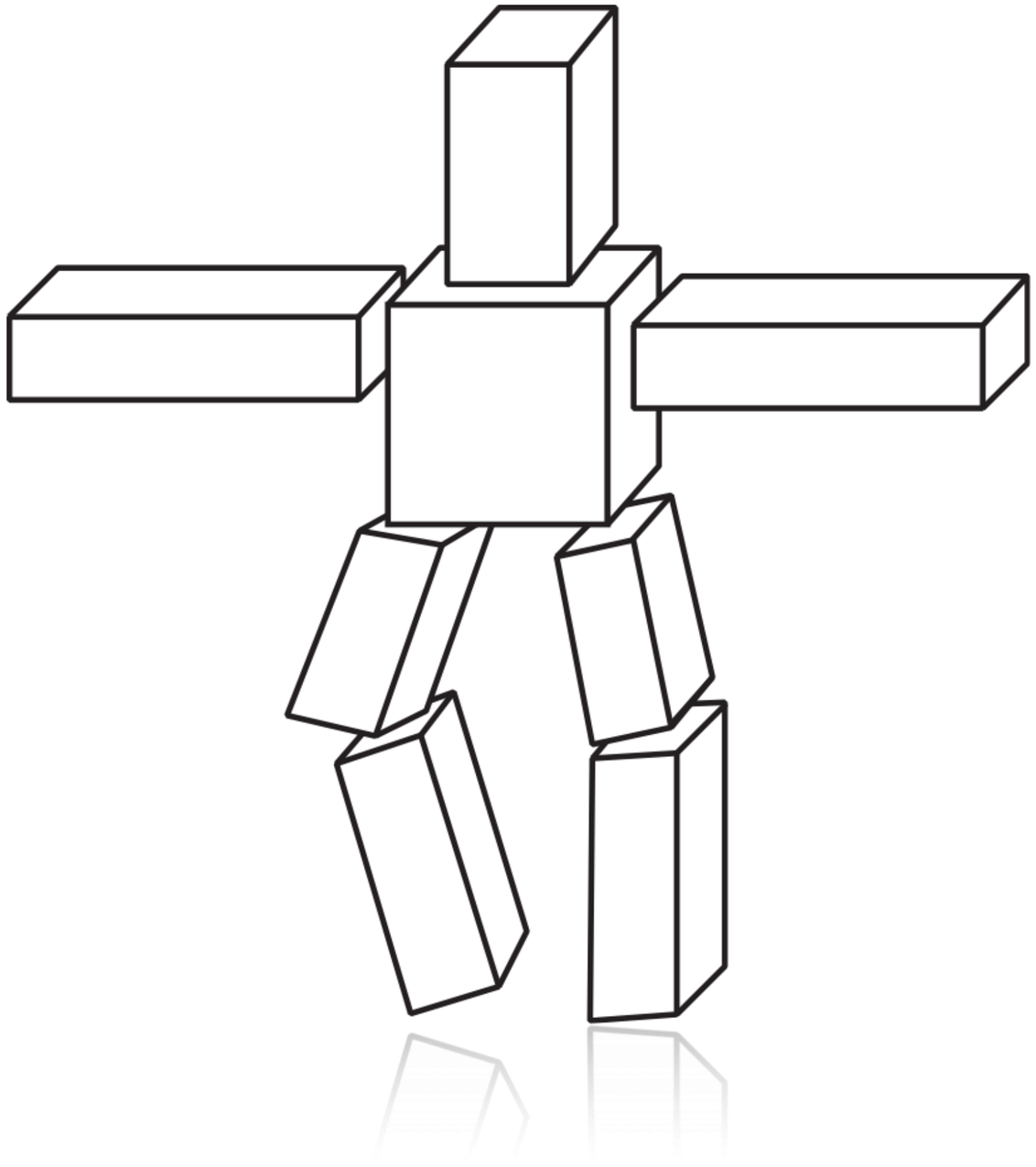
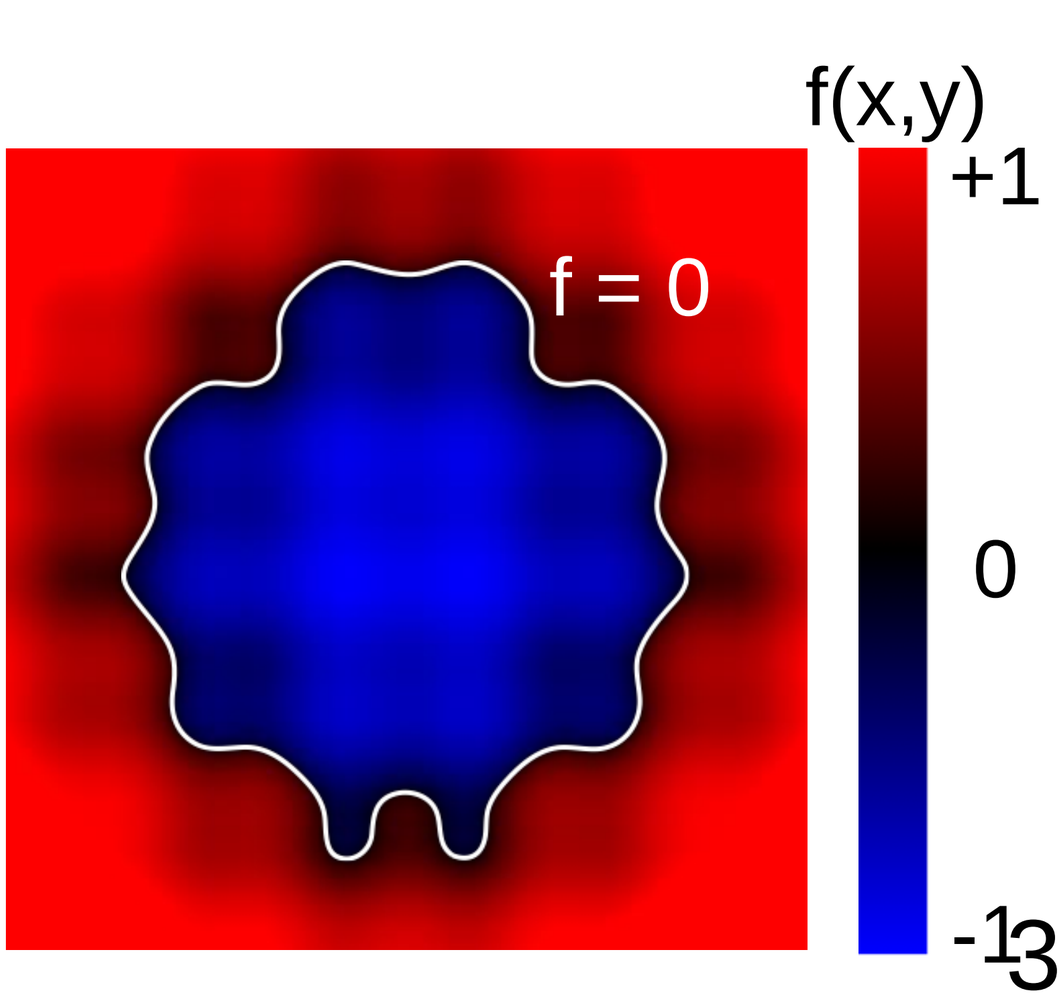
Perspective Projection Transformations, Geometry and Texture Mapping

How do we go beyond cube-man?



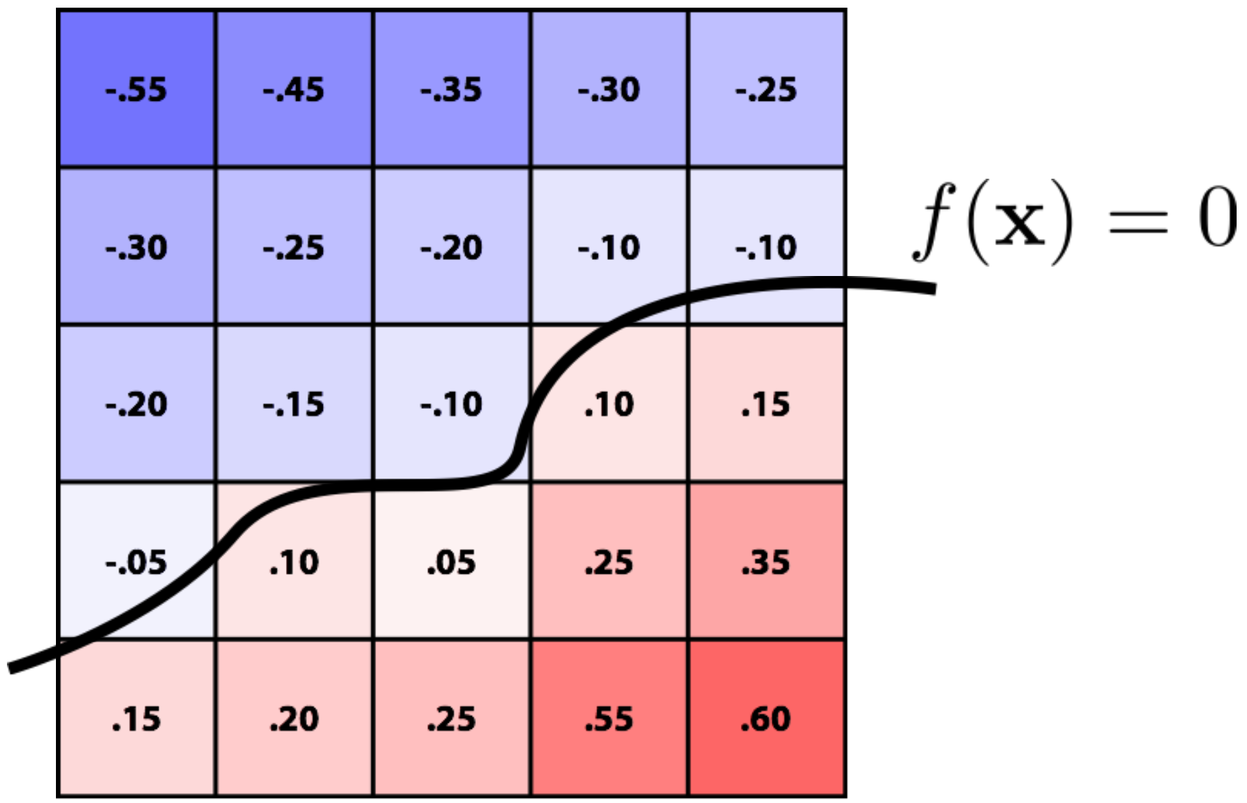
“Implicit” Representations of Geometry

- Points aren’t known directly, but satisfy some relationship
  - unit sphere: all points x such that $x^2+y^2+z^2=1$
  - More generally, f(x,y,z) = 0



Level Set Methods (Implicit)

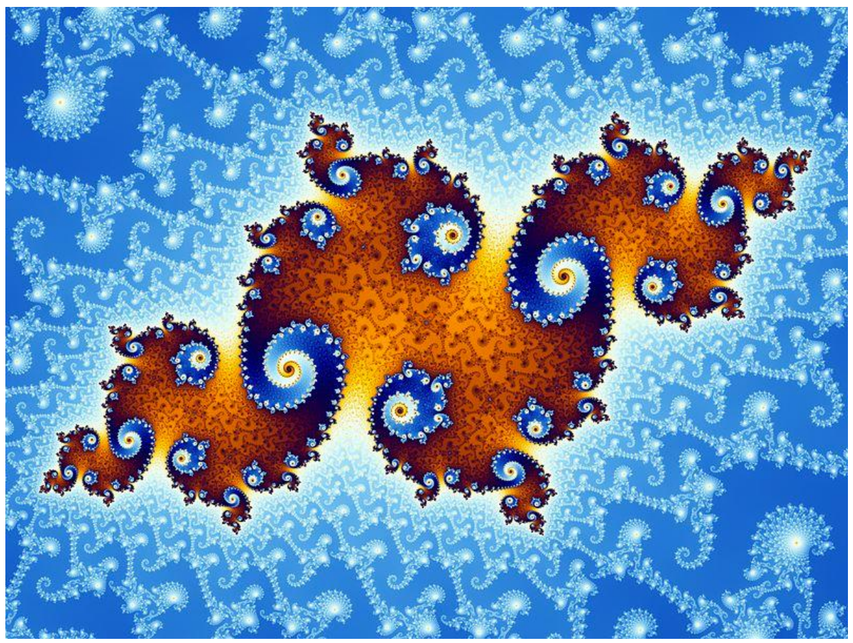
- Implicit surfaces have some nice features (e.g., merging/splitting)
- But, hard to describe complex shapes in closed form
- Alternative: store a grid of values approximating function



- Surface is found where *interpolated* values equal zero
- Provides much more explicit control over shape

Fractals and L-systems (Implicit)

- No precise definition; structures that exhibit self-similarity, detail at all scales
- New “language” for describing natural phenomena
- Hard to control shape!



L-Systems (Implicit)

Underlying computational system is `simple’

- geometry governed by simple rules
- the end result has interesting, useful, and/or surprising properties
- L = Lindenmayer, the inventor of these types of systems

L-Systems

- An L system is a set of grammar rules,
- Plus a *start symbol* (or an initial group of symbols)
- Plus *semantics* – a way of interpreting the grammar strings

E.g. here is a grammar:

rule 1: A → BBC

rule 2: B → abA

Here is a start string:

AAA

Here's what happens when we apply rules a few times:

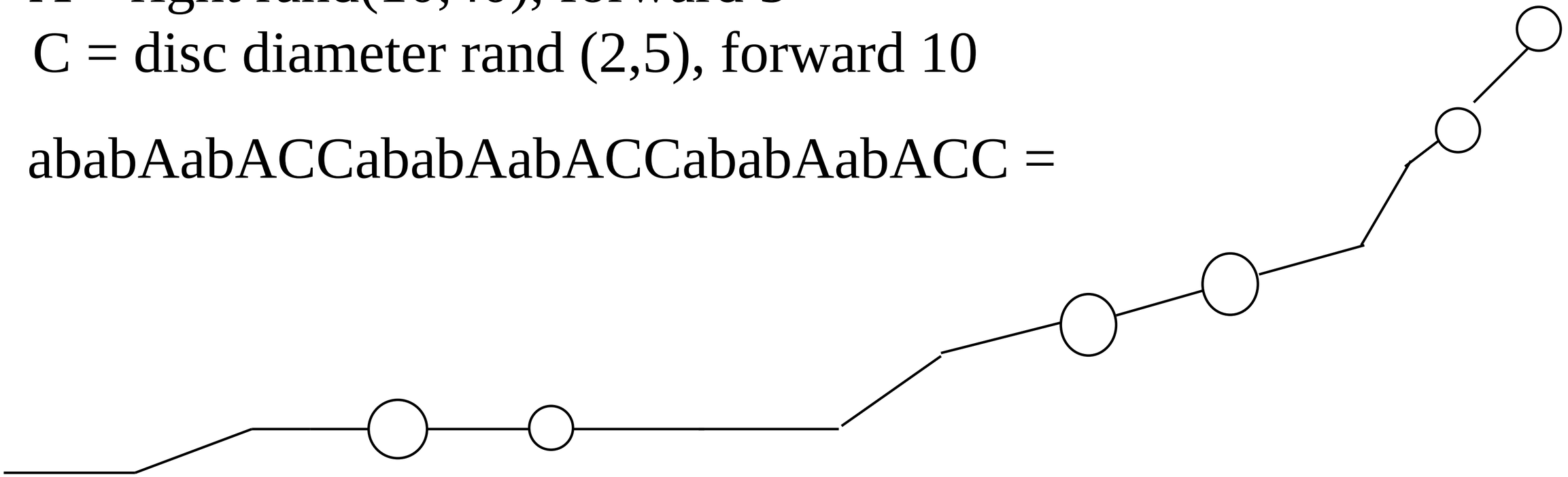
AAA → BBCBBCBBC → abACabACabAC →
abBBCCabBBCCabBBCC → ababAabACCababAabACCababAabACC
Etc …

L-Systems

AAA → BBCBBCBBC → abACabACabAC →
abBBCCabBBCCabBBCC → ababAabACCababAabACCababAabACC

Things become interesting when we add the *semantics*. In this case,
Let: a = forward 10
b = left rand(10,40)
A = right rand(10,40), forward 5
C = disc diameter rand (2,5), forward 10

Then ababAabACCababAabACCababAabACC =



A more interesting examples

- **Grammar symbols: {F,+,–}**
  - **Rules: F→F+F--F+F**
  - **Start symbol (or, *axiom*): F**
- **Note that when a rule is applied in an L system, it is applied simultaneously to all possible positions.**
- **In this case, after just 2 applications of the rule we get:**

F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F

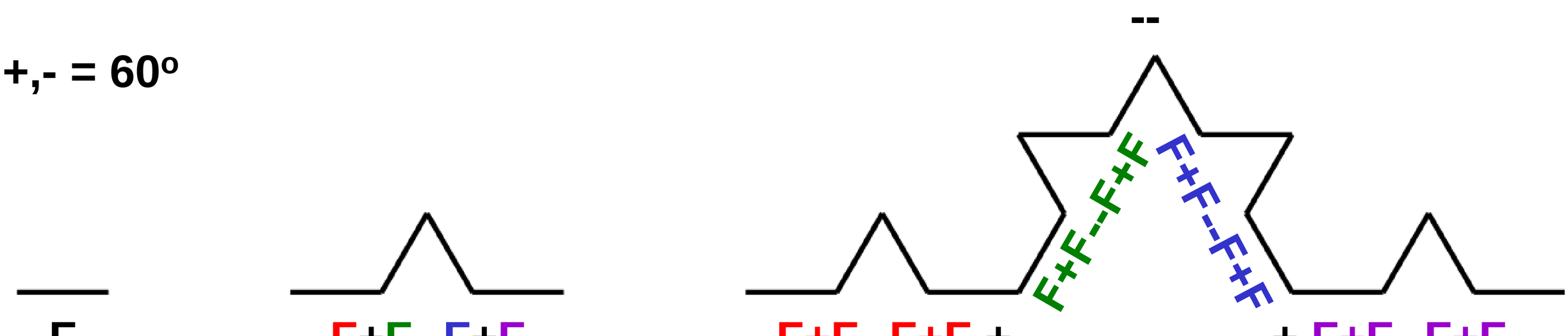
Now let:

