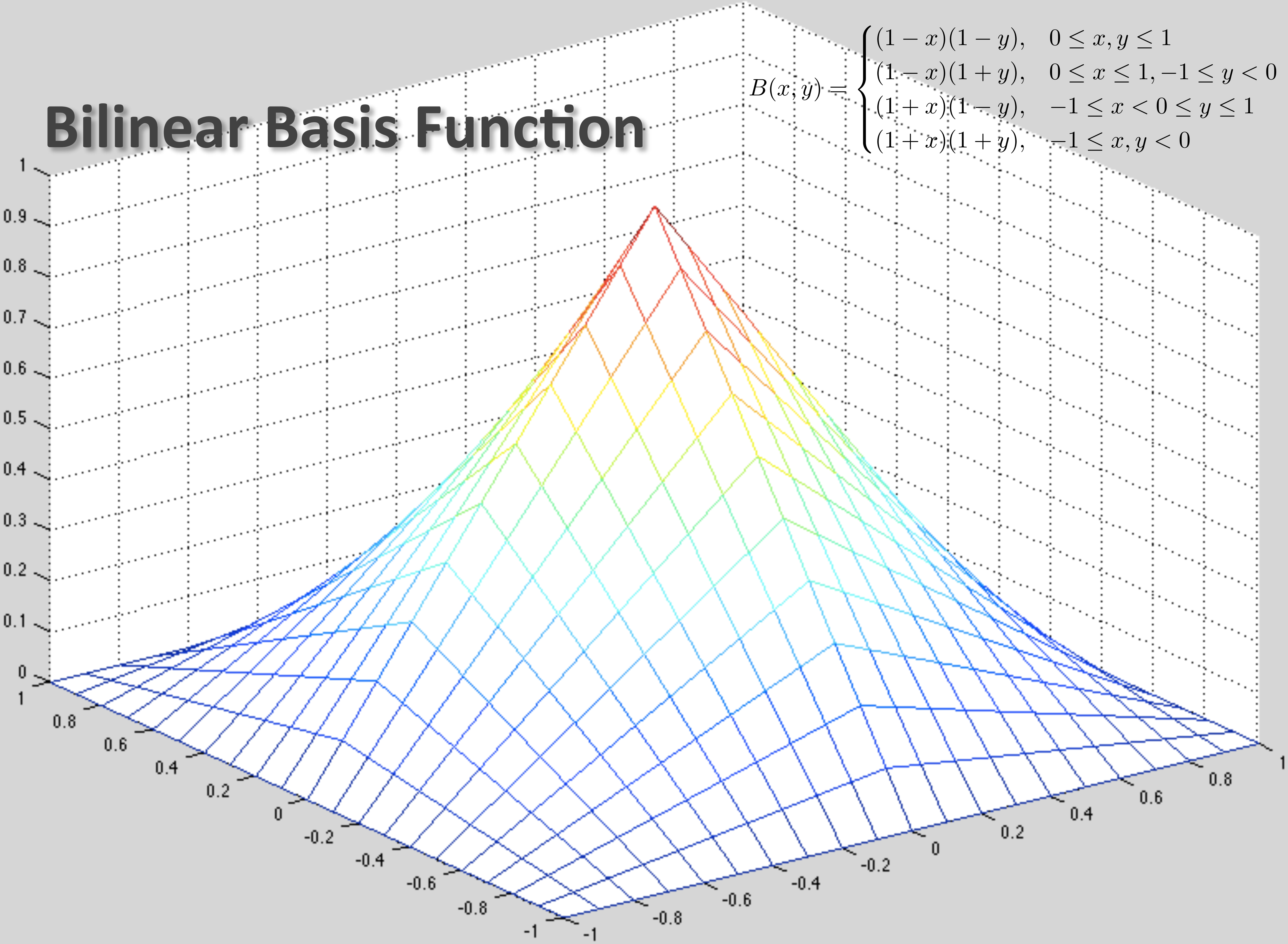
- A texel in a texture map is also a basis function
  - Think about it: it’s a finite set of numbers that together define a function on the continuous 2D domain

Texture Maps

- A texel in a texture map is also a basis function
  - Think about it: it's a finite set of numbers that together define a function on the continuous 2D domain
- The exact shape of the basis function determined by the interpolation method used
  - Most common: bilinear basis, here defined on $[-1,1]^2$

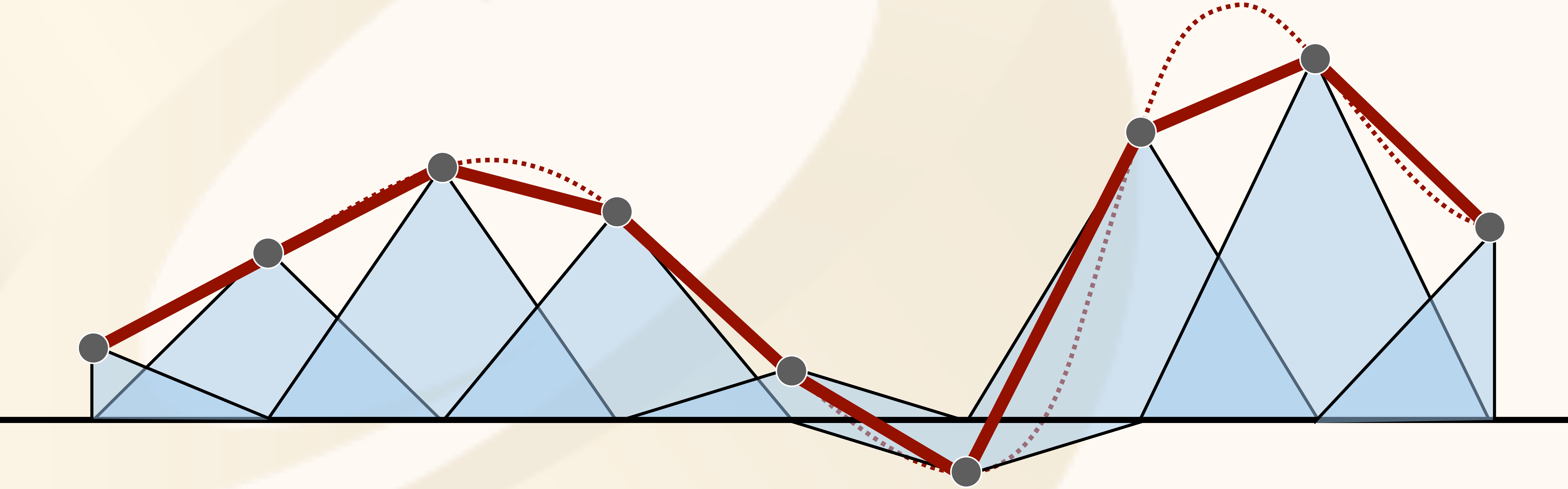
Bilinear Basis Function

$$B(x,y) = \begin{cases} (1-x)(1-y), & 0 \le x, y \le 1 \\ (1-x)(1+y), & 0 \le x \le 1, -1 \le y < 0 \\ (1+x)(1-y), & -1 \le x < 0 \le y \le 1 \\ (1+x)(1+y), & -1 \le x, y < 0 \end{cases}$$



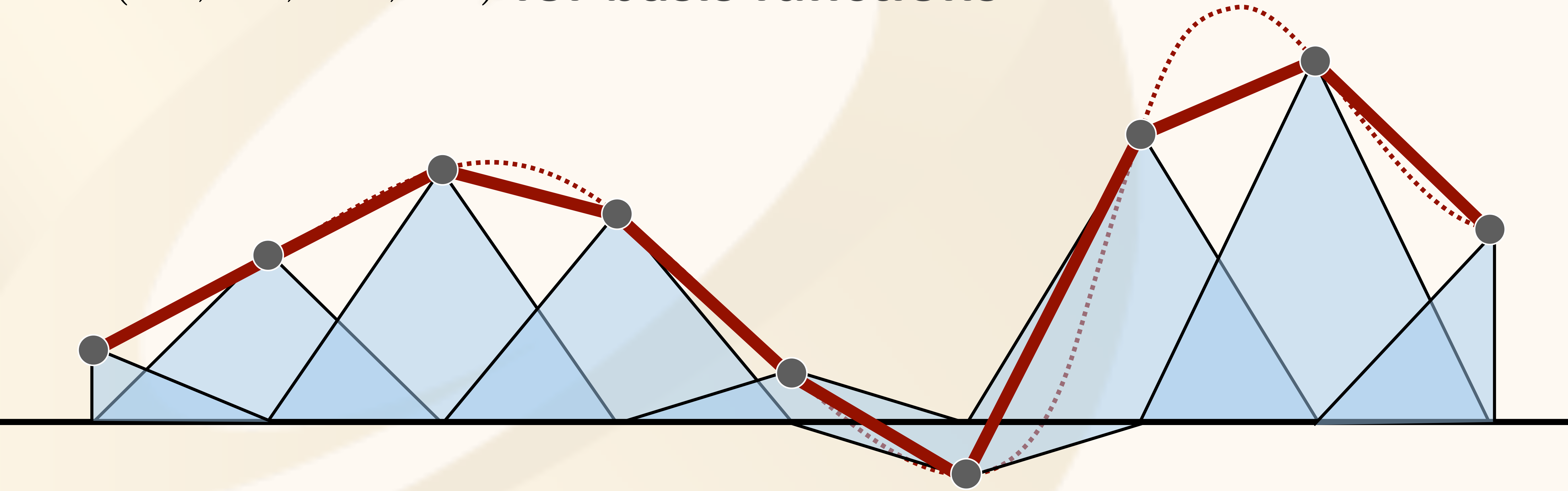
“Projection Operators”