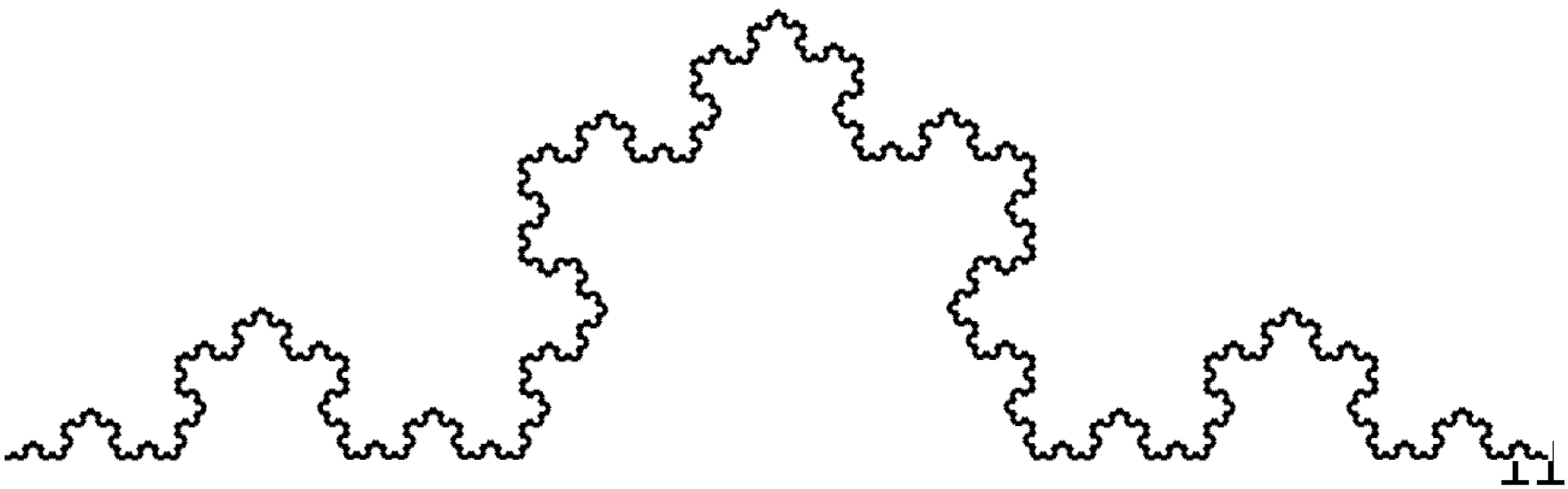
- **“F” – Draw forward one unit**
- **“+” – Rotate left**
- **“–” – Rotate right**

- **Units and angles are global constants**
- **Commands change “cursor” state (position, orientation)**

von Koch Snowflake Curve

+,- = 60º

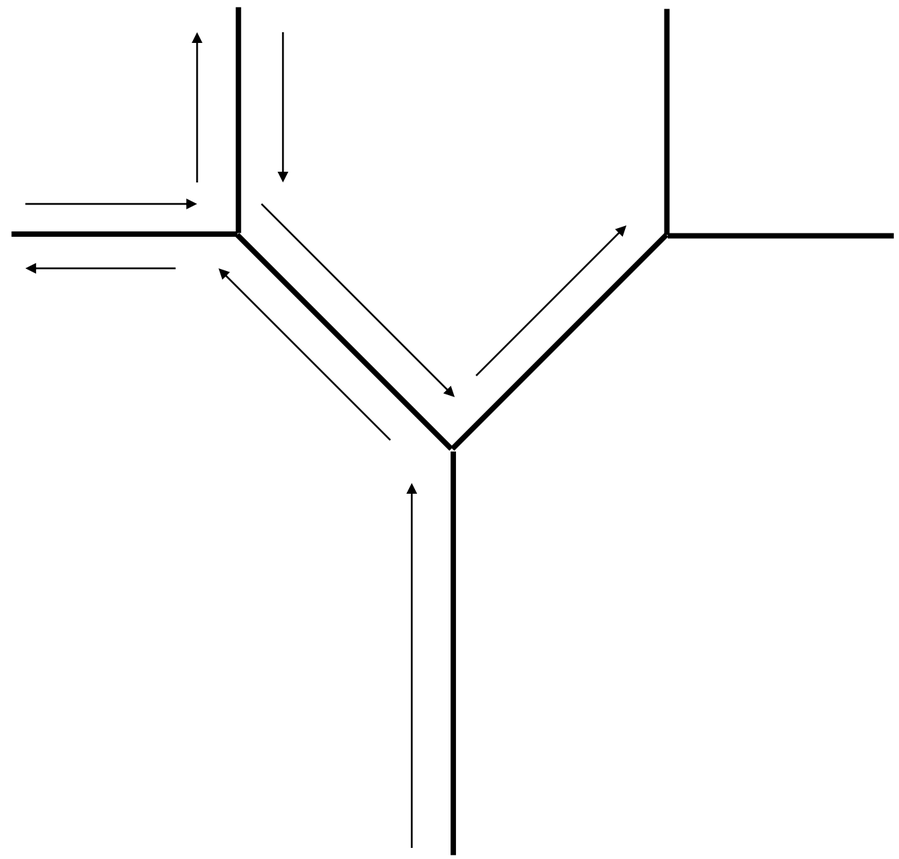




Branching structures

- **Assume:**
- **- “[” – pushes state onto stack**

  - “]” – pops state from stack
- **Use “[” to start a branch and “]” when finished to return to its base**
- **So include “[” and “]” in the grammar rule(s).**



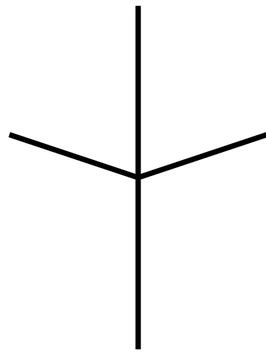
F+F+F----++F----+…

vs.

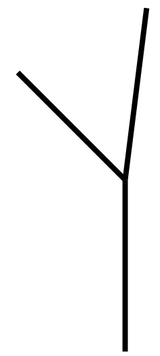
F[+F[+F][-F]][-F[+F][-F]]

Rules for Types of Branching

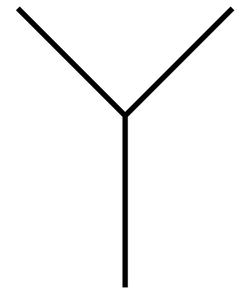
- **Monopodial**
  - **Trunk extends undeviated to top**
  - **Branches extend perpendicularly** T→T[+B][-B]T



- **Sympodial**
  - **Trunk deviates to top**
  - **Branches extend perpendicularly** T→T[---B]+T

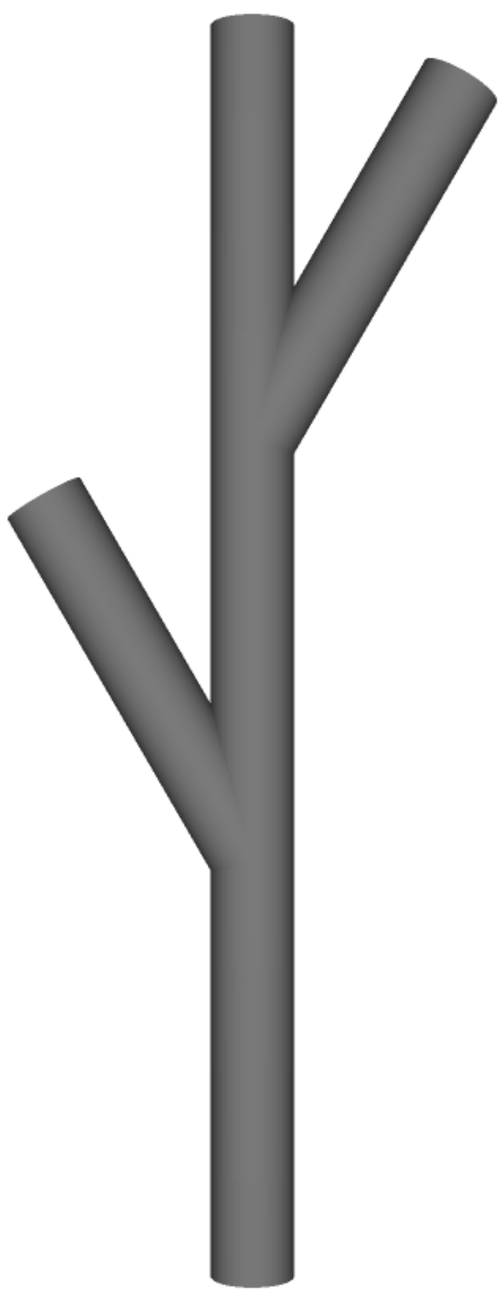


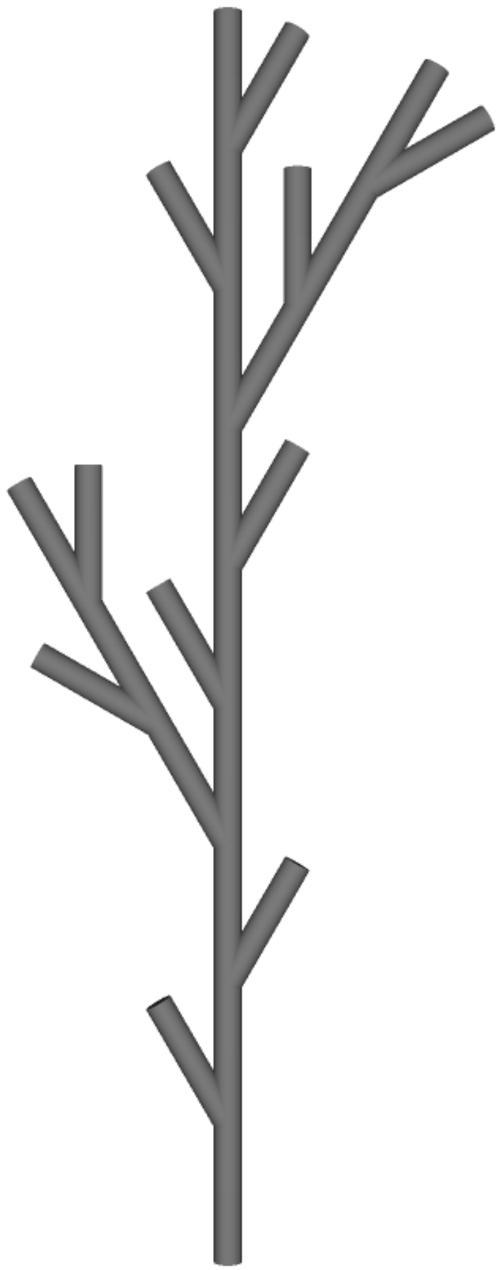
- **Binary**
  - **Trunk terminates at first branching point**
  - **Branches deviate uniformly** T→T[+B][-B]

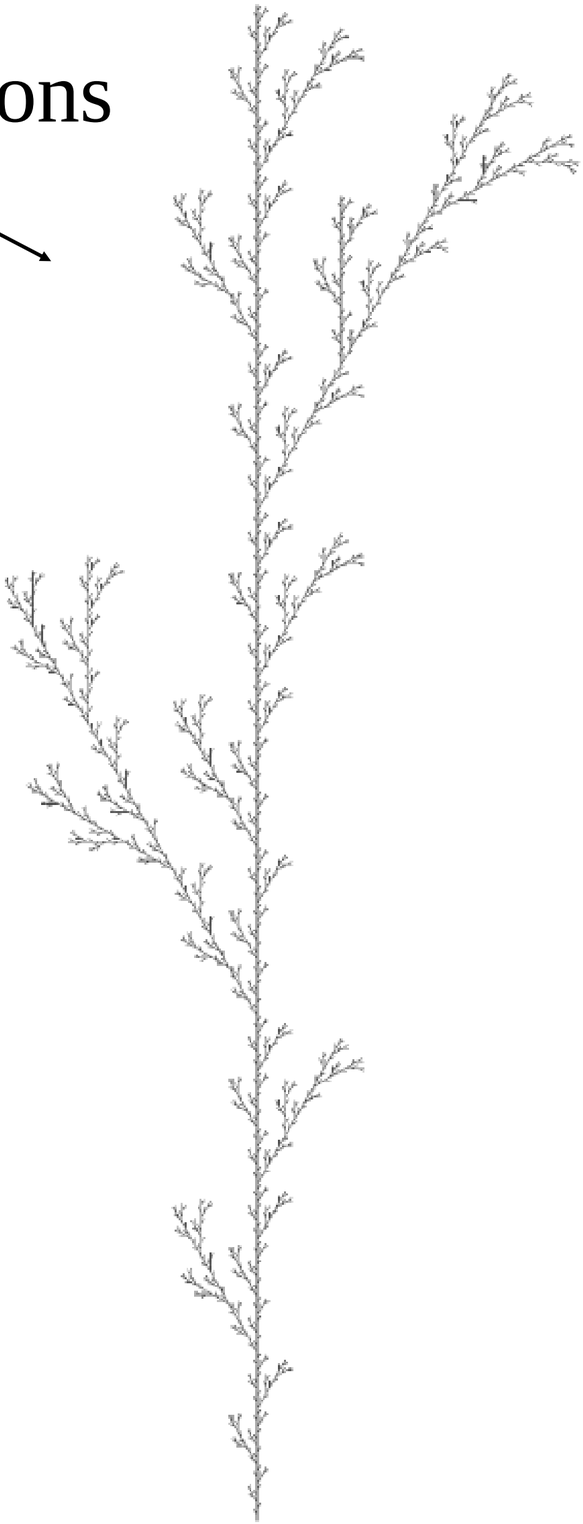


Example of this L system: F→F[+F]F[-F]F

Axiom = F. 1 application; 2 applications; 4 applications







Stochastic L-Systems

- **Conditional firing of rules**

  F → <something>: probability1

  F → <something2>: probability2

  etc …

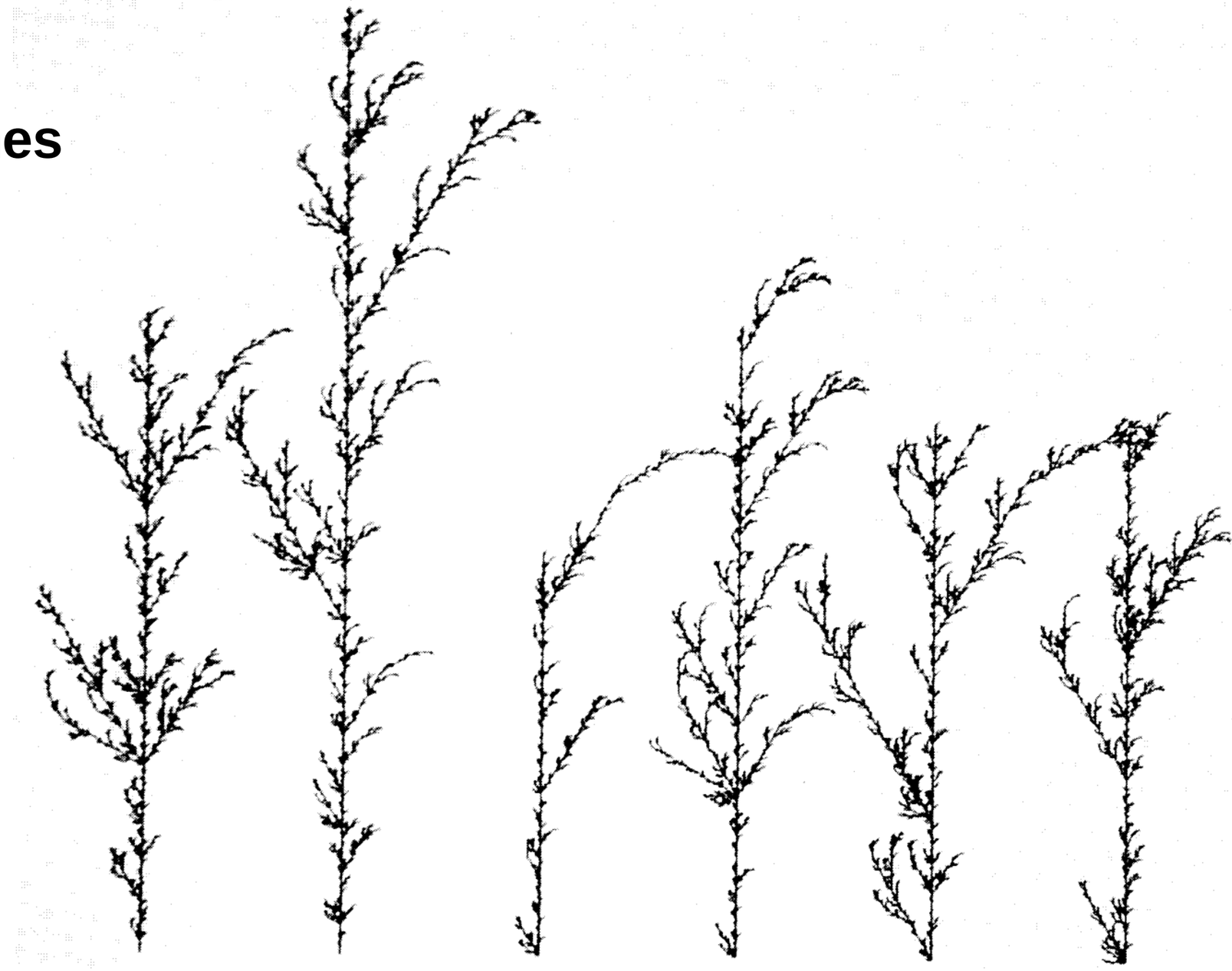
- **Example: Random Bushes**

  F → F[+F]F[-F]F : 0.33

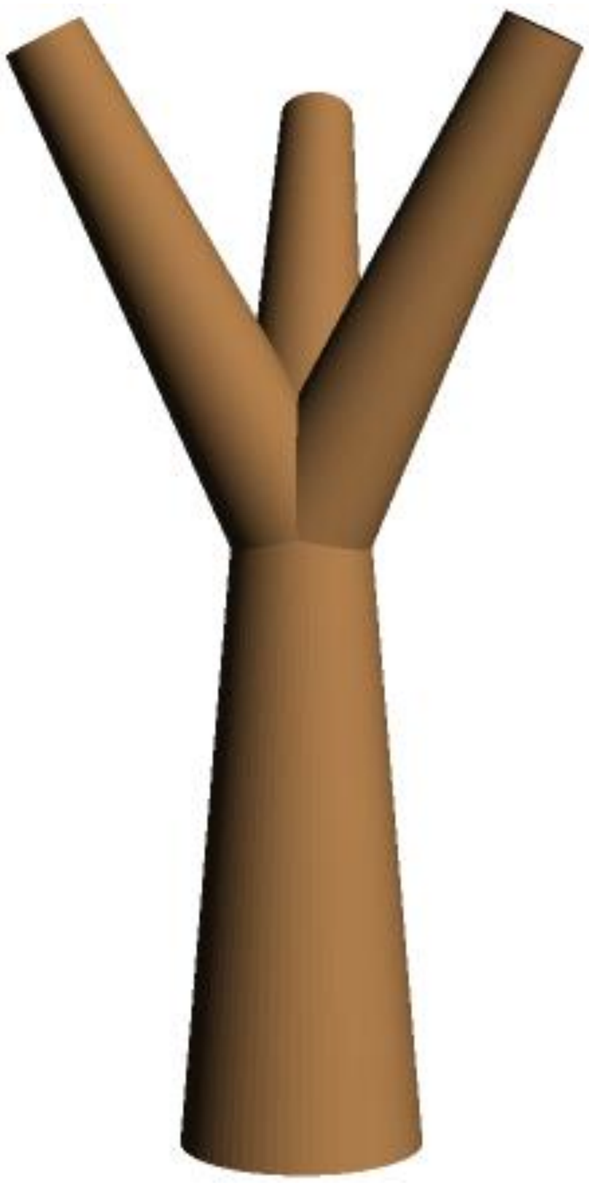
  F → F[+F]F : 0.33

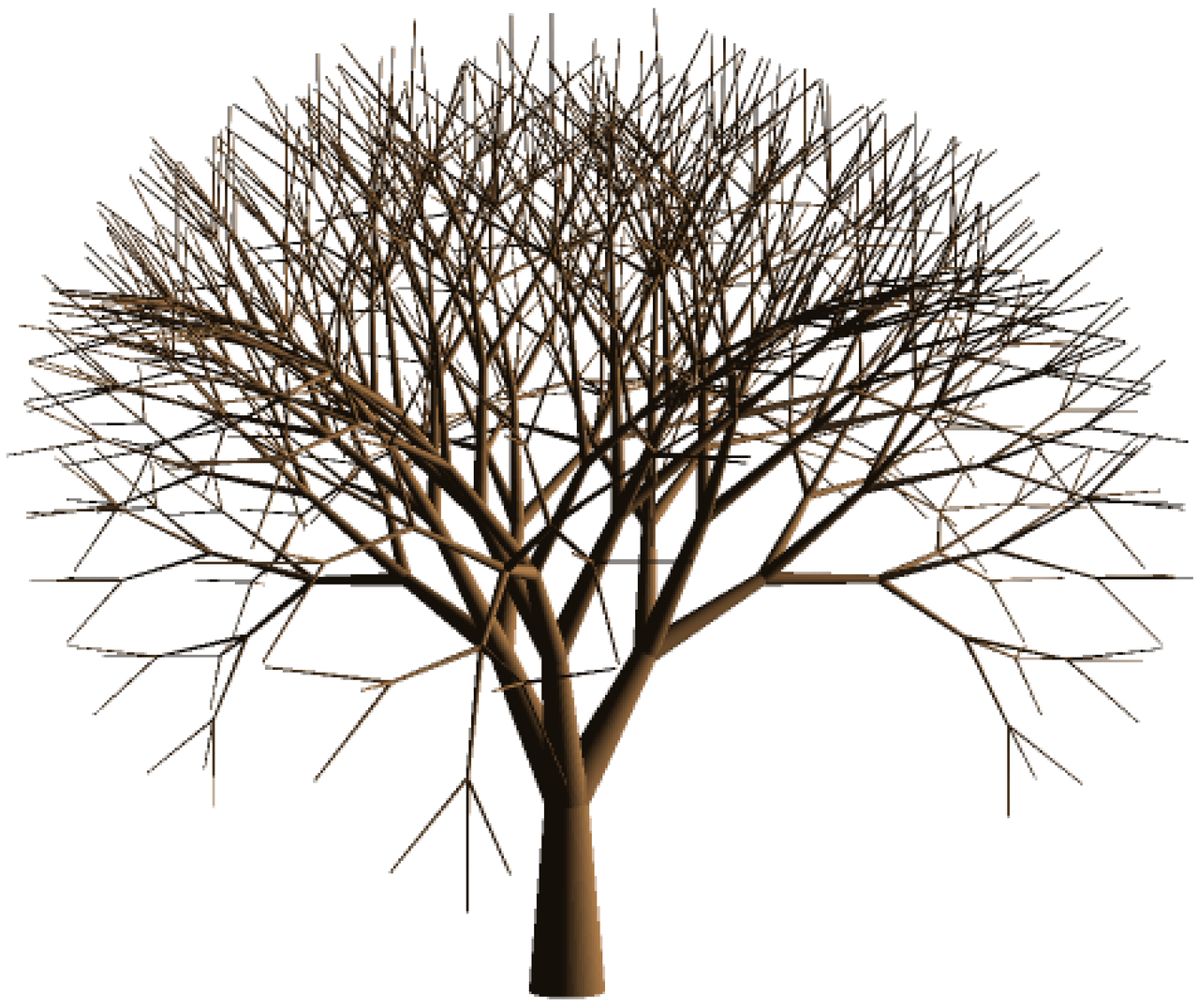
  F → F[-F]F : 0.34

- **If not stochastic, then *deterministic***

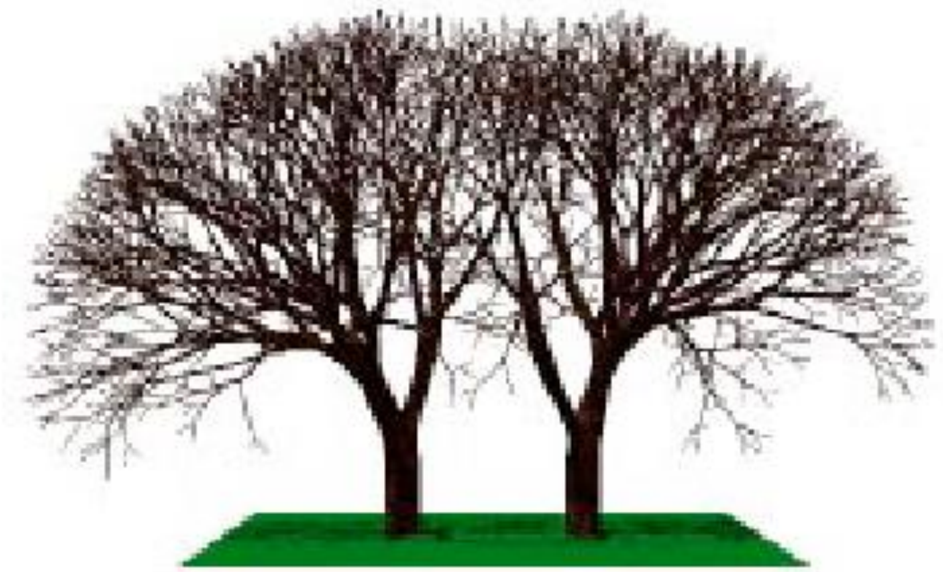


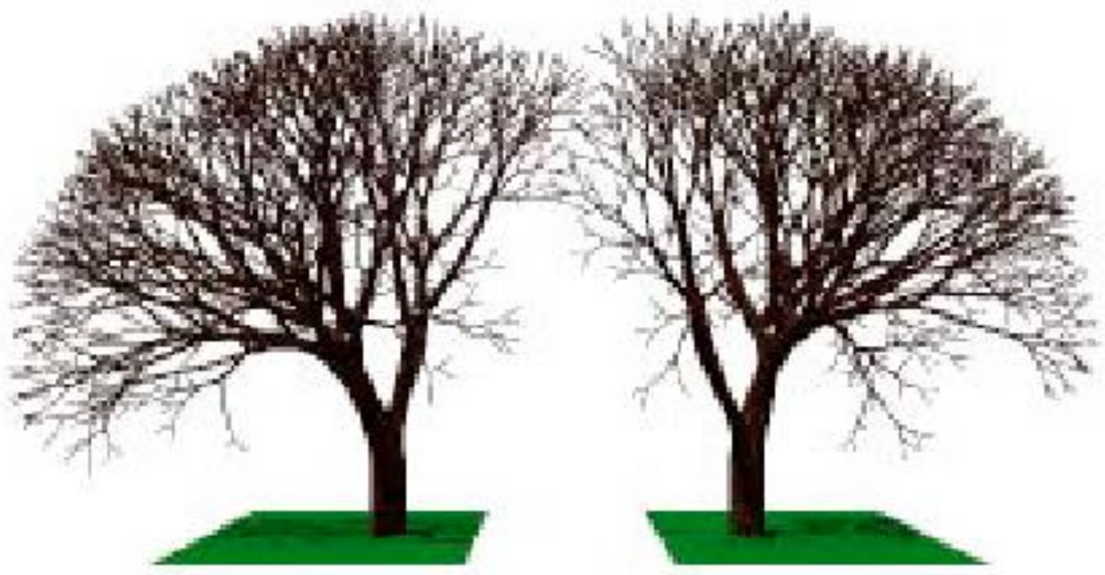
Ternary Tree, using only: F → F[&F][&/F][&\F]





Can get pretty complex





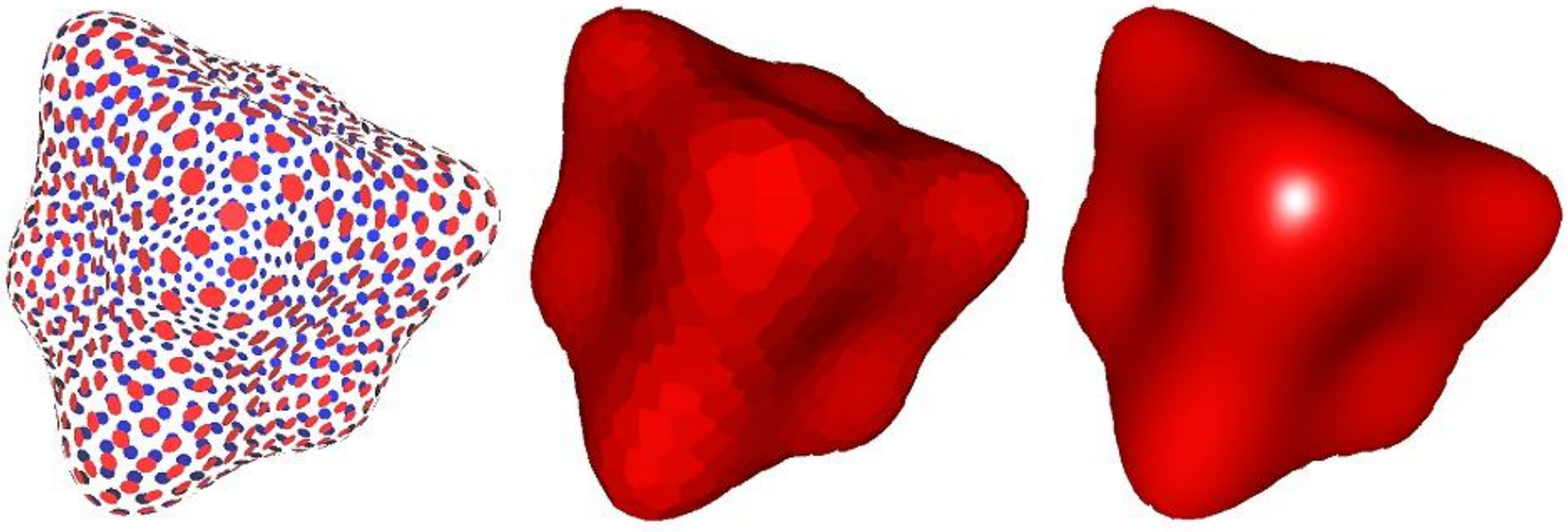
Implicit Representations - Pros & Cons

- **Pros:**
  - **easy to determine if a point is inside/outside (just plug it in!)**
  - **other queries may also be easy (e.g., distance to surface)**
  - **easy to handle changes in topology (e.g., fluid merging)**
- **Cons:**
  - **expensive to find all points in the shape (e.g., for drawing)**
  - ***May be difficult to model complex shapes***

What about explicit representations?

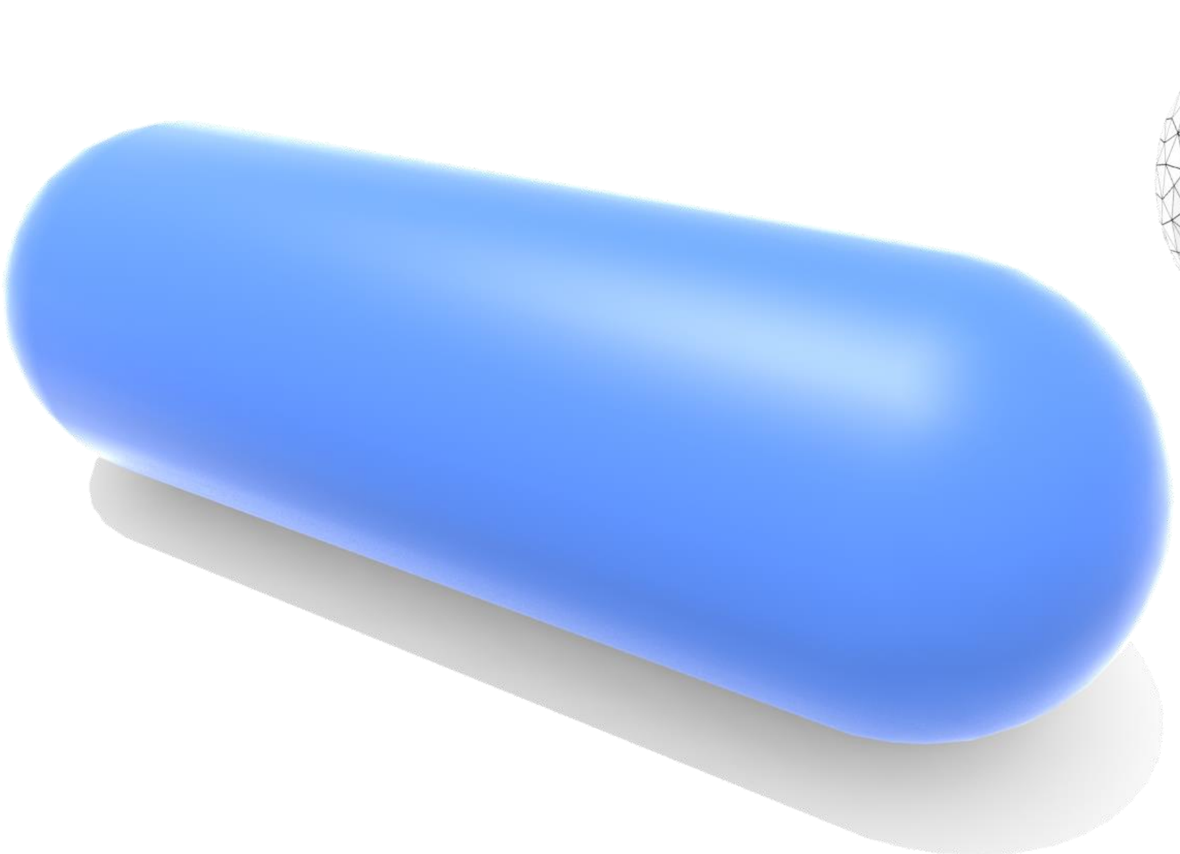
Point Cloud (Explicit)

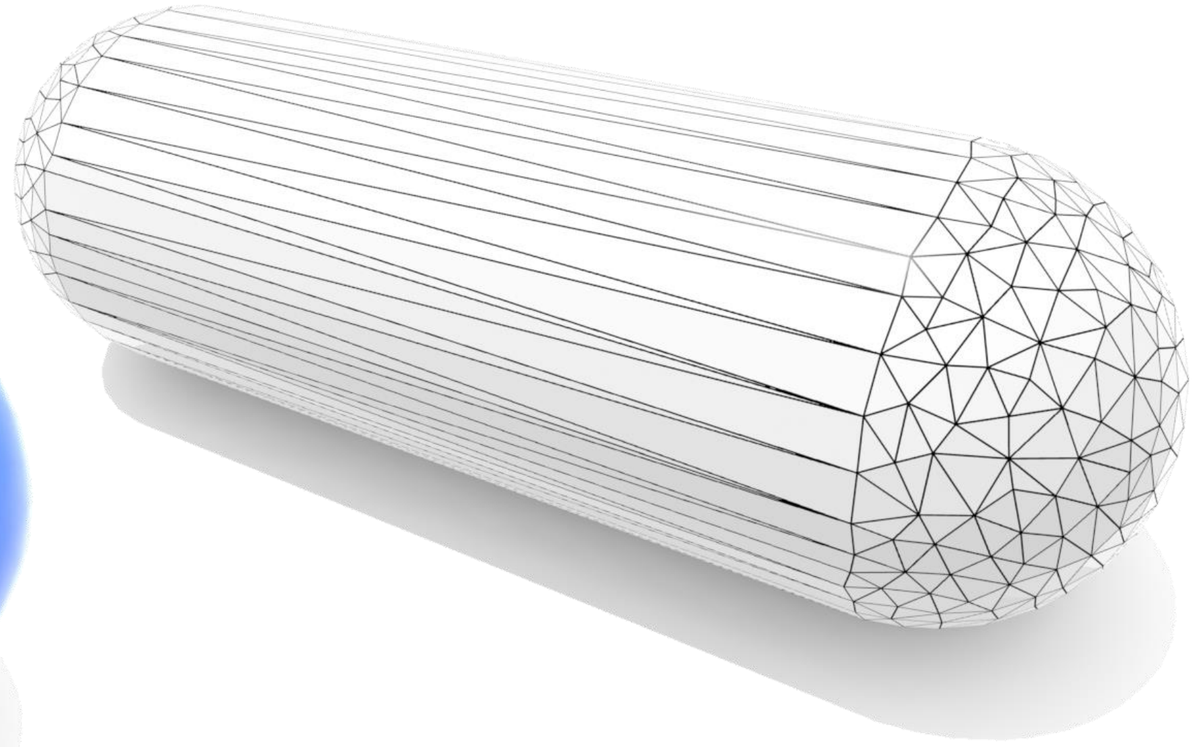
- **Simplest representation: list of points (x,y,z)**
- **Often augmented with *normals***
- **Easily represent any kind of geometry**
- **Useful for LARGE datasets (>>1 point/pixel)**
- **Difficult to draw in undersampled regions**
- **Hard to do processing / simulation**