- What’s going on: we take a function defined on a continuous domain, do something, and get an approximate version out



“Projection Operators”

- Projection can be written as linear operator $\mathcal{P}$
- Take an arbitrary function L, return finite approximation $\mathcal{P}L$ described by vector of weights $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ for basis functions



Different Projections

- Sample at just one point ("point collocation")
  - For vertex basis, look at value at the vertex and use as weight:

$$f(x) \approx \sum_i \underbrace{B_i(x)}_{\substack{\text{Basis function}\\\text{associated}\\\text{with i:th vertex}}}\ \underbrace{f(x_i)}_{\substack{\text{Function value}\\\text{at i:th vertex}}}$$

- This process takes samples at vertices and "smears" them across the triangles to yield a continuously-defined function

Different Projections

- Sample at just one point ("point collocation")
  - For vertex basis, look at value at the vertex and use as weight:

$$f(x) \approx \sum_i B_i(x)\, f(x_i)$$

Basis function associated with i:th vertex

Function value at i:th vertex

- What condition does the basis have to fulfil for this to make sense?

Different Projections

- Sample at just one point ("point collocation")
  - For vertex basis, look at value at the vertex and use as weight:

$$f(x) \approx \sum_i B_i(x)\, f(x_i)$$

Basis function associated with i:th vertex

Function value at i:th vertex

- What condition does the basis have to fulfil for this to make sense? Must have $B_i(x_j) = 0$ when $i \neq j$ (why?)

Different Projections

- Sample at just one point ("point collocation")
  - For vertex basis, look at value at the vertex and use as weight:

$$f(x) \approx \sum_i B_i(x)\, f(x_i)$$

- "Least squares projection", aka L_2 projection
  - Find coefficients that minimize the squared norm of the error integrated over the entire domain

Least Squares Projection

- Task: find $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ such that the *residual*

$$R := \int_S \left(f(x) - \sum_{i=1}^{N} \alpha_i B_i(x) \right)^2 \mathrm{d}x$$

is minimized.

What we want to project

Projection result (only alphas unknown, basis chosen beforehand)

- Residual is input function *f* minus the approximation
- Minimize the squared integral of R over the domain
  - If approximation is exact, this is zero (never happens)
  - Need to solve for the weights alpha

Turns Out To Be Simple

$$\operatorname{argmin}_{\alpha} \quad \int_S \left(f(x) - \sum_{i=1}^{N} \alpha_i B_i(x) \right)^2 \mathrm{d}x$$

Turns Out To Be Simple

$$\operatorname{argmin}_{\alpha} \quad \int_S \left(f(x) - \sum_{i=1}^{N} \alpha_i B_i(x) \right)^2 \mathrm{d}x$$

$\Leftrightarrow$ expand the square

$$\int_S \left(f(x)^2 - 2 \sum_i f(x)\, \alpha_i\, B_i(x) + 2 \sum_i \sum_j \alpha_i\, \alpha_j B_i(x) B_j(x) \right) \mathrm{d}x$$

Turns Out To Be Simple

$$\operatorname{argmin}_{\alpha} \quad \int_S \left(f(x) - \sum_{i=1}^{N} \alpha_i B_i(x) \right)^2 \mathrm{d}x$$

$$\Leftrightarrow$$

$$\int_S \left(\cancel{f(x)}^2 - \cancel{2} \sum_i f(x)\, \alpha_i\, B_i(x) + \cancel{2} \sum_i \sum_j \alpha_i\, \alpha_j B_i(x) B_j(x) \right) \mathrm{d}x$$

Constant
(Does not affect
minimization)

Turns Out To Be Simple

$$\operatorname{argmin}_{\alpha} \quad \int_S \left(f(x) - \sum_{i=1}^{N} \alpha_i B_i(x) \right)^2 \mathrm{d}x$$

$$\Leftrightarrow$$

$$\int_S \left(\cancel{f(x)^2} - \cancel{2} \sum_i f(x)\, \alpha_i\, B_i(x) + \cancel{2} \sum_i \sum_j \alpha_i\, \alpha_j B_i(x) B_j(x) \right) \mathrm{d}x$$

$$\Leftrightarrow$$ (rearrange integration and summation)

$$- \sum_i \alpha_i \boxed{\int_S f(x)\, B_i(x) \mathrm{d}x} + \sum_i \sum_j \alpha_i\, \alpha_j \boxed{\int_S B_i(x) B_j(x) \mathrm{d}x}$$

Independent of alphas, depend on just *f* and *B*s

Independent of alphas, depend only on *B*s

Inner products

$$\sum_i \alpha_i \underbrace{\int_S f(x)\, B_i(x)\mathrm{d}x}_{:= \langle f, B_i \rangle} + \sum_i \sum_j \alpha_i\, \alpha_j \underbrace{\int_S B_i(x) B_j(x)\mathrm{d}x}_{:= \langle B_i, B_j \rangle}$$

- These integrals of products of functions are called *inner products*
- Think about analogy to usual vectors: $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_i^D x_i\, y_i$
  - Again, sums become integrals when dimension D grows without limit

Turns Out To Be Simple

$$\sum_i \alpha_i \underbrace{\int_S f(x)\, B_i(x)\mathrm{d}x}_{:= \langle f, B_i \rangle} + \sum_i \sum_j \alpha_i\, \alpha_j \underbrace{\int_S B_i(x) B_j(x)\mathrm{d}x}_{:= \langle B_i, B_j \rangle}$$

- So the final task is to find alphas that minimize

$$-\sum_i \alpha_i \langle f, B_i \rangle + \sum_i \sum_j \alpha_i\, \alpha_j \langle B_i, B_j \rangle$$

or, in matrix-vector form

$$-\boldsymbol{f}^T \boldsymbol{\alpha} + \boldsymbol{\alpha}^T \boldsymbol{B} \boldsymbol{\alpha}$$

$$f_i = \langle f, B_i \rangle$$
$$B_{i,j} = \langle B_i, B_j \rangle$$

$$-\boldsymbol{f}^T\boldsymbol{\alpha}+\boldsymbol{\alpha}^T\boldsymbol{B}\boldsymbol{\alpha}$$

- It’s a quadratic function in the vector alpha
  - $\boldsymbol{f}$, $\boldsymbol{B}$ are constants, given $f(x)$ and the basis functions $B_i(x)$
- What happens when you differentiate a quadratic function and set to zero?

A Linear System

- Least squares projection solution given by

$$\boldsymbol{B\alpha} = \boldsymbol{f}$$

where $f_i = \langle f, B_i \rangle$ and $B_{i,j} = \langle B_i, B_j \rangle$

Easy Special Case: Box Functions

- Least squares projection solution given by

$$\boldsymbol{B}\boldsymbol{\alpha} = \boldsymbol{f}$$

where $f_i = \langle f, B_i \rangle$ and $B_{i,j} = \langle B_i, B_j \rangle$

- What if we use the piecewise constant box basis?
  - Then $B_{i,j} = 0$ when i != j. (Why?)

Easy Special Case: Box Functions

- Least squares projection solution given by

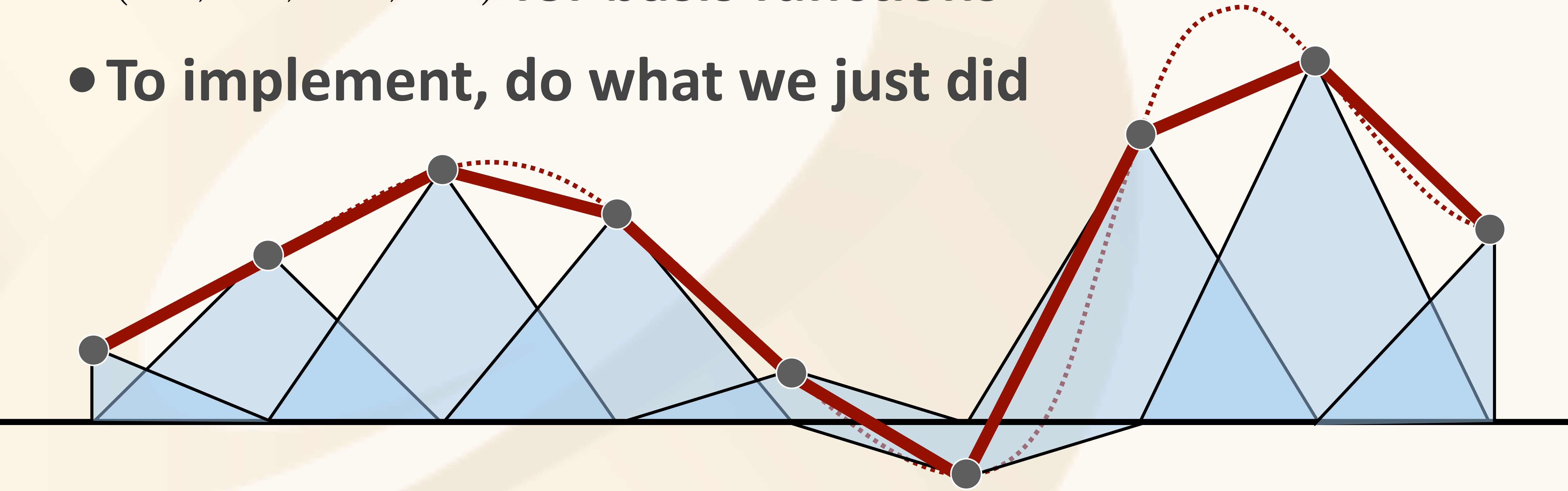
$$\boldsymbol{B\alpha} = \boldsymbol{f}$$