Polygonal Mesh (Explicit)

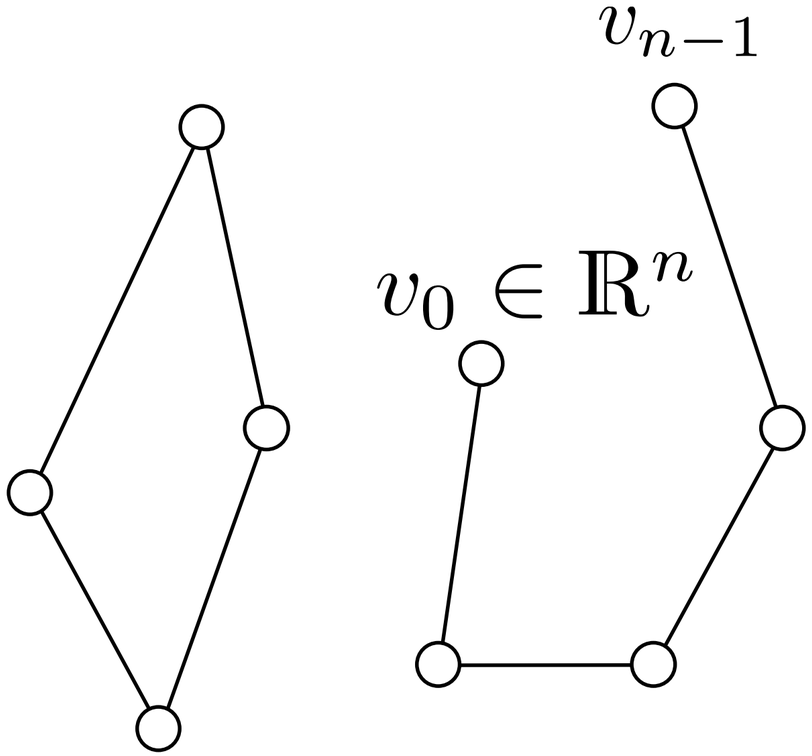
- **Store vertices *and* polygons (most often triangles or quads)**
- **Easier to do processing/simulation, adaptive sampling**
- **More complicated data structures**
- **Perhaps most common representation in graphics**





Polygonal Mesh

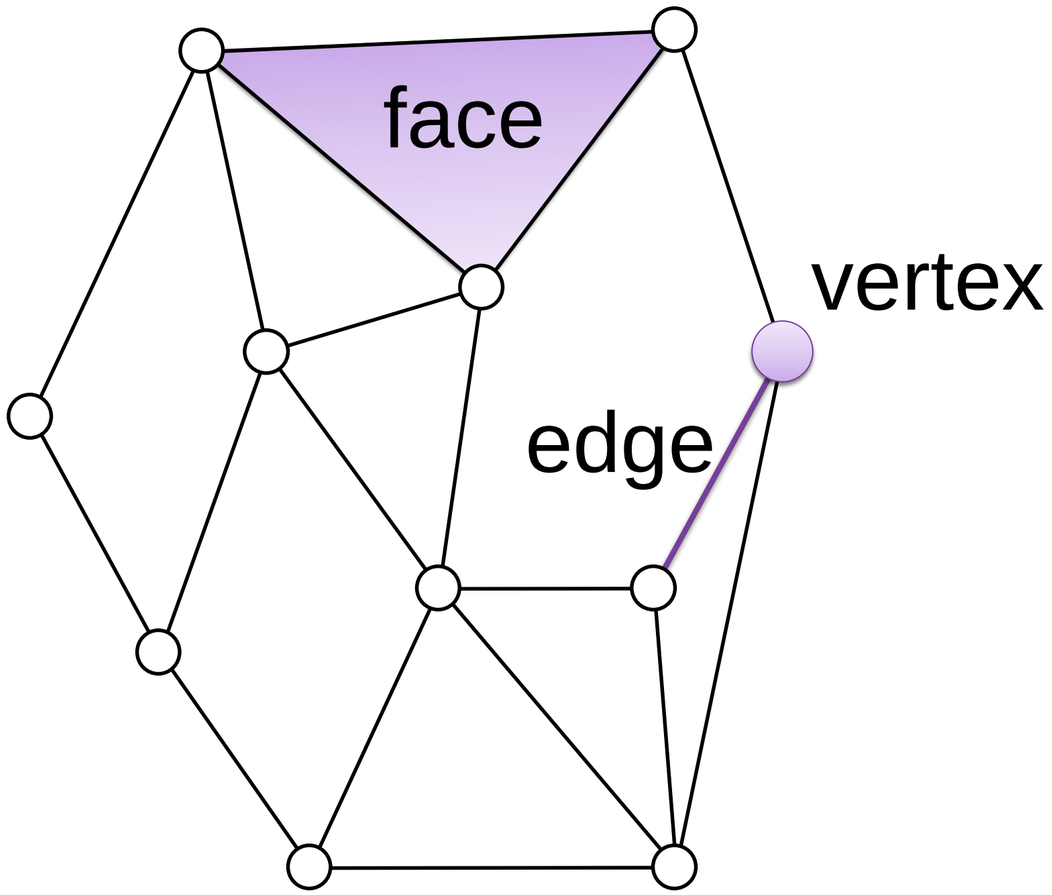
- **What is a polygon**



- Vertices
  $v_0, v_1, \ldots, v_{n-1}$
- Edges
  $\{(v_0, v_1), \ldots, (v_{n-2}, v_{n-1})\}$
- Planar and non-self-intersecting

Polygonal Mesh

- **Set of connected polygons**

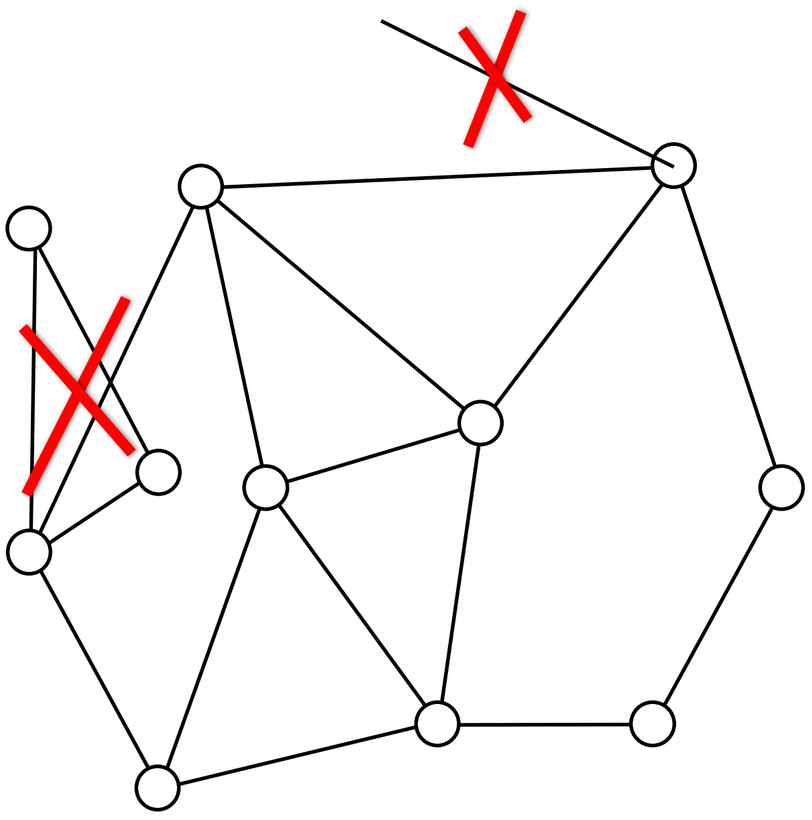


$$M = \langle V, E, F \rangle$$

vertices edges faces

Polygonal Mesh

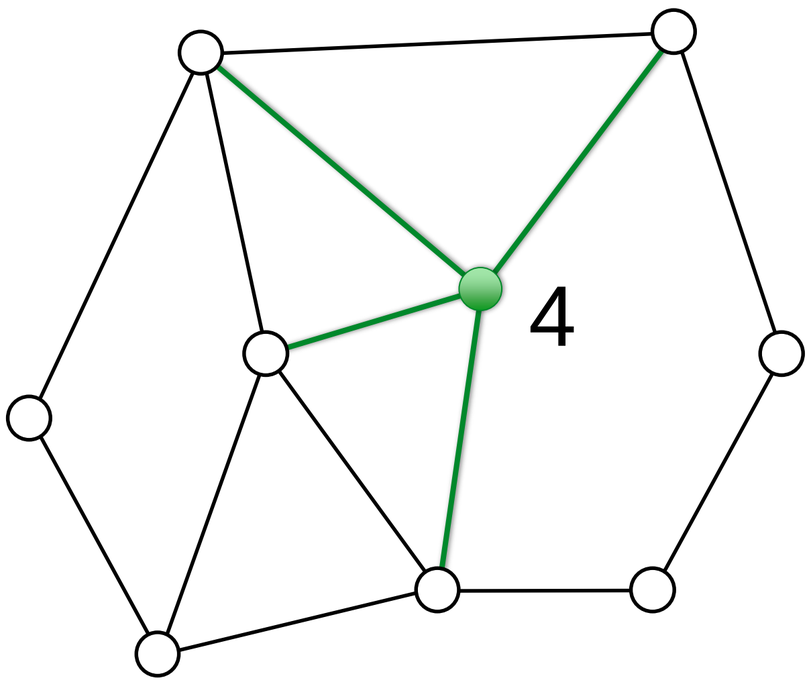
- **Properties**



- Every edge belongs to at least one polygon
- The intersection of two polygons in M is either empty, a vertex, or an edge

Polygonal Mesh

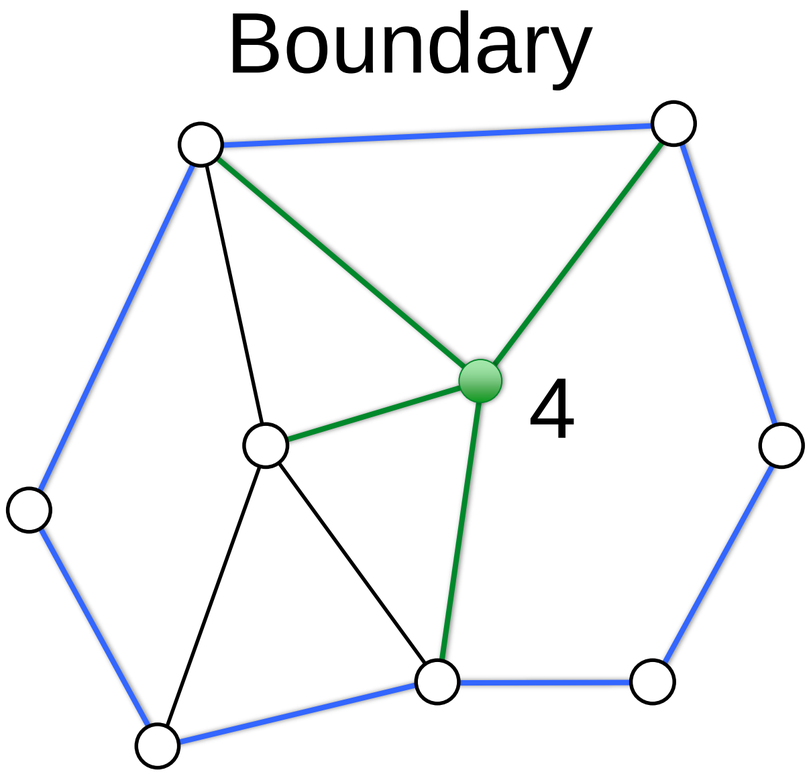
- **Definitions**



- **Vertex degree (Valence)**: Number of edges incident to a vertex

Polygonal Mesh

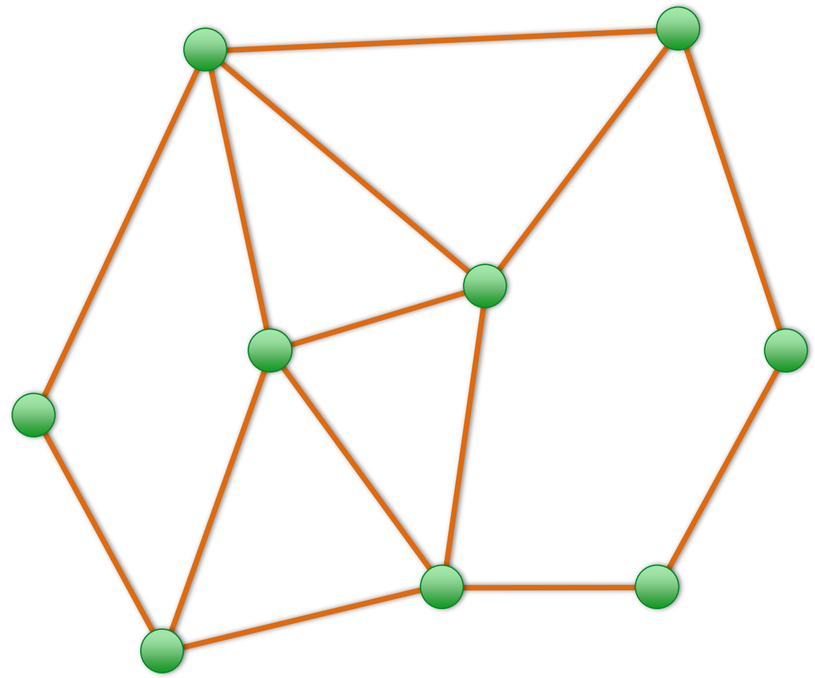
- **Definitions**



- **Vertex degree (valence)**: Number of edges incident to a vertex
- **Boundary**: the set of all edges that belong to only one polygon

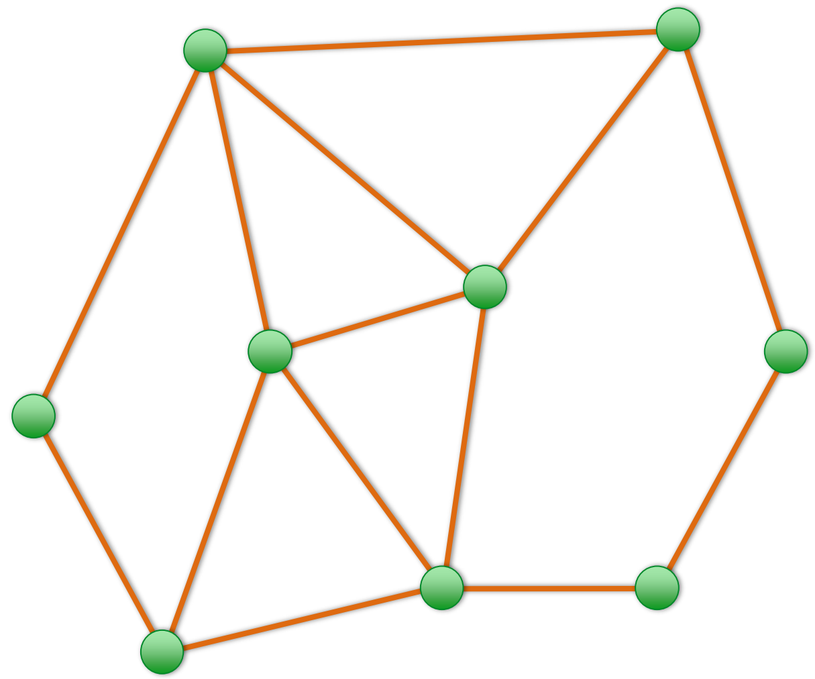
Mesh Data Structures

- Store geometry & topology
  - **Geometry**: vertex locations
  - **Topology**: how vertices are connected (edges/faces)
- Attributes: Normal, color, etc.



Mesh Data Structures

- Operations to be supported
  - Rendering

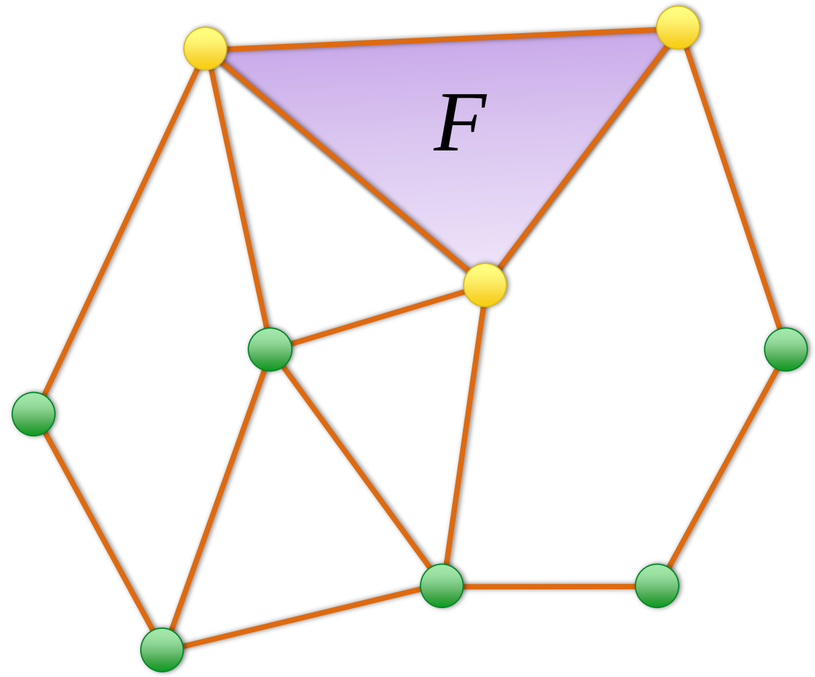


Mesh Data Structures

- **Operations to be supported**
  - **Rendering**
  - **Geometry queries**

Example:

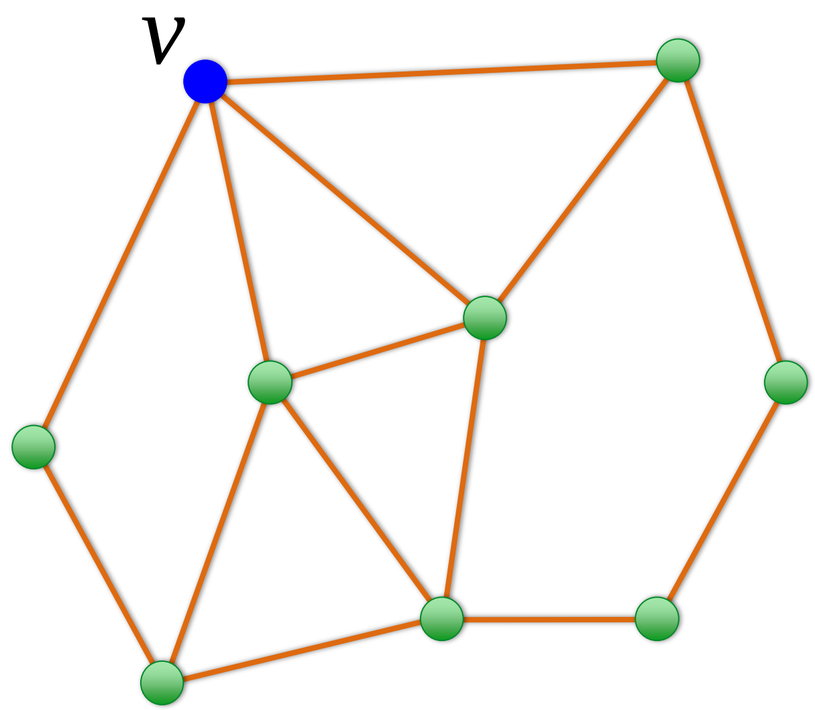
What are the vertices of face F ?



Mesh Data Structures

- **Operations to be supported**
  - **Rendering**
  - **Geometry queries**
  - **Modifications**

Example:
Remove vertex v.



Mesh Data Structures

- **Triangle List**

Triangles		
Vertex Coord.	**Vertex Coord.**	**Vertex Coord.**
(x_0, y_0, z_0)	**(x_1, y_1, z_1)**	**(x_2, y_2, z_2)**
(x_3, y_3, z_3)	**(x_4, y_4, z_4)**	**(x_5, y_5, z_5)**
...	**...**	**...**

- Simple
- No connectivity
- Redundant
- STL file format

Mesh Data Structures

- **Indexed Face Set**

Triangles		
Vertex Index	**Vertex Index**	**Vertex Index**
1	2	0
…	**…**	**…**

Vertices	
Index	Coord.
0	**(x_0, y_0, z_0)**
1	**(x_1, y_1, z_1)**
2	**(x_2, y_2, z_2)**
…	**…**

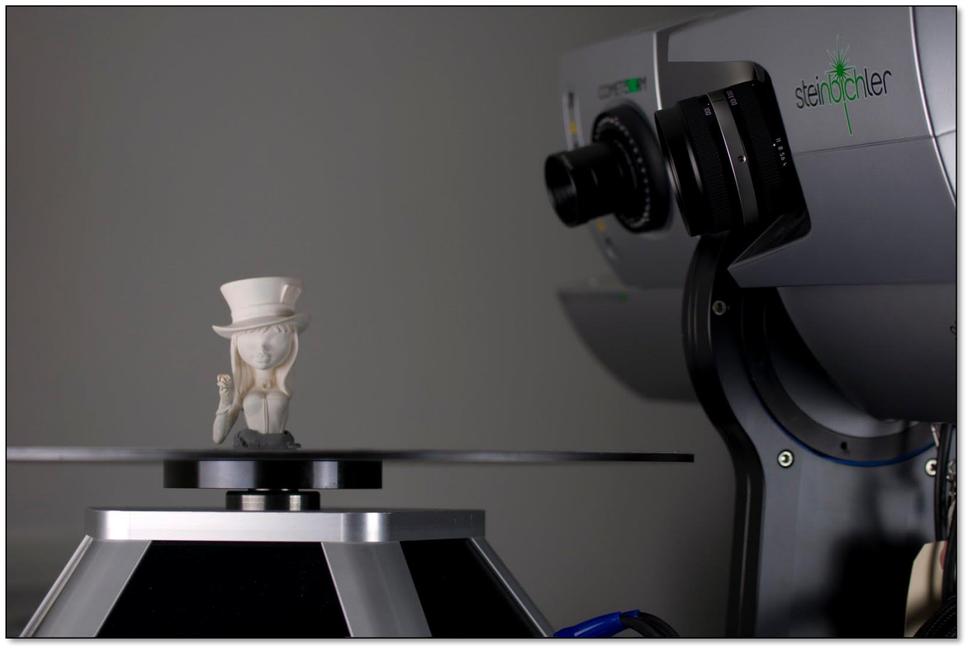
Mesh Data Structures

- **Indexed Face Set**
  - **Avoids redundancy**
  - **OBJ, OFF, WRL file formats**
  - **Stores connectivity, but still**
    - **Costly geometric queries**
    - **Costly mesh modifications**
- **More exotic data structures (e.g. half edge data structure) address these limitations**

Where is all this data coming from?