where $f_i = \langle f, B_i \rangle$ and $B_{i,j} = \langle B_i, B_j \rangle$

- What if we use the piecewise constant box basis?
  - Then $B_{i,j} = 0$ when i != j. (Why?)
  - In fact, the $B_{i,j}$ are just the areas under the boxes
  - Convince yourself that then the basis coefficients are just area averages of *f* over the boxes!

“Projection Operators” Recap

- Projection can be written as linear operator $\mathcal{P}$
- Take an arbitrary function L, return finite approximation $\mathcal{P}L$ described by vector of weights $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ for basis functions
- To implement, do what we just did



OK, Why all the Trouble?

- Video

Radiosity Derivation

- Rendering equation

$$L = \mathcal{T} L + E$$

- Now let's search for an approximate solution in terms of basis functions, i.e. try to find coefficients s.t.

$$L(x) \approx \sum_i \alpha_i \, B_i(x)$$

Radiosity Derivation

- Rendering equation

$$L = \mathcal{T}L + E$$

- This amounts to applying the projection operator:

PT = Transport followed by projection

$$\mathcal{P}L = \mathcal{P}\mathcal{T}(\mathcal{P}L) + \mathcal{P}E$$

PL = approximate solution in terms of basis functions

PE = projected emission function

Lo and Behold

- The discretized rendering equation

$$\mathcal{P}L = \mathcal{P}\mathcal{T}(\mathcal{P}L) + \mathcal{P}E$$

is actually a *finite* system of linear equations!

- Why?
  - Clearly, both sides are finite basis expansions because we always apply *P* to every term
  - Hence, for the LHS and RHS to match, the basis coefficients for PL on both side must be equal

$$\mathcal{P}L = \mathcal{P}\mathcal{T}(\mathcal{P}L) + \mathcal{P}E \tag{1}$$

- Let’s write things out a bit

Alphas are the unknowns we seek!

$$\mathcal{P}L = \sum_i \alpha_i B_i \tag{2}$$

$$\mathcal{P}L = \mathcal{P}\mathcal{T}(\mathcal{P}L) + \mathcal{P}E \tag{1}$$

- Let's write things out a bit

Alphas are the unknowns we seek!

$$\mathcal{P}L = \sum_i \alpha_i B_i \tag{2}$$

Substitute (2) into (1)

$$= \mathcal{P}\mathcal{T} \sum_j \alpha_j B_j + \mathcal{P}E$$

$$\mathcal{P}L = \mathcal{P}\mathcal{T}(\mathcal{P}L) + \mathcal{P}E \tag{1}$$

- Let’s write things out a bit

Alphas are the unknowns we seek!

$$\mathcal{P}L = \sum_i \alpha_i \, B_i \tag{2}$$

Substitute (2) into (1)

$$= \mathcal{P}\mathcal{T} \sum_j \alpha_j \, B_j + \mathcal{P}E$$

Move *PT* inside the sum
(can be done as they’re both linear)

$$= \sum_j \alpha_j \left(\mathcal{P}\mathcal{T} \, B_j \right) + \mathcal{P}E$$

PTB$_j$ does not depend on the alphas or the emission!

$$\mathcal{P}L = \mathcal{P}\mathcal{T}(\mathcal{P}L) + \mathcal{P}E \tag{1}$$

- Let’s write things out a bit

Alphas are the unknowns we seek!

$$\mathcal{P}L = \sum_i \alpha_i\, B_i \tag{2}$$

Substitute (2) into (1)

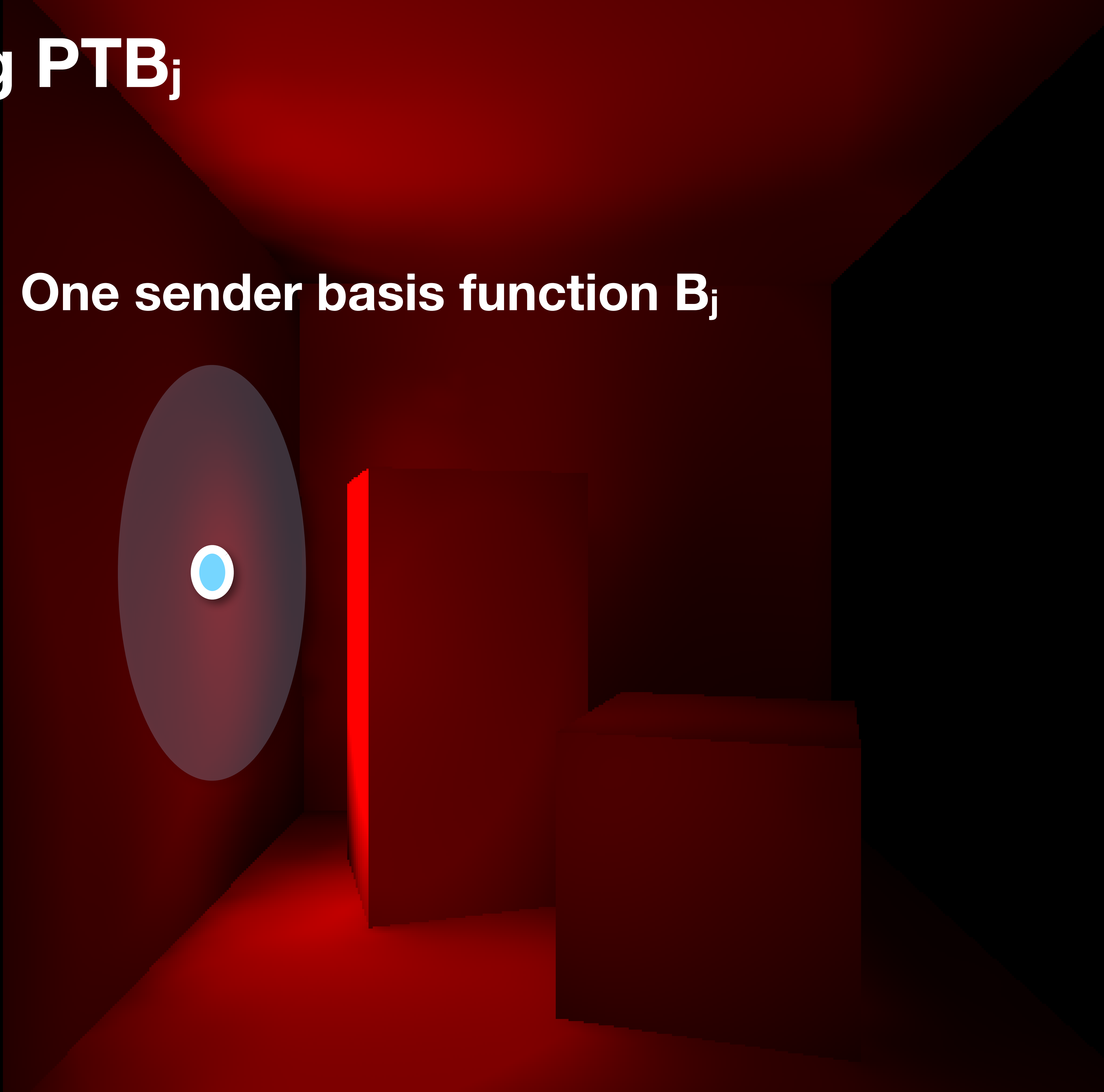
$$= \mathcal{P}\mathcal{T} \sum_j \alpha_j\, B_j + \mathcal{P}E$$

Move *PT* inside the sum
(can be done as they’re both linear)

$$= \sum_j \alpha_j \left(\mathcal{P}\mathcal{T}\, B_j\right) + \mathcal{P}E$$

TB_j is the once-bounce illumination received by all surfaces when the basis function B_j acts as an emitter. P merely projects it.