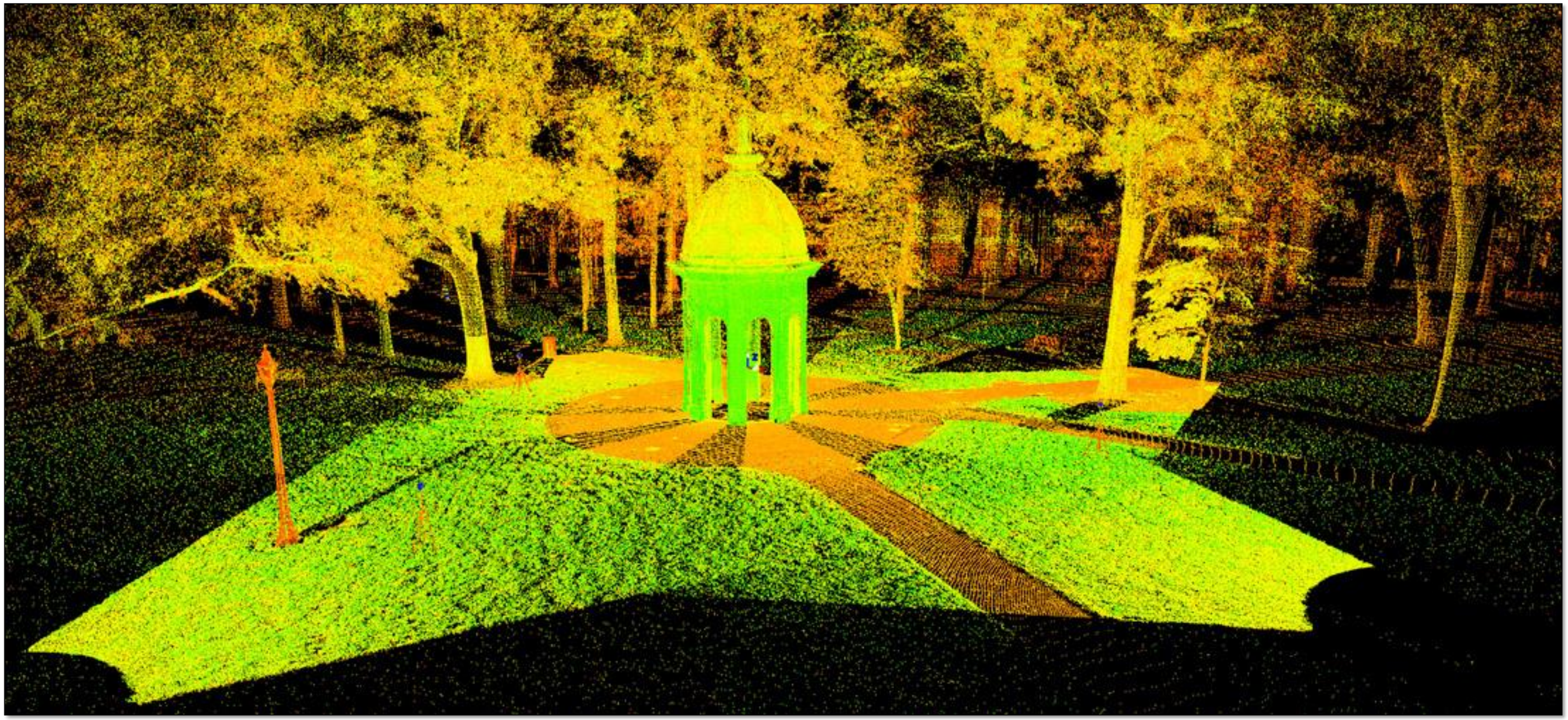
Sources of geometry

- **Acquired real-world objects via 3D Scanning**



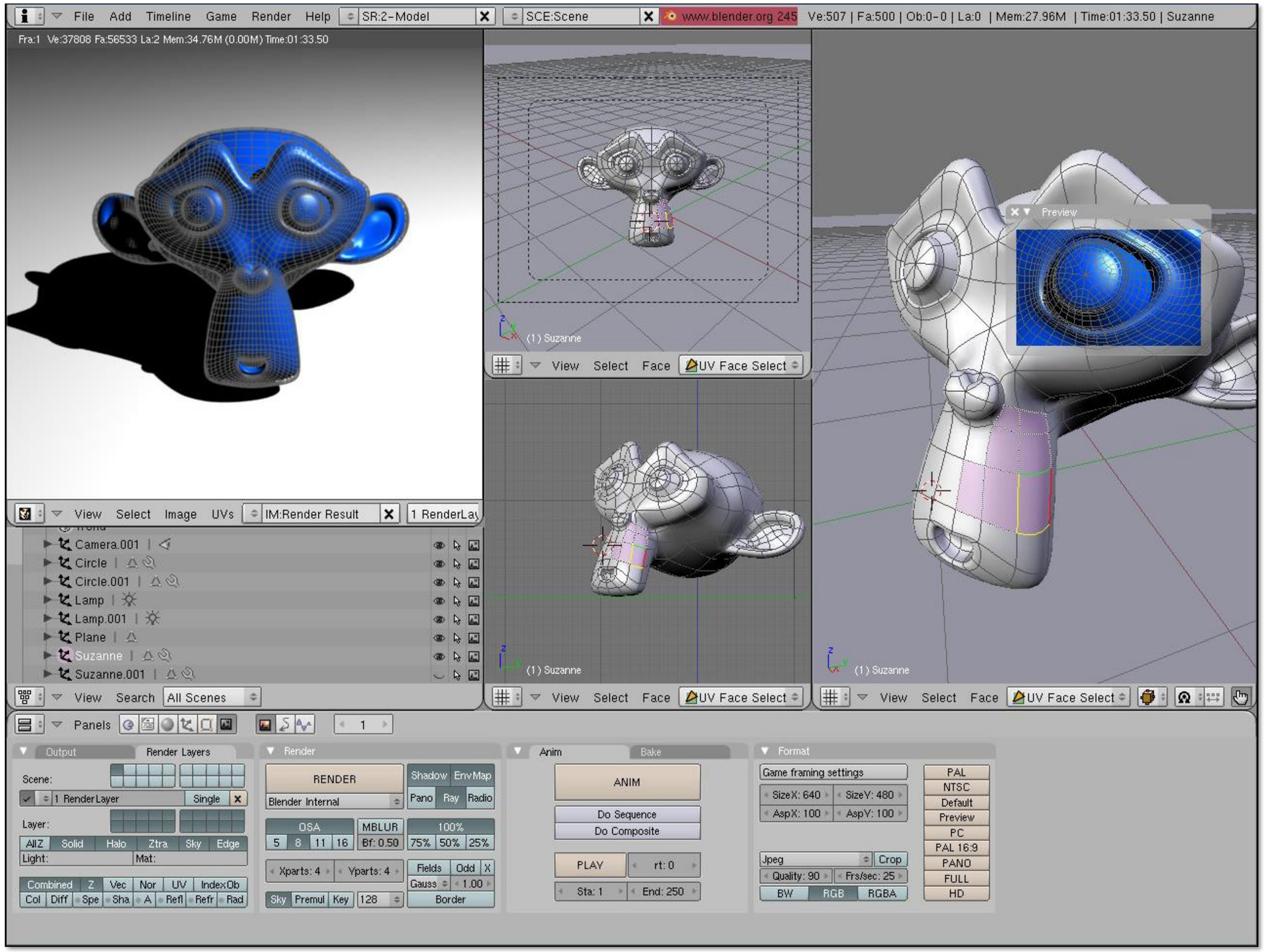
Sources of geometry

- **Acquired real-world objects via 3D Scanning**



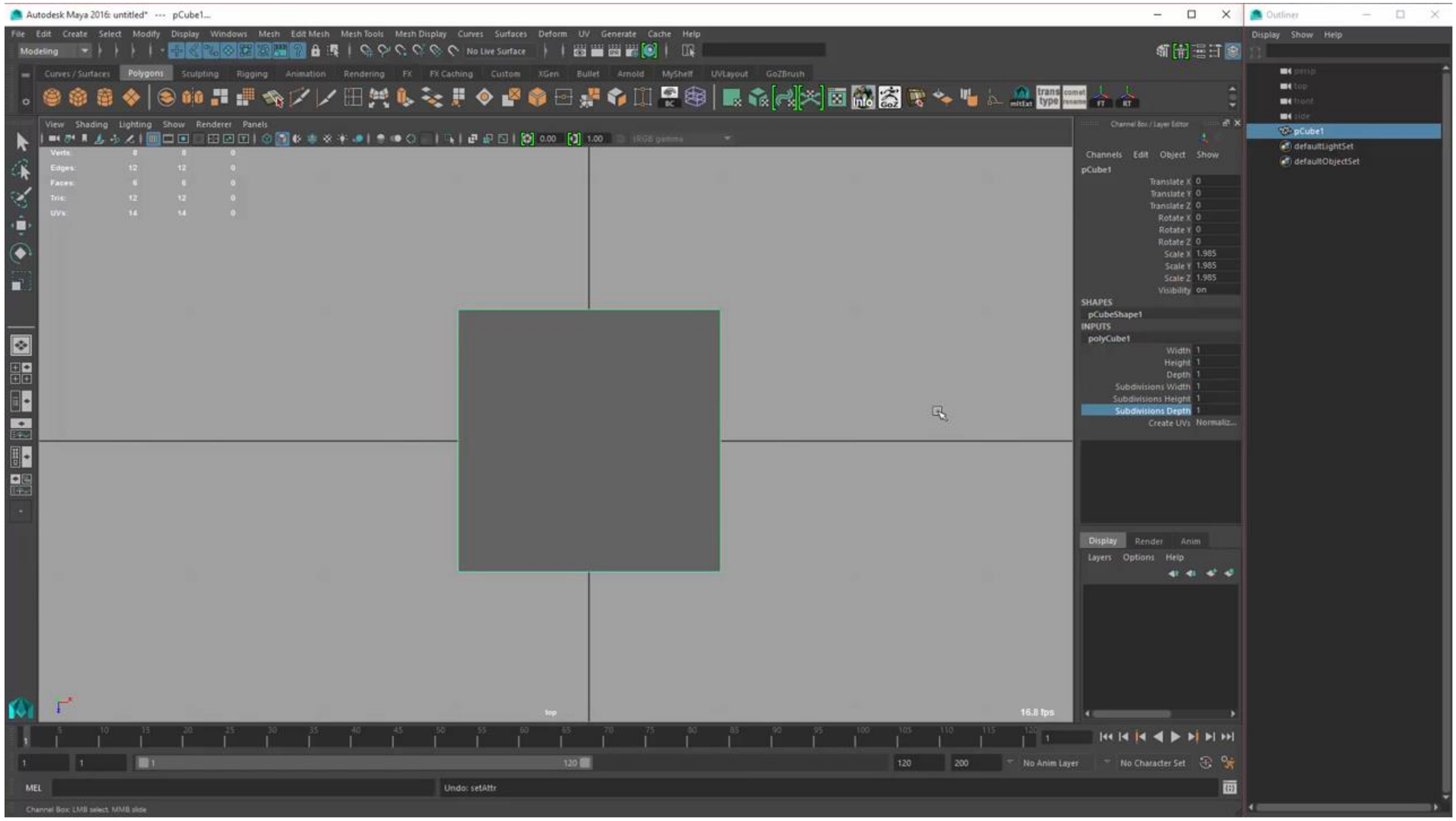
Sources of geometry

- **Digital 3D modeling**



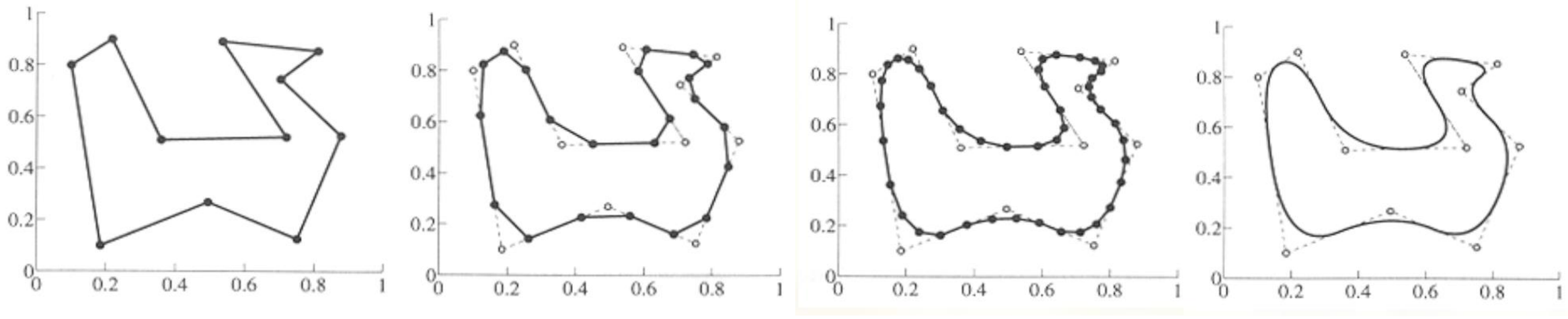
Sources of geometry

- **Digital 3D modeling**



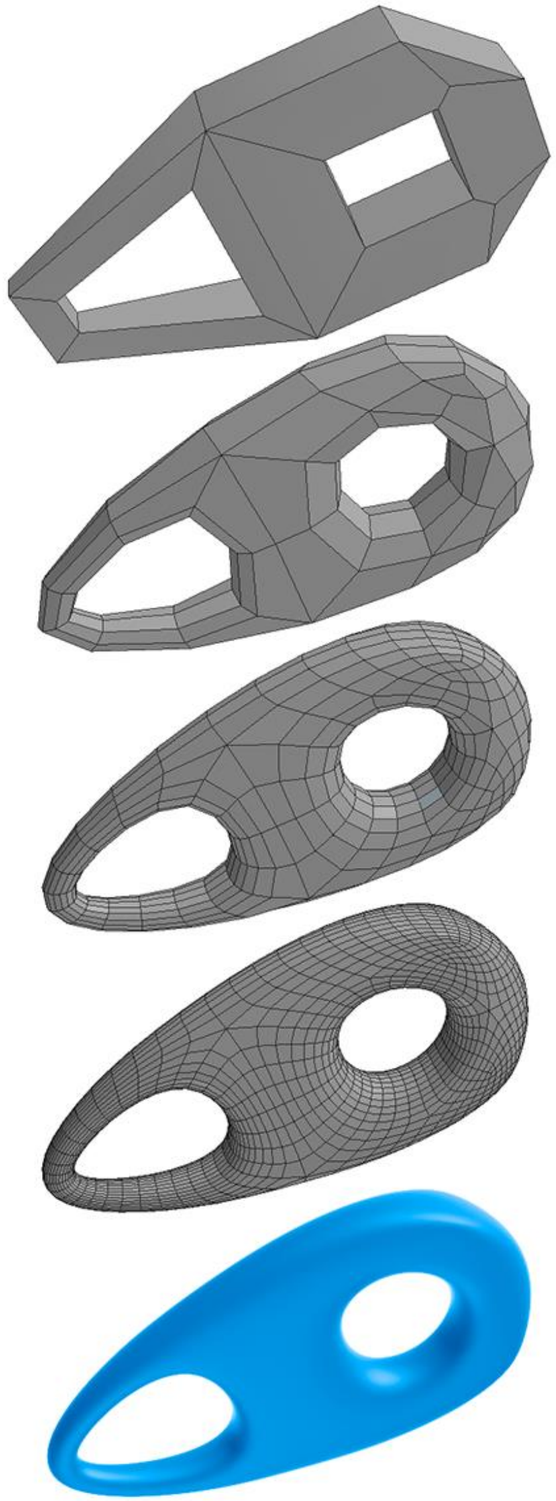
Subdivision Surfaces (Explicit)

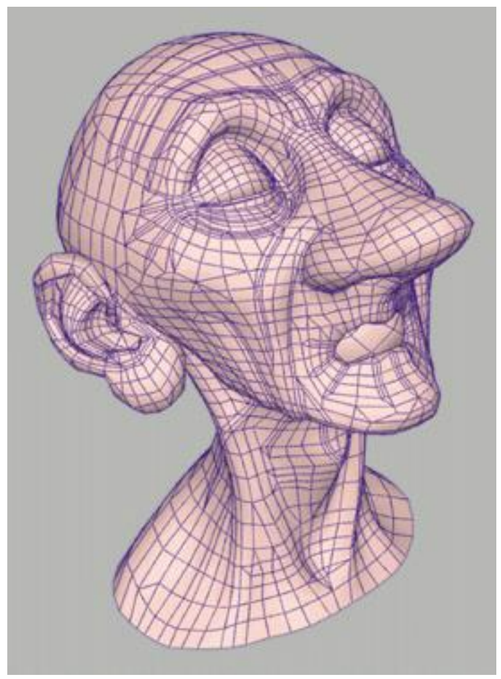
- Smooth out a control curve
  - Insert new vertex at each edge midpoint
  - Update vertex positions according to fixed rule
  - For careful choice of averaging rule, yields smooth curve
    - E.g. average with “next” neighbor (Chaikin)



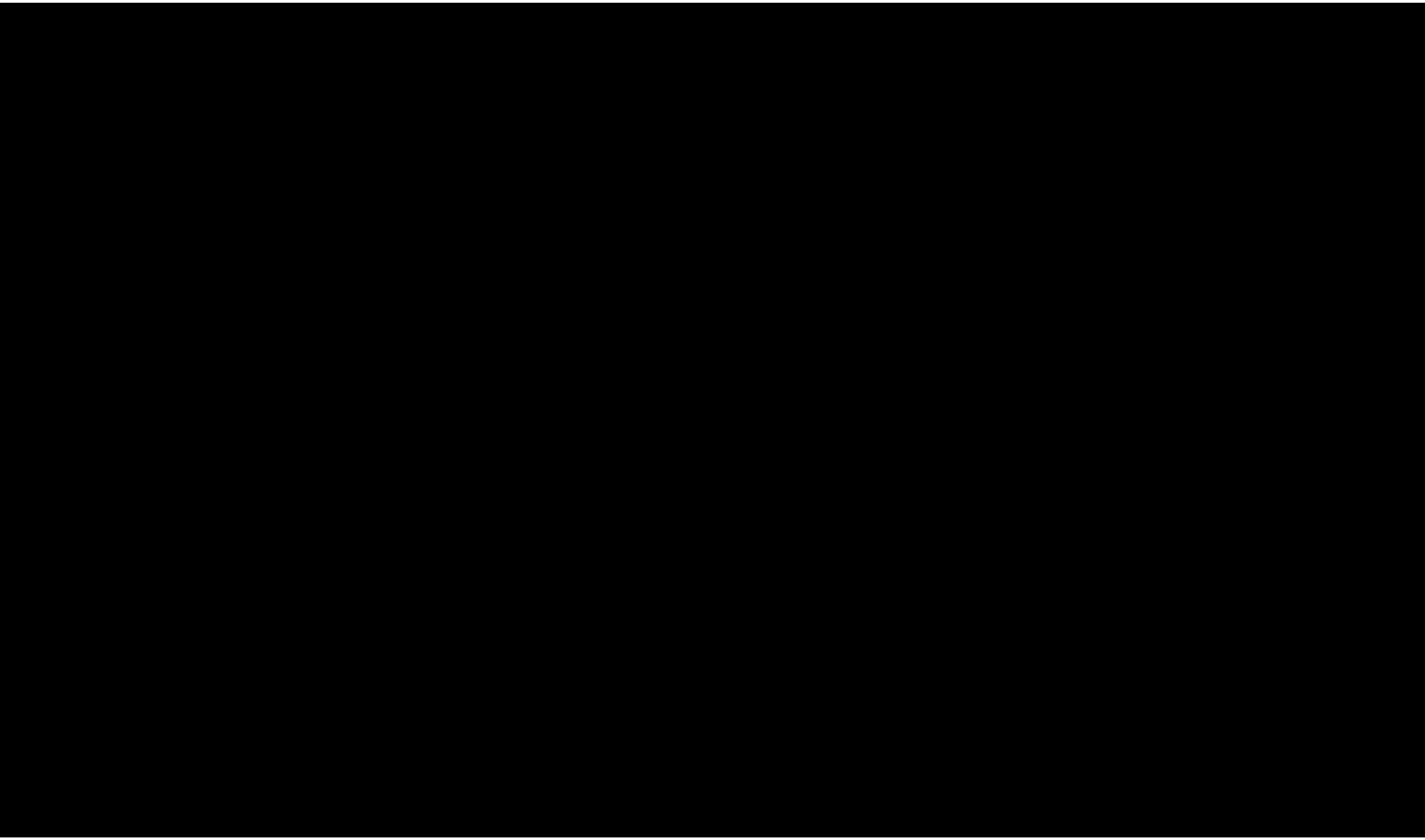
Subdivision Surfaces (Explicit)

- Start with coarse polygon mesh (“control cage”)
- Subdivide each element
- Update vertices via local averaging
- Many possible rule:
  - Catmull-Clark (quads)
  - Loop (triangles)
  - ...
- Common issues:
  - interpolating or approximating?
  - continuity at vertices?





Subdivision in Action (Pixar's "Geri's Game")

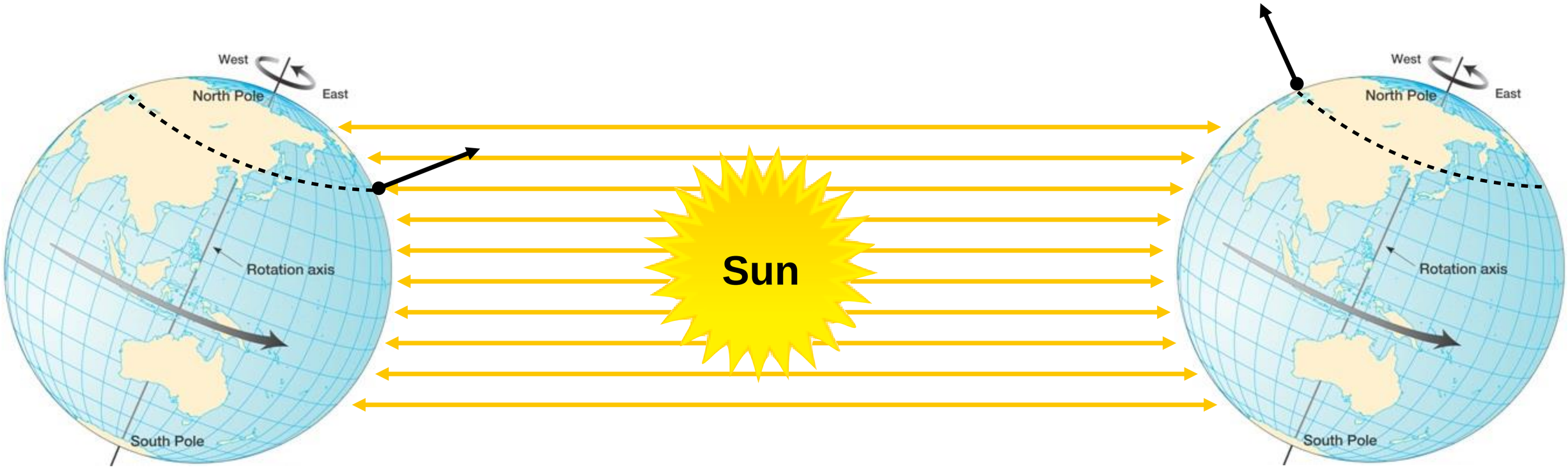


So, that’s where triangles come from…

About those surface normals…

Why do some parts of a surface look lighter or darker?

Why do we have seasons?



Summer
(Northern hemisphere)

Winter
(Northern hemisphere)

Earth’s axis of rotation: ~23.5° off axis

[Image credit: Pearson Prentice Hall]

Beam power in terms of irradiance

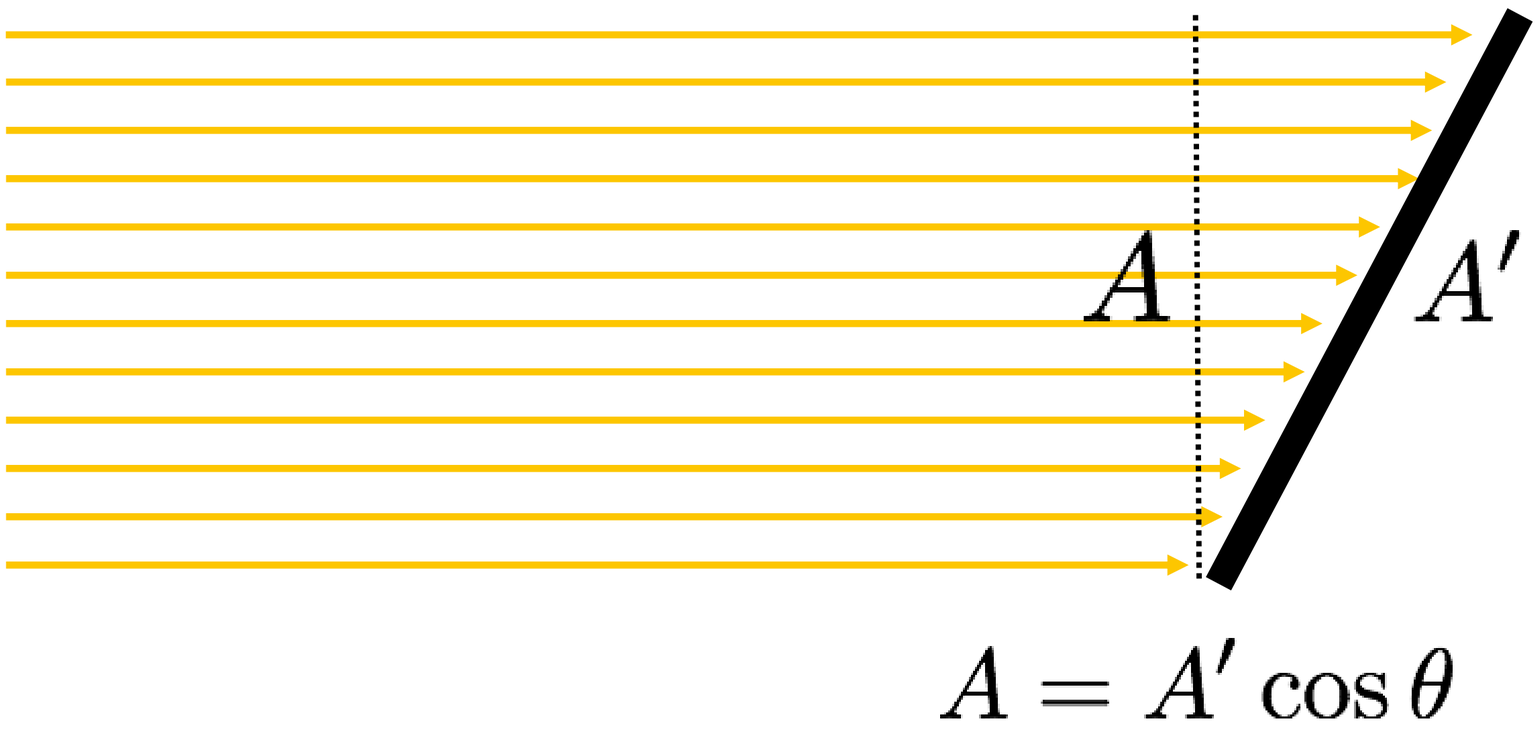
Consider beam of light with flux Φ incident on surface with area *A*

Irradiance
(energy per time, per area)

$$E = \frac{\Phi}{A}$$

Projected area

Consider beam with flux Φ incident on angled surface with area *A′*



A = projected area of surface relative to direction of beam

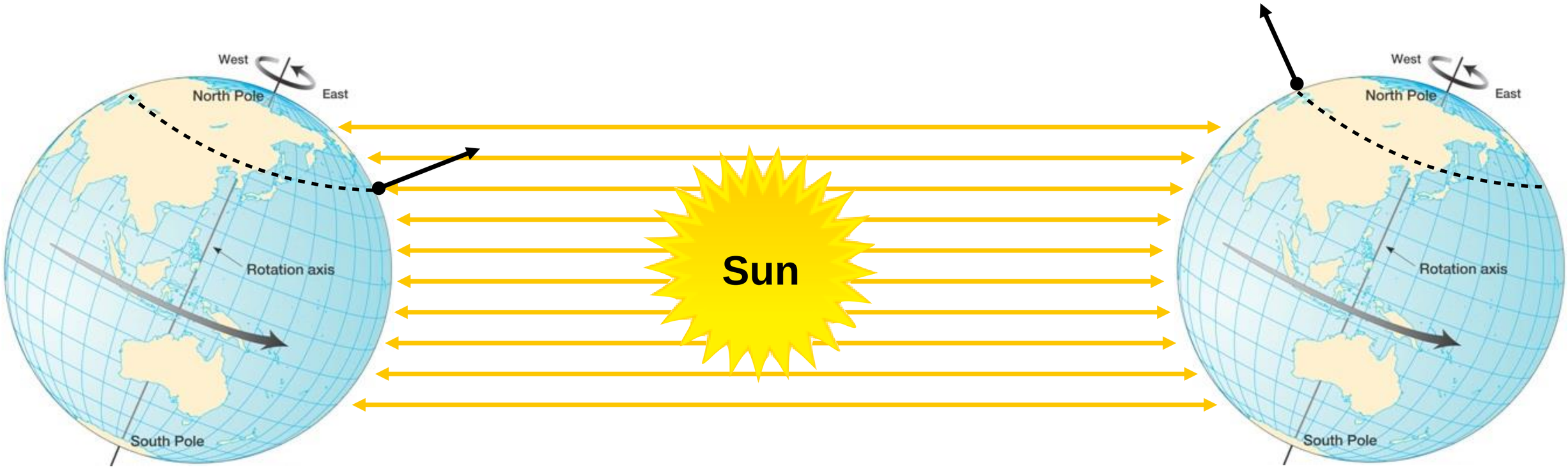
Lambert's Law

Irradiance at surface is proportional to cosine of angle between light direction and surface normal.

$$A = A' \cos\theta$$

$$E = \frac{\Phi}{A'} = \frac{\Phi \cos\theta}{A}$$

Why do we have seasons?