Visualizing PTB_j



One sender basis function B_j

Red = The one-bounce illumination received by other surfaces when B_j is the only emitter

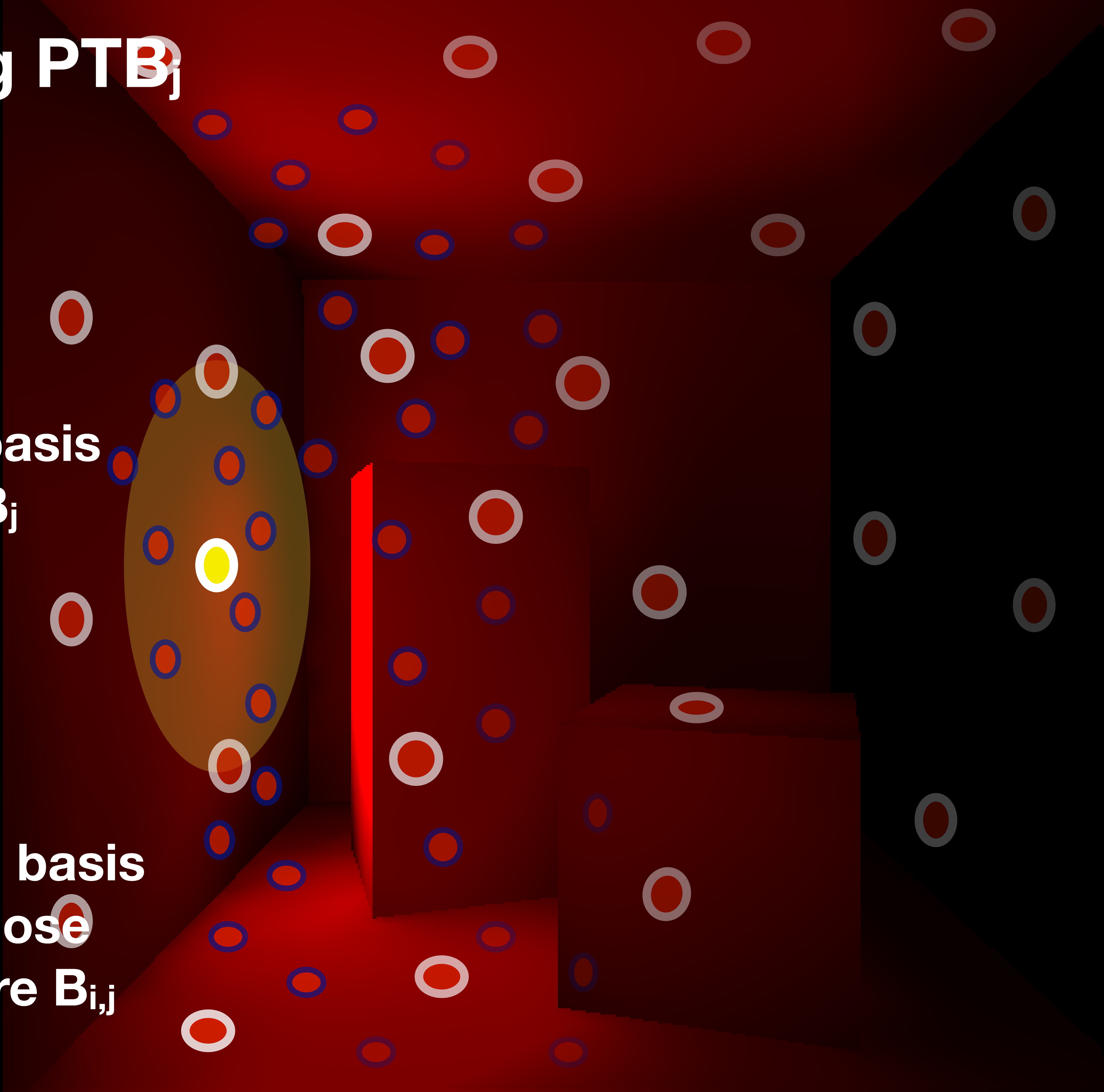
Let's Finish It

- $\mathcal{P}\mathcal{T}B_j$ is the basis expansion of the one-bounce illumination that results when the emission is B_j
- Because it is a basis expansion, it has its own basis coefficients. We'll call them $B_{i,j}$:

$$(\mathcal{P}\mathcal{T}B_j)(x) = \sum_i B_{i,j}\, B_i(x)$$

- "how to scale the *i*th basis function B_i so that when summed together, they together represent the scene illuminated by the *j*th basis function B_j"

Visualizing PTB_j



One sender basis function B_j

Many receiving basis functions whose coefficients are $B_{i,j}$

Final Radiosity Equation

- The abstract projected equation

$$\mathcal{P}L = \mathcal{P}\mathcal{T}(\mathcal{P}L) + \mathcal{P}E$$

is actually the linear system

$$\boldsymbol{\alpha} = \boldsymbol{B}\boldsymbol{\alpha} + \boldsymbol{e}$$

where the components of alpha are the unknown coefficients, the matrix **B** consists of the basis coefficients of PTB_j for all j as shown before, and **e** is the basis coefficient vector projected emission PE.

Important Point

$$\boldsymbol{\alpha} = \boldsymbol{B}\boldsymbol{\alpha} + \boldsymbol{e}$$

- This is all good, but we *never ever* form the matrix B explicitly. Why?

Important Point

$$\boldsymbol{\alpha} = \boldsymbol{B}\boldsymbol{\alpha} + \boldsymbol{e}$$

- This is all good, but we *never ever* form the matrix B explicitly. Why?

- We can easily have 10M basis functions in the scene => matrix is $10M^2 = 10^{14}$ float3 entries = 10^{15} bytes
  - We really don’t have the time to compute them
  - Nor space to store them

- Solution: use iterative methods

Iterative Linear Solver

- *Iterative method* means we **don't** first invert the matrix and then use a direct solver like Gaussian elimination
- instead, compute matrix-vector products and iterate
- No, you don't need the full matrix in memory to compute matrix-vector products
  - See Jacobi iteration, Gauss-Seidel iteration, conjugate gradient method, Krylov subspace methods
  - Some *very smart approximate product algorithms* are known for some particular matrices/operators

Discrete Radiosity Equation

$$\boldsymbol{\alpha} = \boldsymbol{B}\boldsymbol{\alpha} + \boldsymbol{e}$$

- **e** is the vertex color vector where only the emitting polygons' vertices have a nonzero radiosity
- Turns out we can apply the Neumann series here, too!

$$\boldsymbol{\alpha} = \boldsymbol{e} + \boldsymbol{B}\boldsymbol{e} + \boldsymbol{B}^2\boldsymbol{e} + \ldots$$

- ... and this is almost precisely what Max Payne's lighting solver does, as well as you in Assn 2!
  - Just one possible iteration for this equation, you'll find lots of others in textbooks (Jacobi, Gauss-Seidel, Southwell)
  - *Max Payne 2 does Southwell + smart partitioning, ask me*

Iterative Radiosity Solution (Jacobi)

$$\boldsymbol{\alpha} = \boldsymbol{e} + \boldsymbol{B}\boldsymbol{e} + \boldsymbol{B}^2\boldsymbol{e} + \dots$$

- Initialize: $\boldsymbol{\alpha} = \mathbf{e}, \boldsymbol{\beta} = \mathbf{e}$
- Then iterate:

1. $\boldsymbol{\beta} \leftarrow \mathbf{B}\boldsymbol{\beta}$
2. $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} + \boldsymbol{\beta}$

until happy

- What happens:

$$\boldsymbol{\beta} = \{\mathbf{e}, \mathbf{Be}, \mathbf{BBe}, \dots\}$$

$$\boldsymbol{\alpha} = \{\mathbf{e}, \mathbf{e} + \mathbf{Be}, \mathbf{e} + \mathbf{Be} + \mathbf{BBe}, \dots\}$$

Computing the Product

- How to compute $\mathbf{B}\beta$?
  - Using the basis expansion with coefficients β as the emission, compute at the one-bounce illumination cast on the scene and determine its projection coefficients.
  - When using vertex basis, very simple: evaluate the hemispherical irradiance integral at each vertex and turn it into outgoing radiance using albedo
  - And don’t forget to divide by pi :)
- Note! Do not update values of β while computing the full matrix product
  - Store product in temp vector and then update once all vertices have been computed

One Last Practical Detail

- We don’t actually store outgoing radiosity, but incident irradiance instead
  - Why? So that we can modulate the lighting using textures

- So, our basis expansion gives us irradiance, we turn it into radiosity by dividing by pi and multiplying by albedo in the shader

Pseudocode Using Vertex Basis

```
// these are vectors of length N, where N is the number of vertices
// they store radiosity before multiplied by albedo
vector alpha, beta, temp, e;
e = project(E);                      // set the colors of emitter vertices
alpha = beta = e;                    // init

for bounce=1 to numBounces
    clear(temp);                     // set to zero
    for i=1 to N                     // loop over vertices
        B = formBasis(vertices[i]);  // you already know how
        res = Vec3f(0);
        // M is the number of rays to sample hemisphere with
        for j=1 to M
            Wi = drawCosineWeightedDirection();   // you know how
            y = rayCast( vertices[i], Wi );       // you know how
            // get the radiosity for the hit point y, rho/pi is BRDF
            Li = rho(y)/pi * interpolateIrradiance( y, beta );
            res = res + Li;
        end
        temp[i] = res/M;
    end
    beta = temp;
    alpha = alpha + beta;
end
```

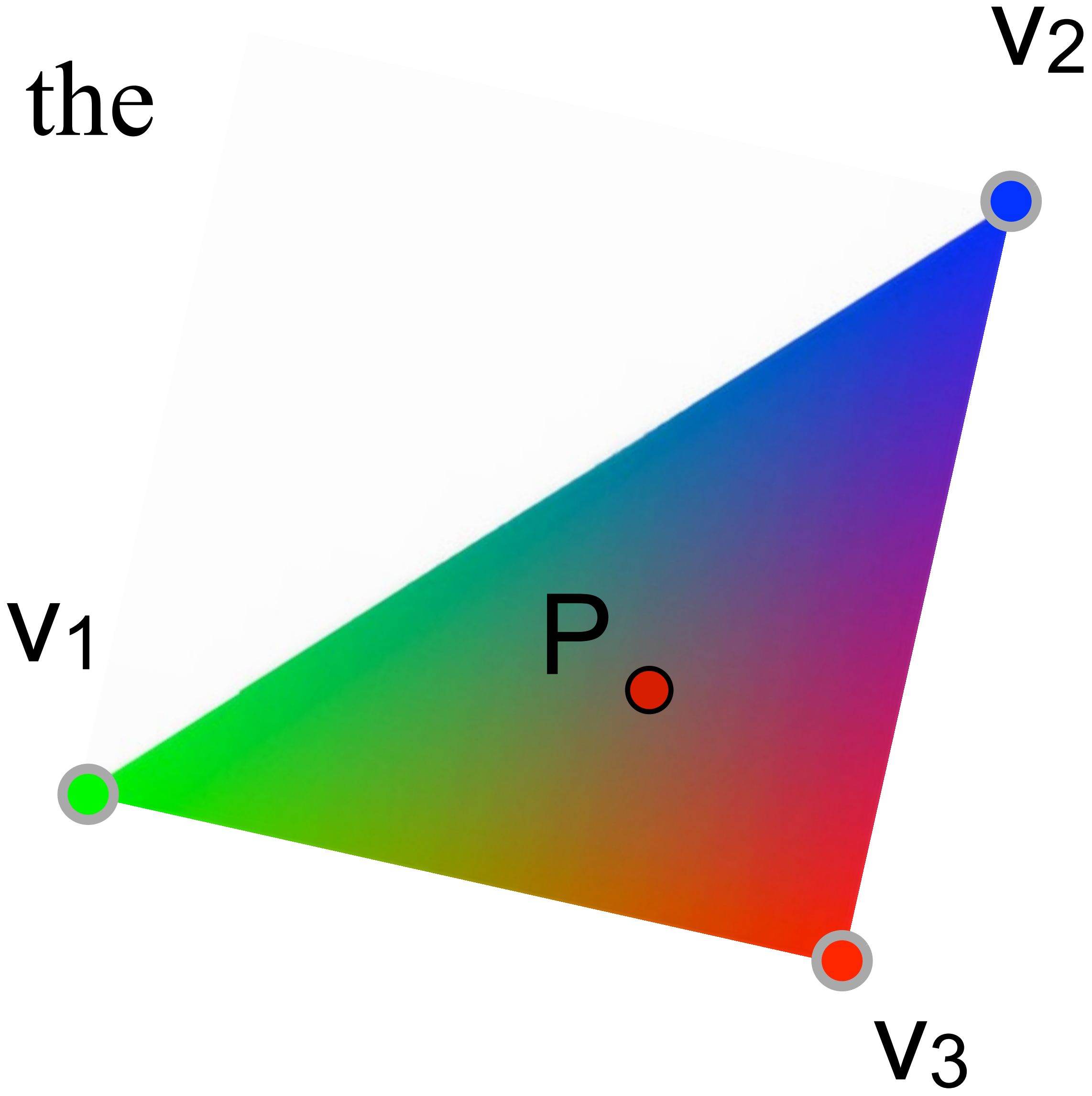
Interpolation

- `interpolateIrradiance(y, beta)` takes the hit point y and interpolates the irradiance values from the corresponding corner vertices using barycentrics

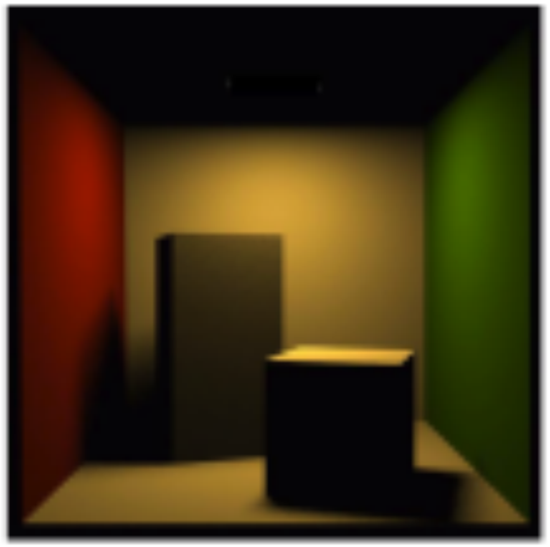
- You remember this from C3100...

Barycentric Interpolation Recap

- Values v_1, v_2, v_3 defined at **a**, **b**, **c**
  - Colors, normal, texture coordinates, etc.
- $\mathbf{P}(\alpha, \beta, \gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$ is the point...
- $v(\alpha, \beta, \gamma) = \alpha v_1 + \beta v_2 + \gamma v_3$ is the barycentric interpolation of v_1-v_3 at point **P**
  - Sanity check: $v(1,0,0) = v_1$, etc.
- I.e, once you know α, β, γ, you can interpolate values using the same weights.
  - Convenient!

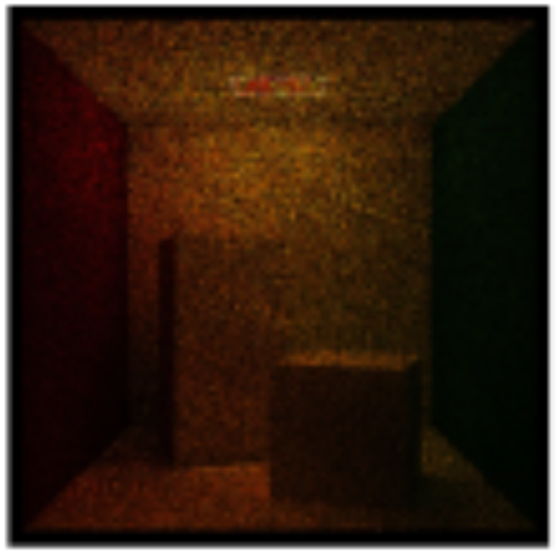


E



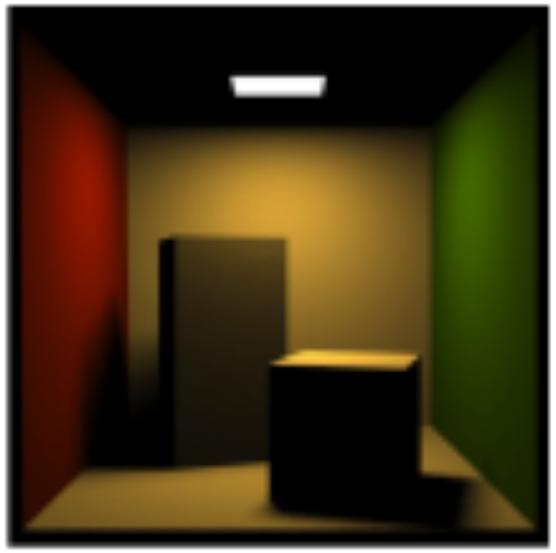
$\mathcal{T}E$

$\mathcal{T}\mathcal{T}E$

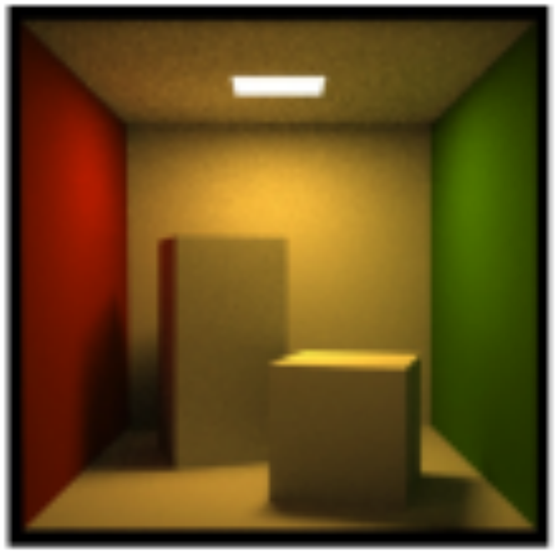


$\mathcal{T}\mathcal{T}\mathcal{T}E$

E



$E + \mathcal{T}E$



$E + \mathcal{T}E$
$+\mathcal{T}^2E$

$E + \mathcal{T}E$
$+\mathcal{T}^2E$
$+\mathcal{T}^3E$

adapted from **Pat Hanrahan, Spring 2010**

Discussion

- This was for vertex-based interpolation
- Often one uses texture maps, so-called *lightmaps*, for storing the irradiance
  - This is what we did (video)
  - Why? To get detailed illumination, you need many vertices
  - Downside: building UV parameterizations over the scene hard
  - Also, we computed the hemispherical integrals using the GPU using a so-called hemicube technique

- However, the main ingredients of the lighting solver are *precisely the same*

Discussion 2

- The loop over vertices is embarrassingly parallel
  - We had a simple distributed cluster running this in Max Payne
  - But need to synchronize across bounces

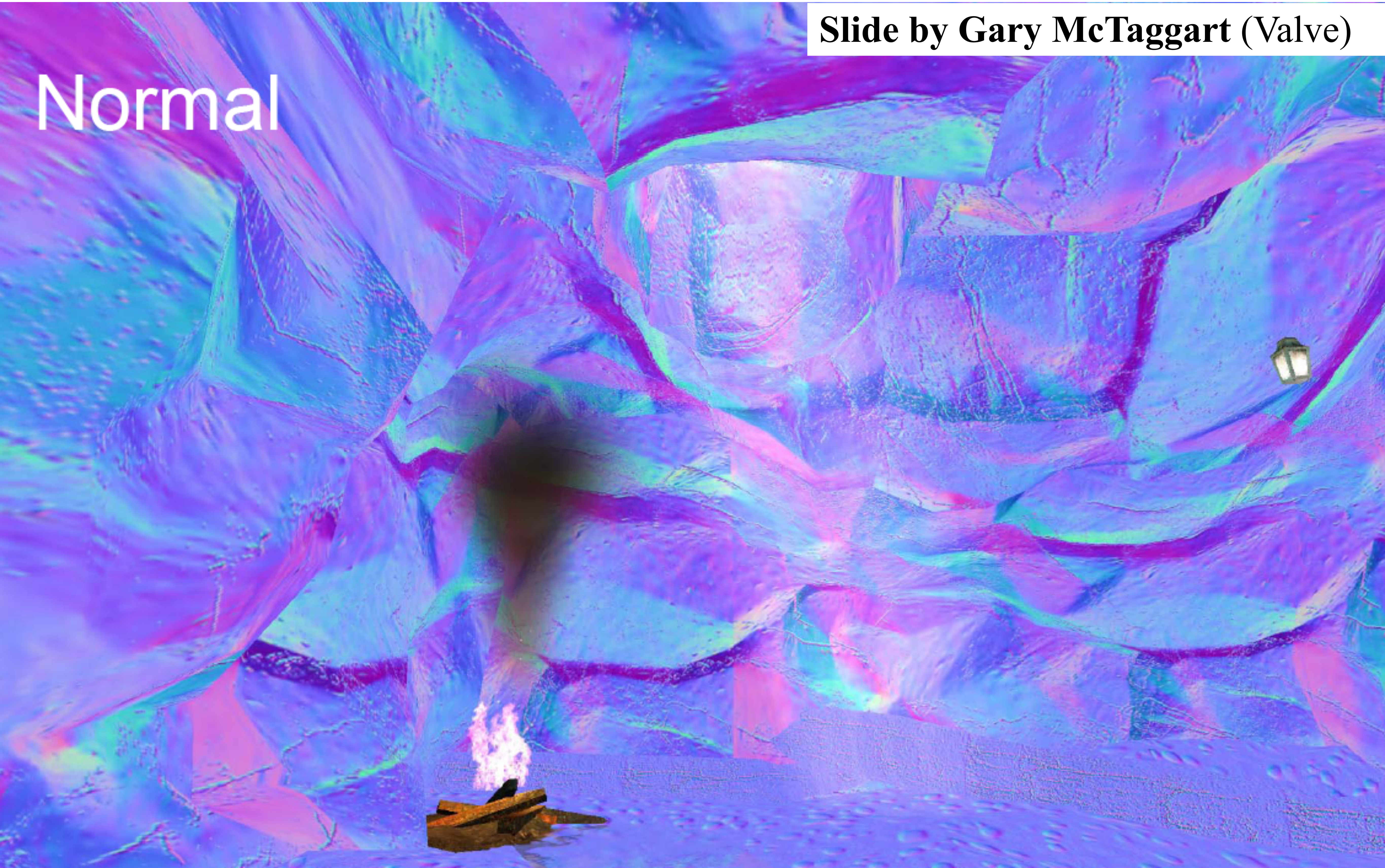
- But you can be even smarter
  - In Max Payne 2, we solved each room in the scene separately in its own cluster node
    - Less data to transfer over network, faster gathering integrals
  - Then, light was propagated between the rooms through 4D light fields or *Lumigraphs*
  - Corresponds to a two-level block-structured iteration on the large linear system

Discussion 3

- You can also store directional information, not just irradiance
  - This allows you to combine radiosity and normal maps
  - Even if the irradiance is coarsely-sampled, you still get nice surface detail
  - “Spherical Harmonics” and “vector irradiance” are keywords
  - *Extra credit in your assignment*
- Also, as you notice, the lighting is static
  - But you can allow the lighting to vary in some predetermined linear space => Precomputed Radiance Transfer (VIDEO)
  - See my master’s thesis and ToG paper for an in-depth introduction to PRT

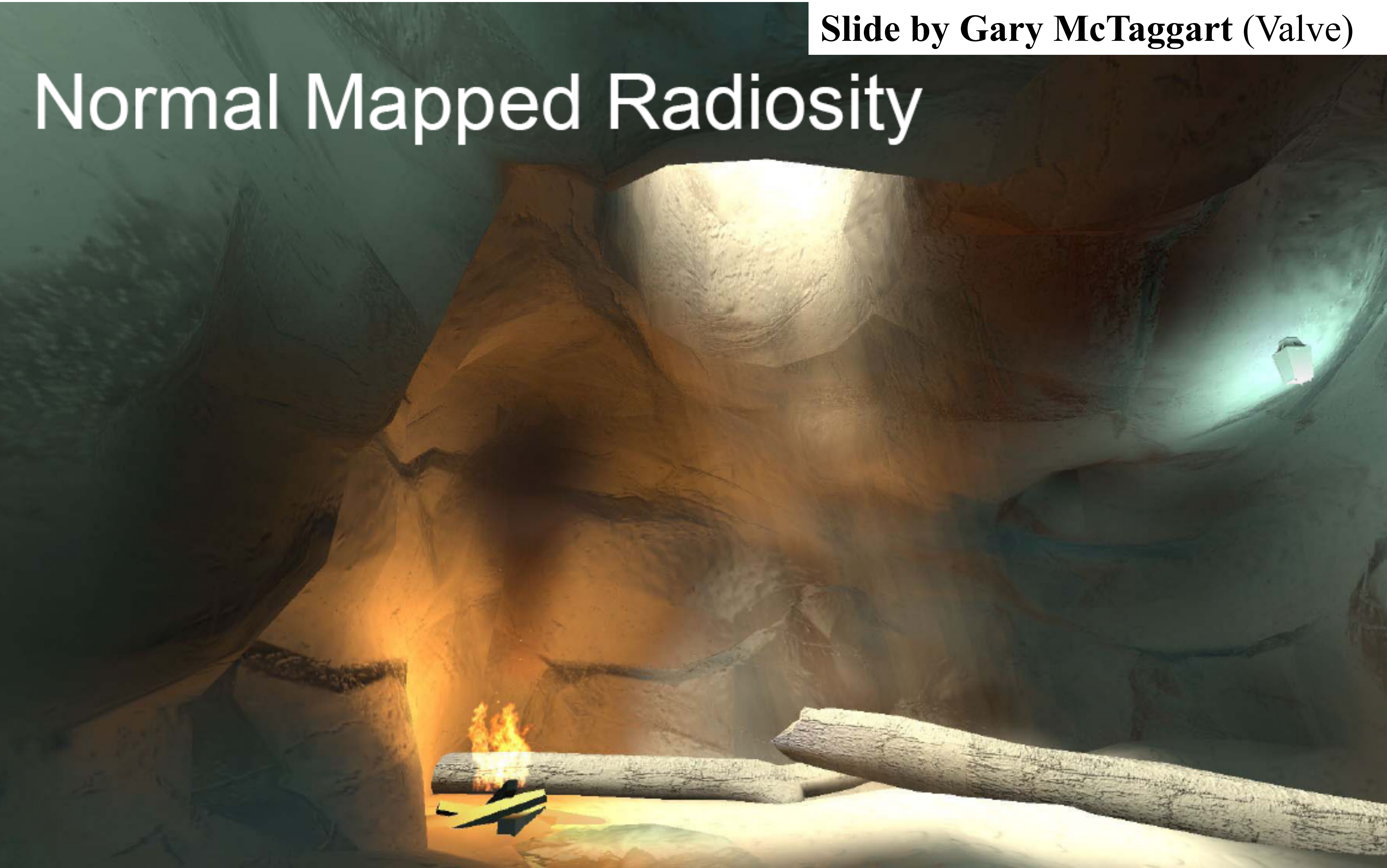
Radiosity + Normals in Half-Life 2

Slide by Gary McTaggart (Valve)



Slide by Gary McTaggart (Valve)

Slide by Gary McTaggart (Valve)



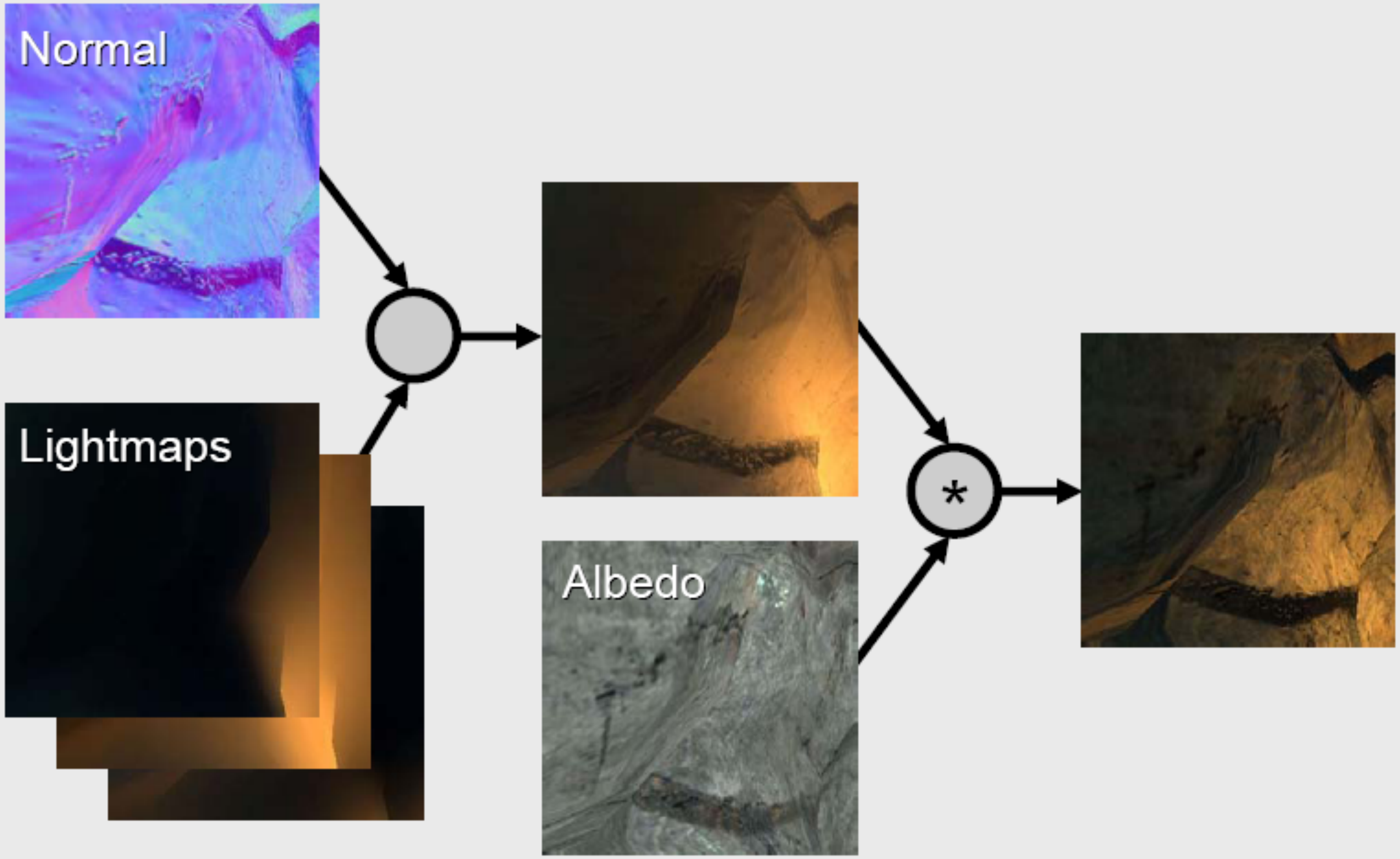
Normal Mapped Radiosity

Slide by Gary McTaggart (Valve)

Slide by Gary McTaggart (Valve)

Albedo * Normal Mapped Radiosity

Radiosity Normal Mapping Shade Tree



Discussion 4

- It often makes sense to compute direct lighting separately and only use basis functions for indirect

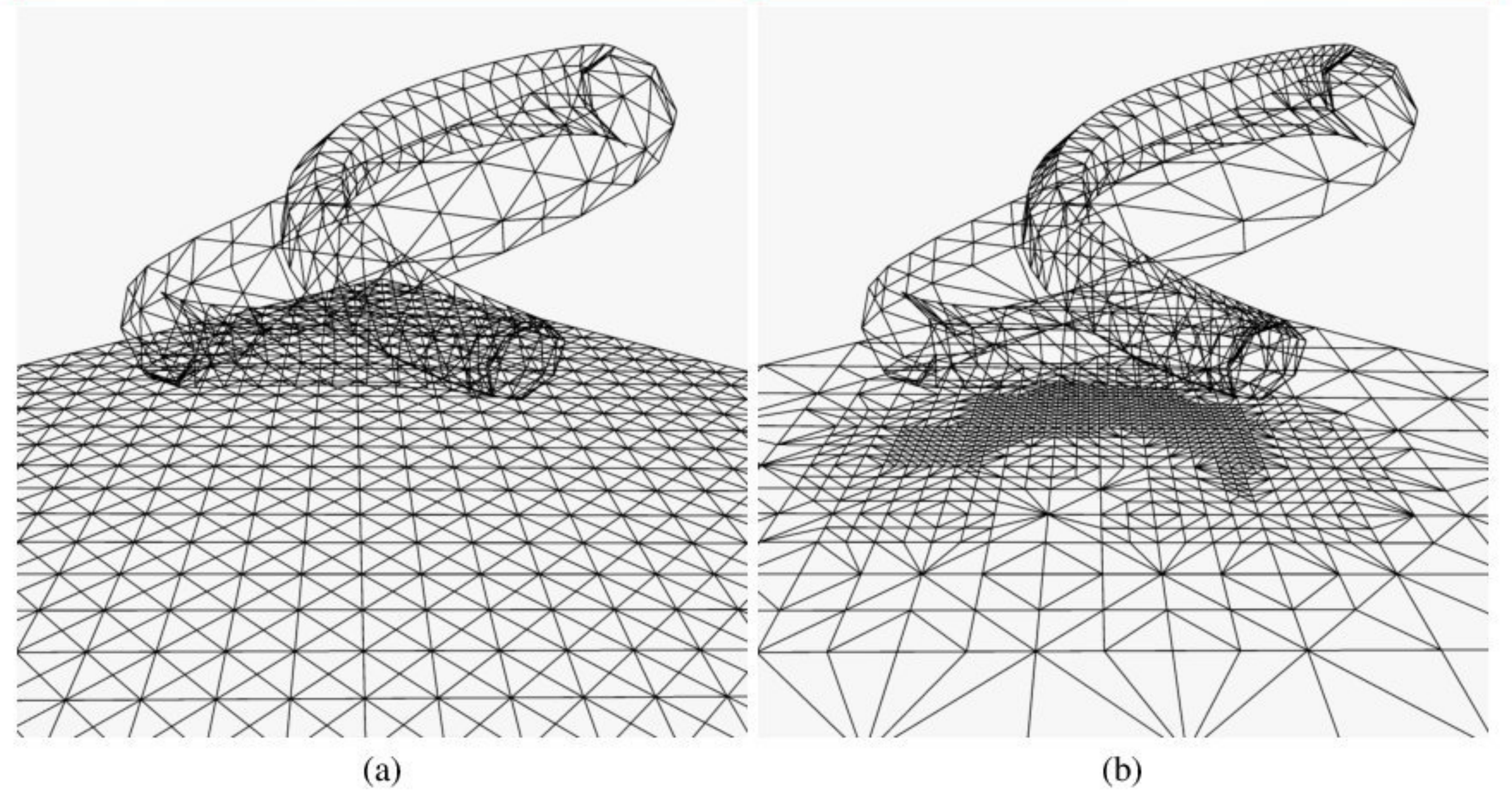
- Also, does it make sense to compute the lighting at a high resolution where it doesn’t vary very fast..?

Discussion 4

- It often makes sense to compute direct lighting separately and only use basis functions for indirect
- Also, does it make sense to compute the lighting at a high resolution where it doesn't vary very fast..?
  - You're right, it doesn't
- Adaptive refinement means you compute coarsely, then subdivide where you think you need to

Adaptive Refinement Example

Krivanek 2004



(a) (b)

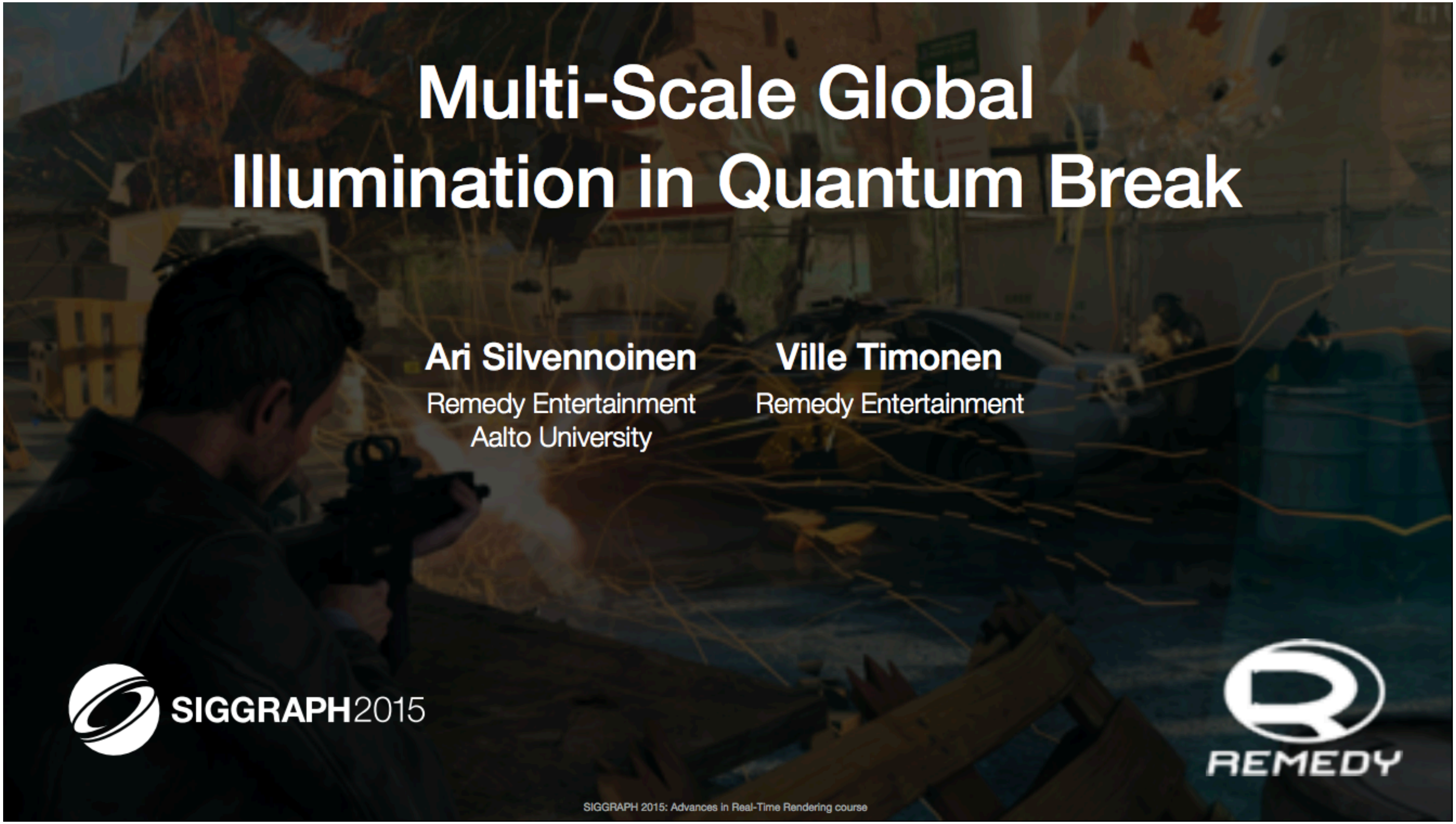
Figure 5: (a) Uniform subdivision (1953 vertices and 3504 triangles). (b) Adaptive subdivision (1540 vertices, 3720 triangles).

Final Conclusions

- Meshing is hard
- Lightmaps are hard (but they are still used)

- You can get around limitations of both by using meshless basis functions (Lehtinen et al. 2008)
  - Also supports adaptive refinement
  - Rendering cost is pretty high, though.

Modern Take (link)



Multi-Scale Global Illumination in Quantum Break

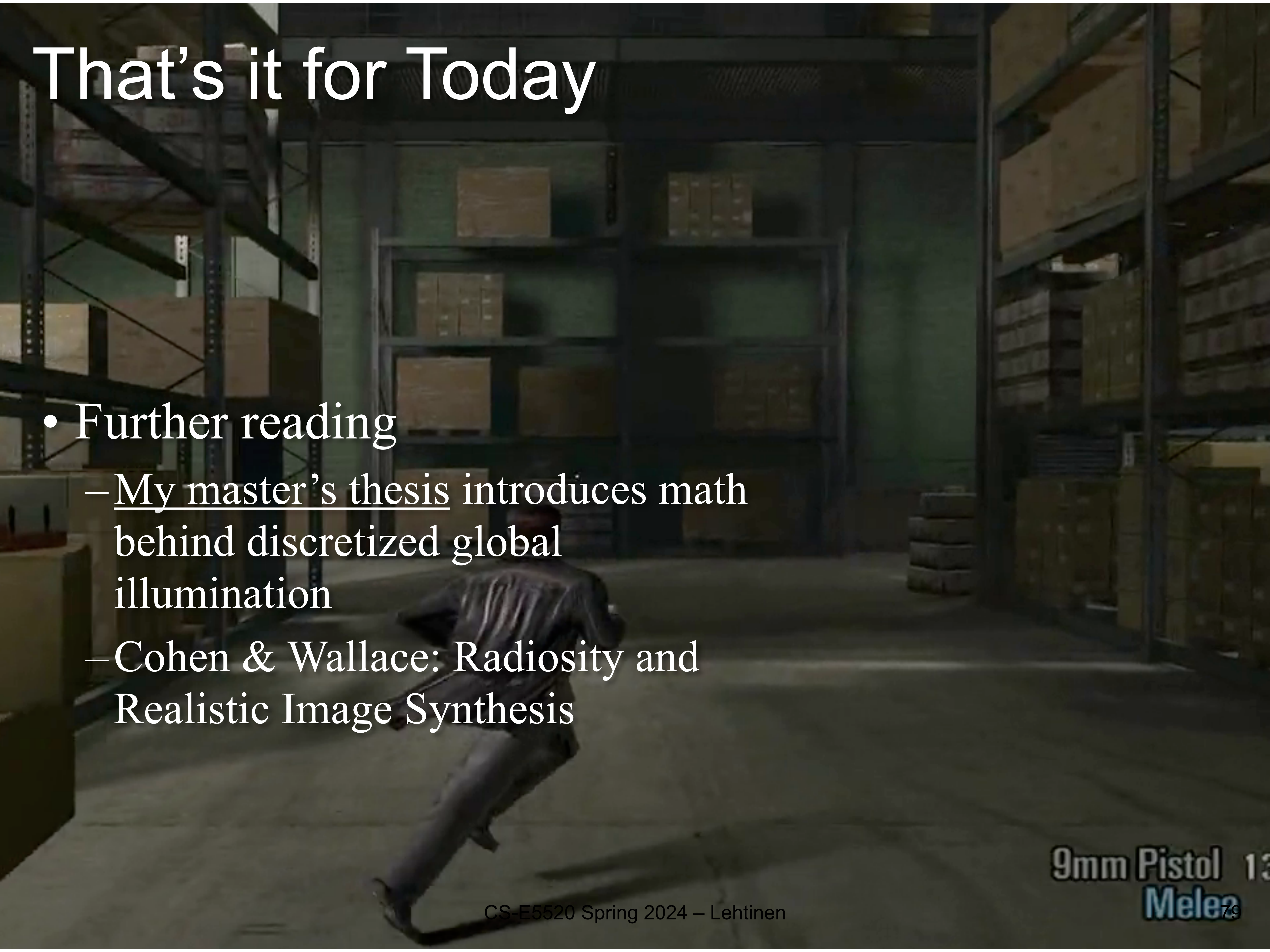
Ari Silvennoinen
Remedy Entertainment
Aalto University

Ville Timonen
Remedy Entertainment

SIGGRAPH 2015: Advances in Real-Time Rendering course

Direct-to-indirect precomputed light transport using meshless hierarchical basis functions

That’s it for Today



- Further reading
  - My master’s thesis introduces math behind discretized global illumination
  - Cohen & Wallace: Radiosity and Realistic Image Synthesis