Summer
(Northern hemisphere)

Winter
(Northern hemisphere)

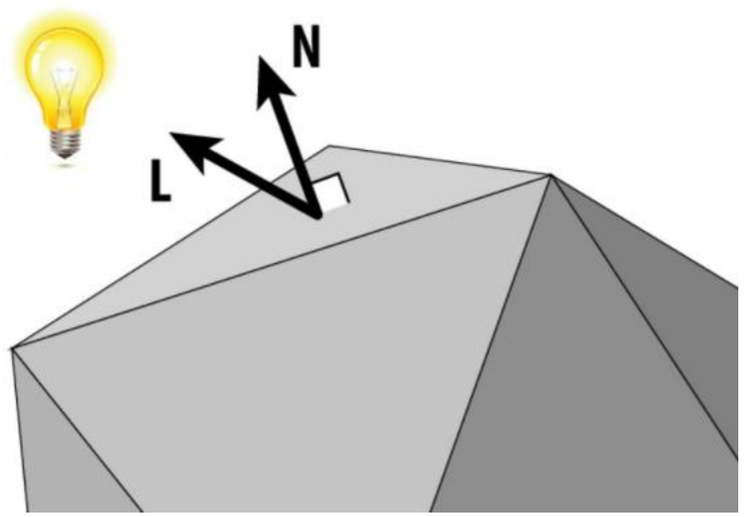
Earth’s axis of rotation: ~23.5° off axis

[Image credit: Pearson Prentice Hall]

N-dot-L lighting

- **Most basic way to shade a surface: take dot product of unit surface normal (N) and unit direction to light (L)**

```
double surfaceColor( Vec3 N, Vec3 L )
{
    return dot( N, L );
}
```

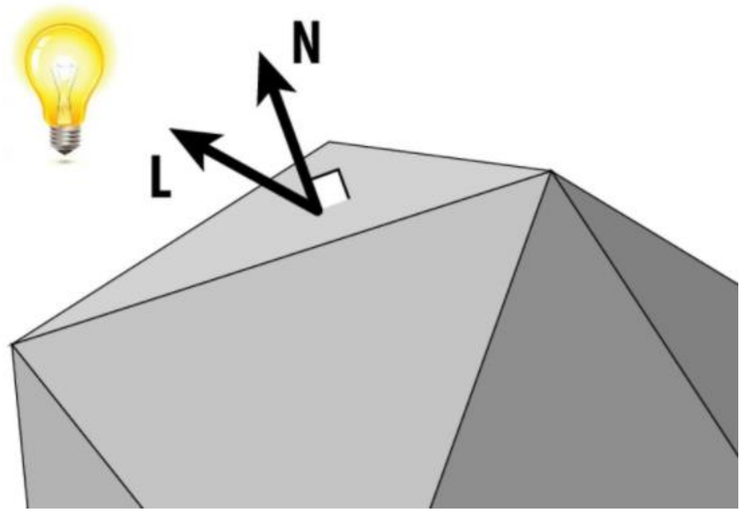


(Q: What's wrong with this code?)

N-dot-L lighting

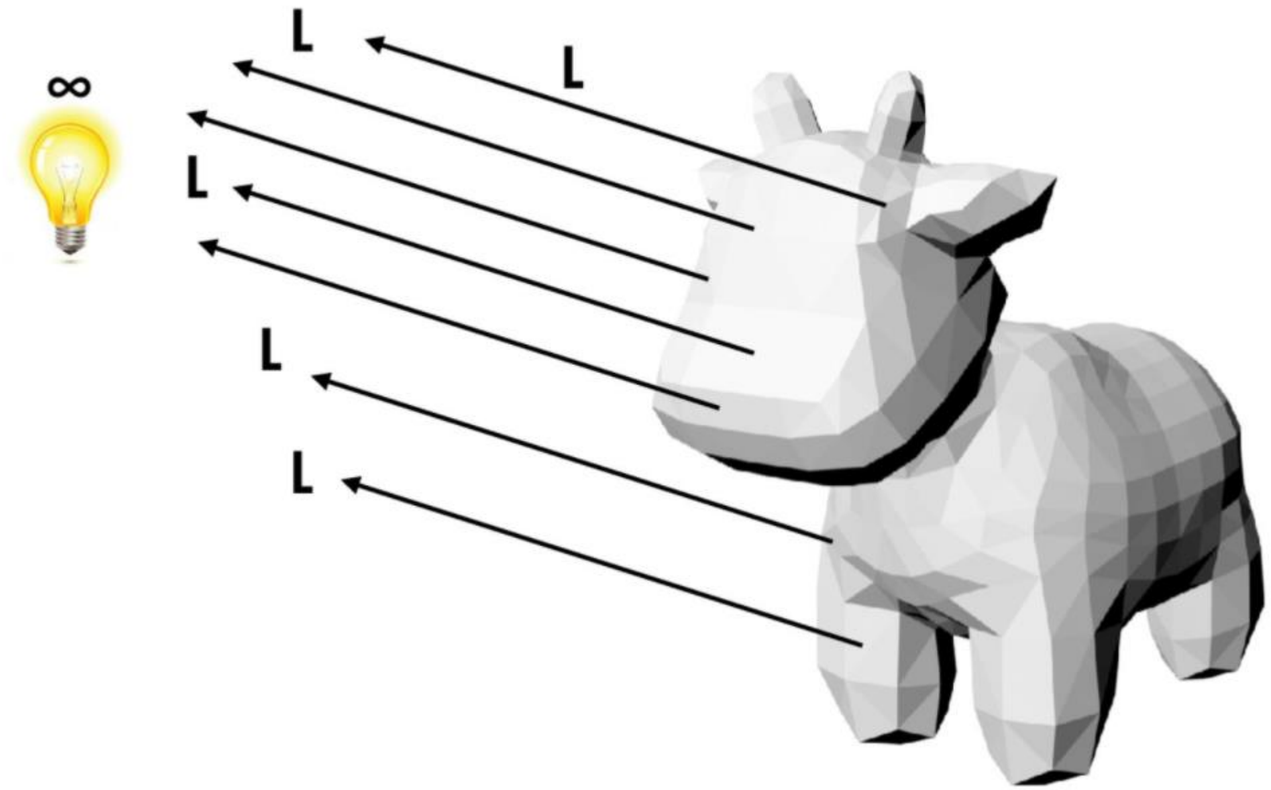
- Most basic way to shade a surface: take dot product of unit surface normal (N) and unit direction to light (L)

```
double surfaceColor( Vec3 N, Vec3 L )
{
    return max( 0., dot( N, L ));
}
```



Example: “directional” lighting

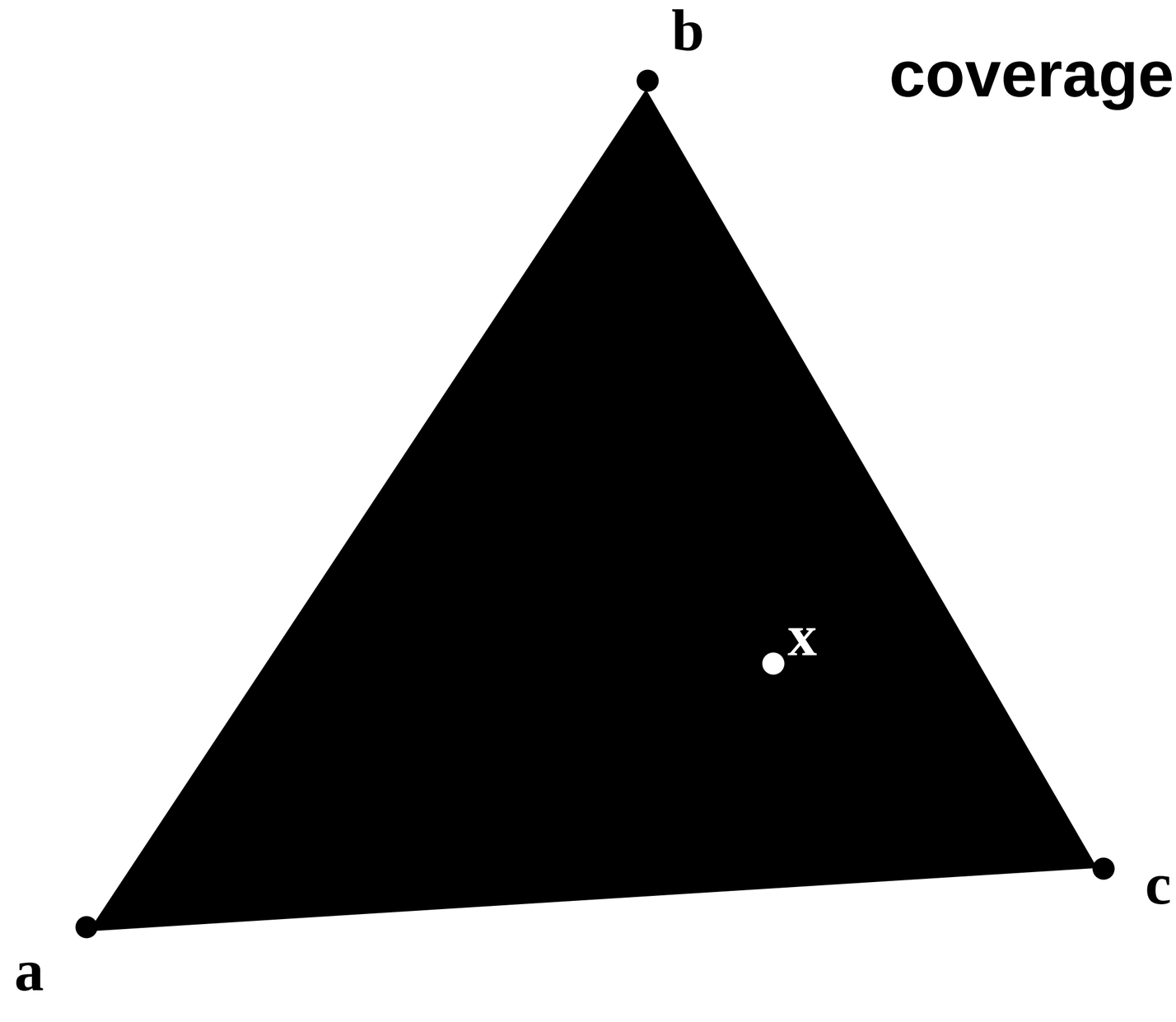
- Common abstraction: infinitely bright light source “at infinity”
- All light directions (L) are therefore identical



Back to triangles

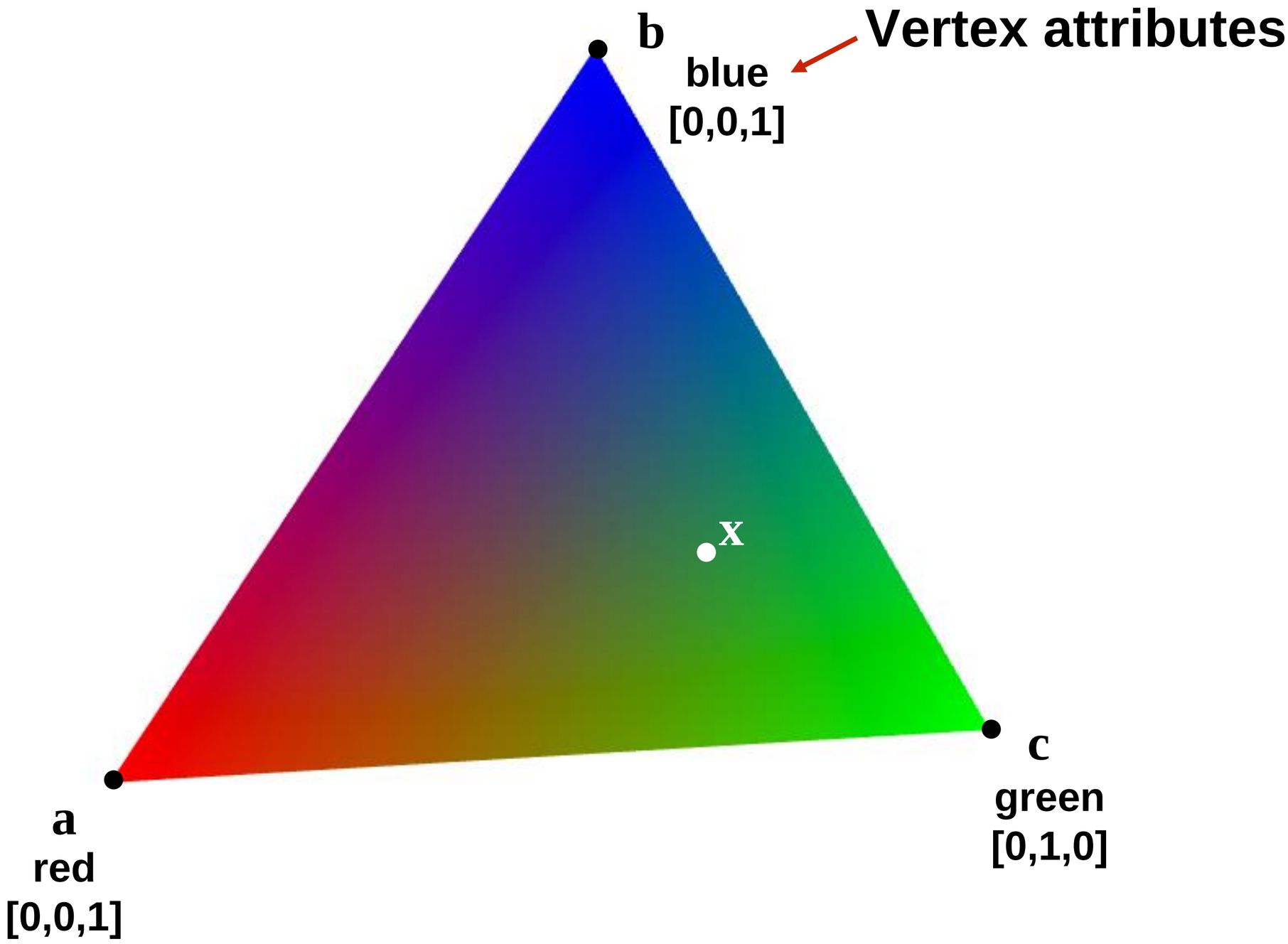
Coverage(x,y)

A few lectures ago we discussed how to sample coverage given the 2D position of the triangle's vertices.



$$\text{coverage}(x,y) = \begin{cases} 1 & \text{if the triangle contains point } (x,y) \\ 0 & \text{otherwise} \end{cases}$$

Consider sampling color(x,y)



What is the triangle's color at point x?

Review: interpolation in 1D

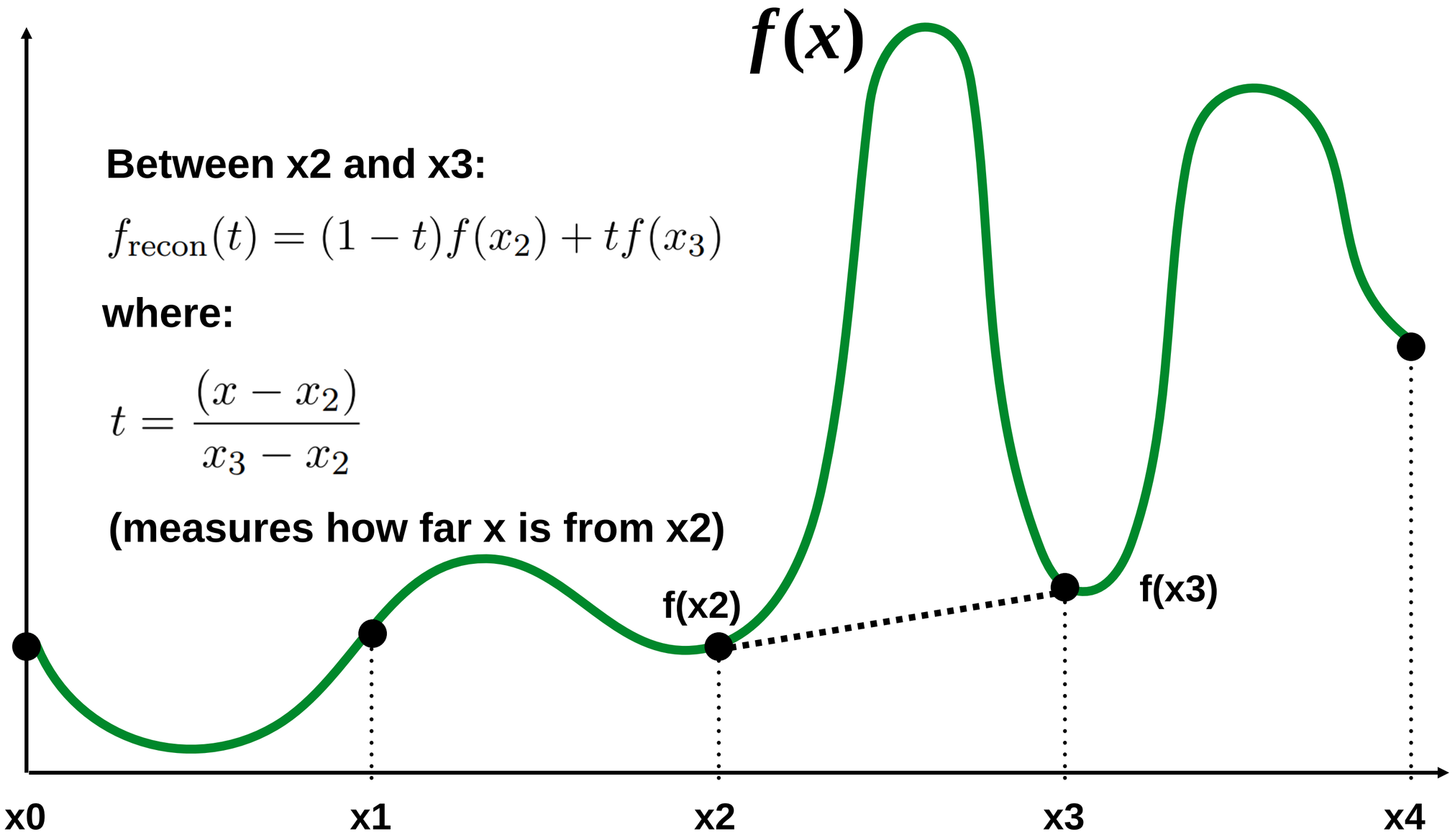
Between x2 and x3:

$$f_{\text{recon}}(t) = (1-t)f(x_2) + tf(x_3)$$

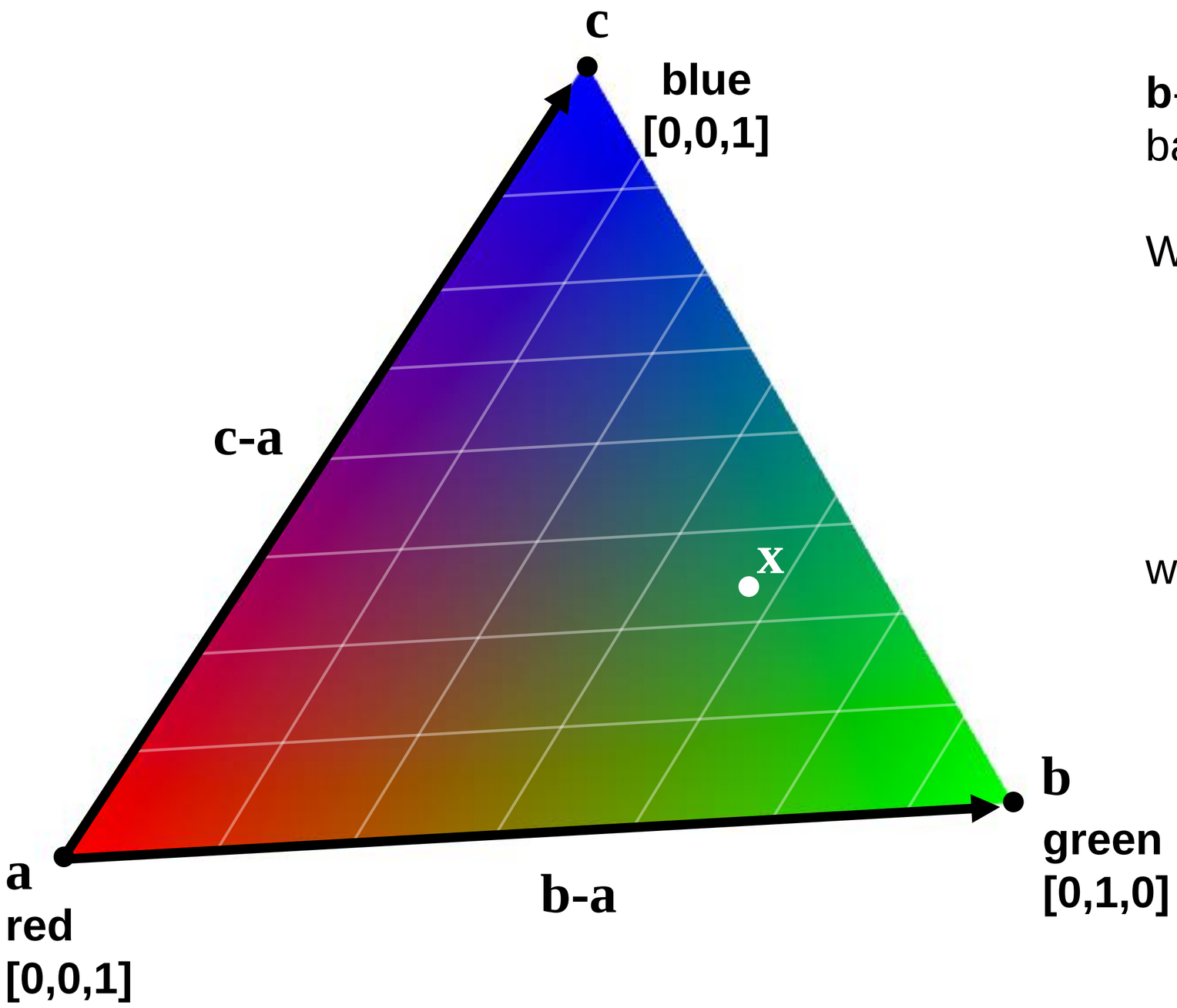
where:

$$t = \frac{(x - x_2)}{x_3 - x_2}$$

(measures how far x is from x2)



Interpolation via barycentric coordinates



b-**a** and **c**-**a** form a non-orthogonal basis for points in triangle.

We can therefore write:

$$\begin{aligned}\mathbf{x} &= \mathbf{a} + \beta(\mathbf{b}-\mathbf{a}) + \gamma(\mathbf{c}-\mathbf{a}) \\ &= (1-\beta-\gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c} \\ &= \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}\end{aligned}$$

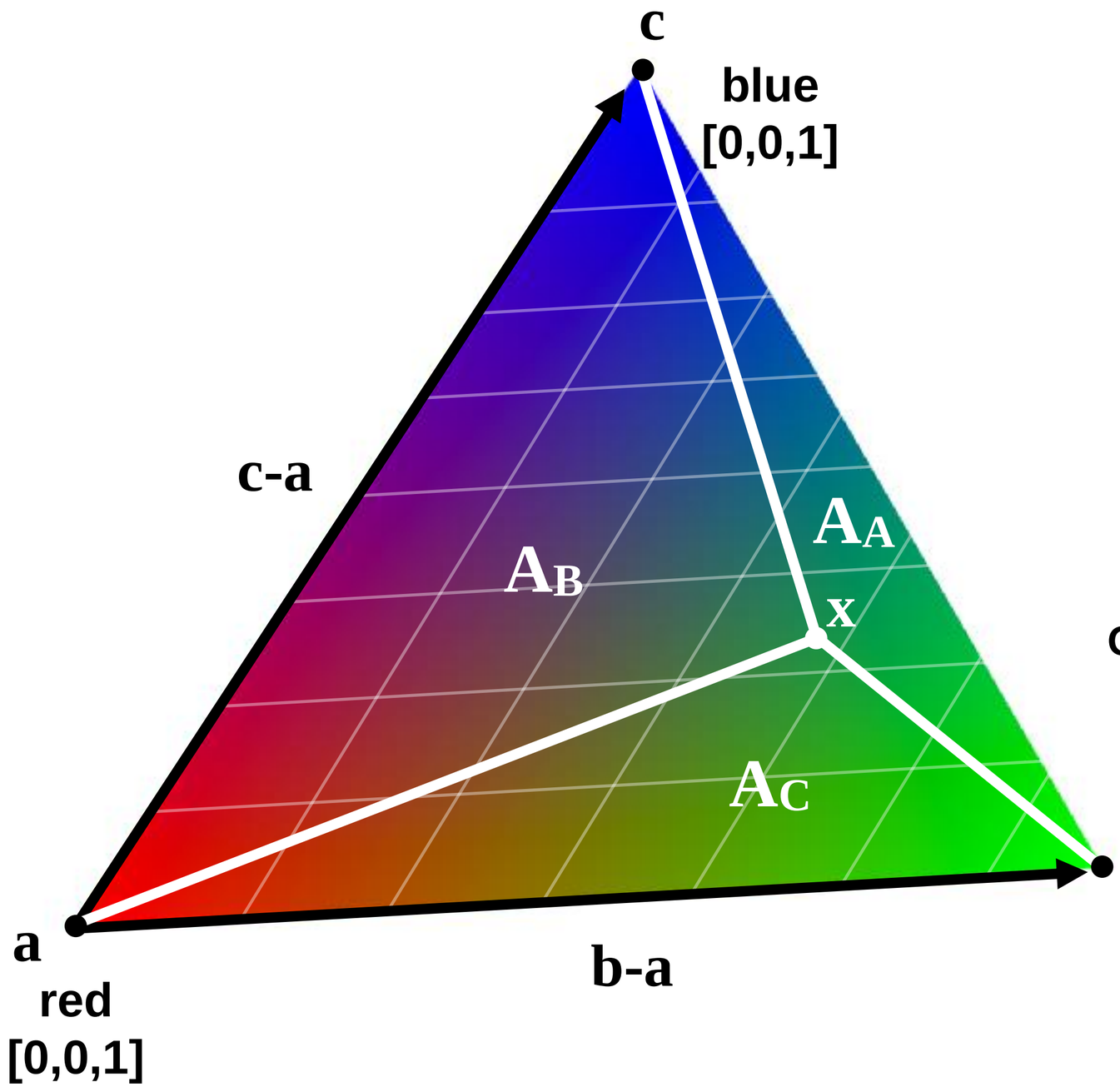
with

$$\alpha + \beta + \gamma = 1$$

Color at **x** is a combination of colors stored at triangle vertices.

$$\mathbf{x}_{\text{color}} = \alpha\mathbf{a}_{\text{color}} + \beta\mathbf{b}_{\text{color}} + \gamma\,\mathbf{c}_{\text{color}}$$

Barycentric coordinates as ratio of areas



$$\alpha = A_A/A$$

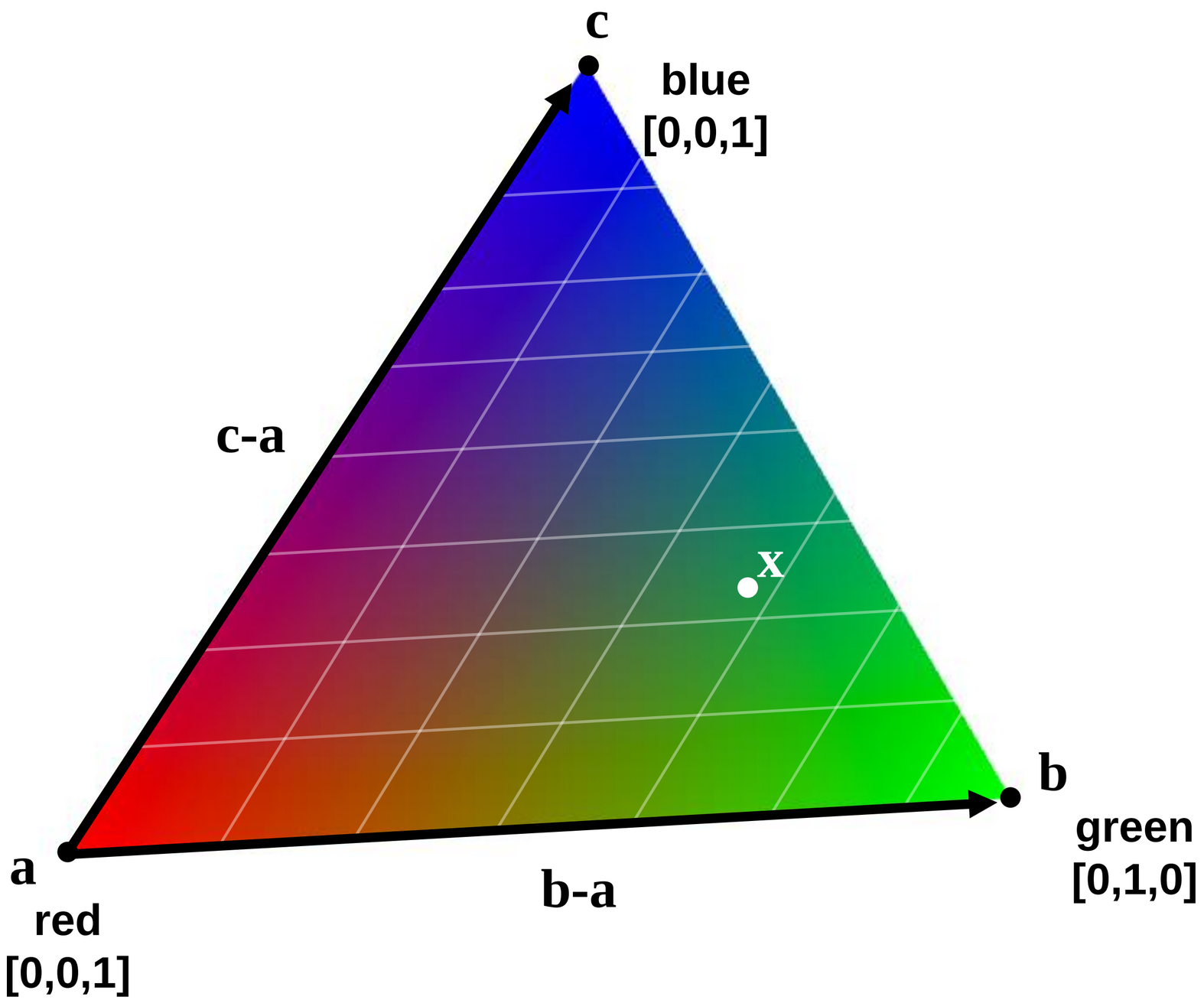
$$\beta = A_B/A$$

$$\gamma = A_C/A$$

Q: Why must coordinates sum to 1?
Why must coordinates be between 0 and 1?

In the context of a triangle, Barycentric coordinates are also called areal coordinates

But what are we interpolating again?

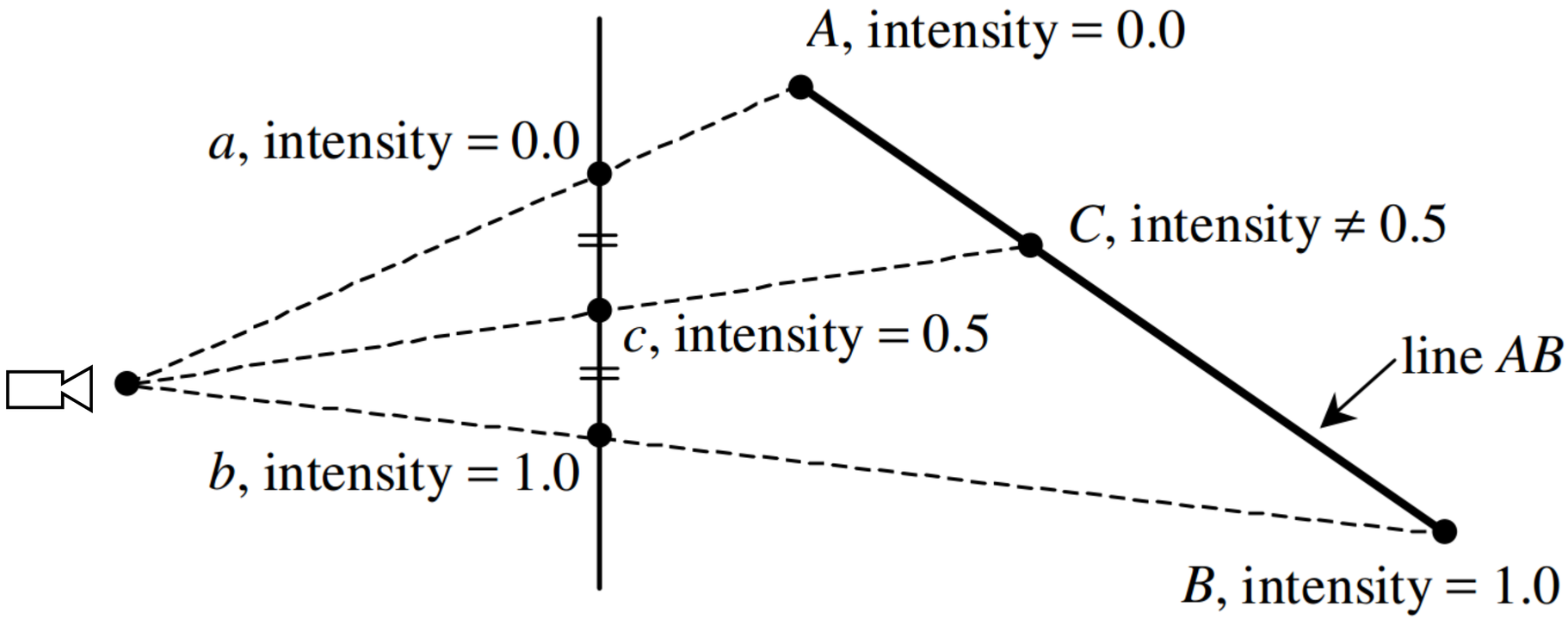


What are **a**, **b** and **c**?

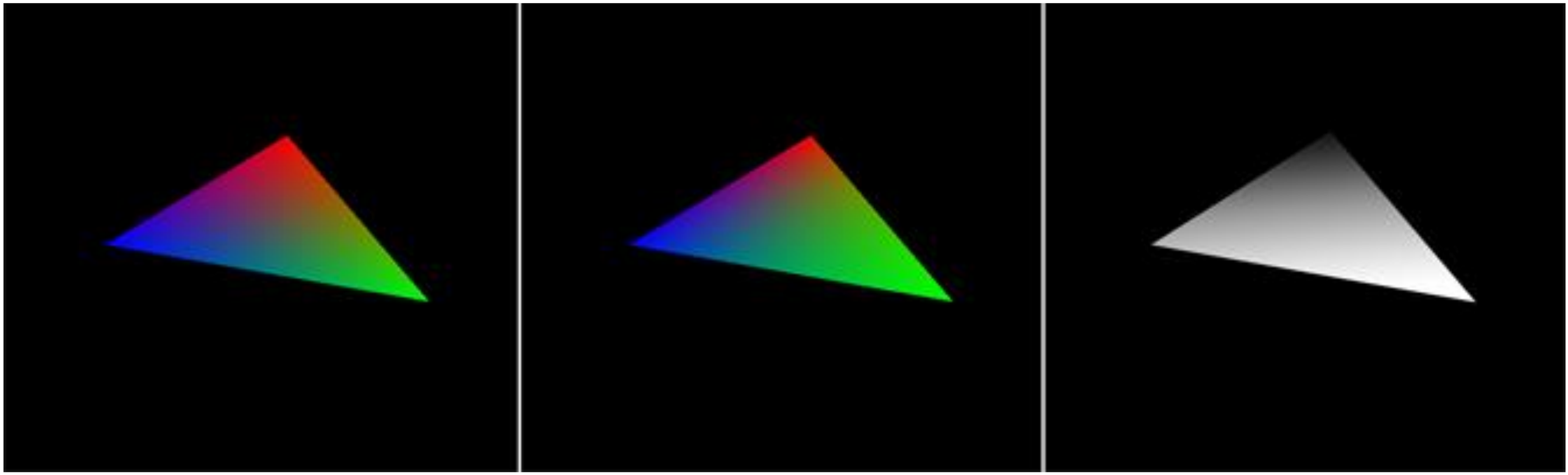
Perspective-incorrect interpolation

Due to perspective projection (homogenous divide), barycentric interpolation of values in screen XY coordinates does not correspond to values that vary linearly on original triangle!

Attribute values must be interpolated linearly in 3D object space.



An example: perspective-incorrect interpolation



Perspective-correct interpolation

Attribute values (f) vary linearly across triangle in 3D. Due to perspective projection, f/z varies linearly in screen coordinates, and not f directly.

Basic recipe:

- **Compute depth z at each vertex**
- **Evaluate Z:=$1/z$, P:= f/z at each vertex**
- **Interpolate Z and P using standard (2D) barycentric coordinates**
- **At each *fragment*, divide interpolated P by interpolated Z to get f**

Works for any surface attribute f that varies linearly across triangle: e.g., color, depth, texture coordinates

For complete derivation, see Low, “Perspective-correct Interpolation”

What do we still need to create a scene like this?

Texture mapping

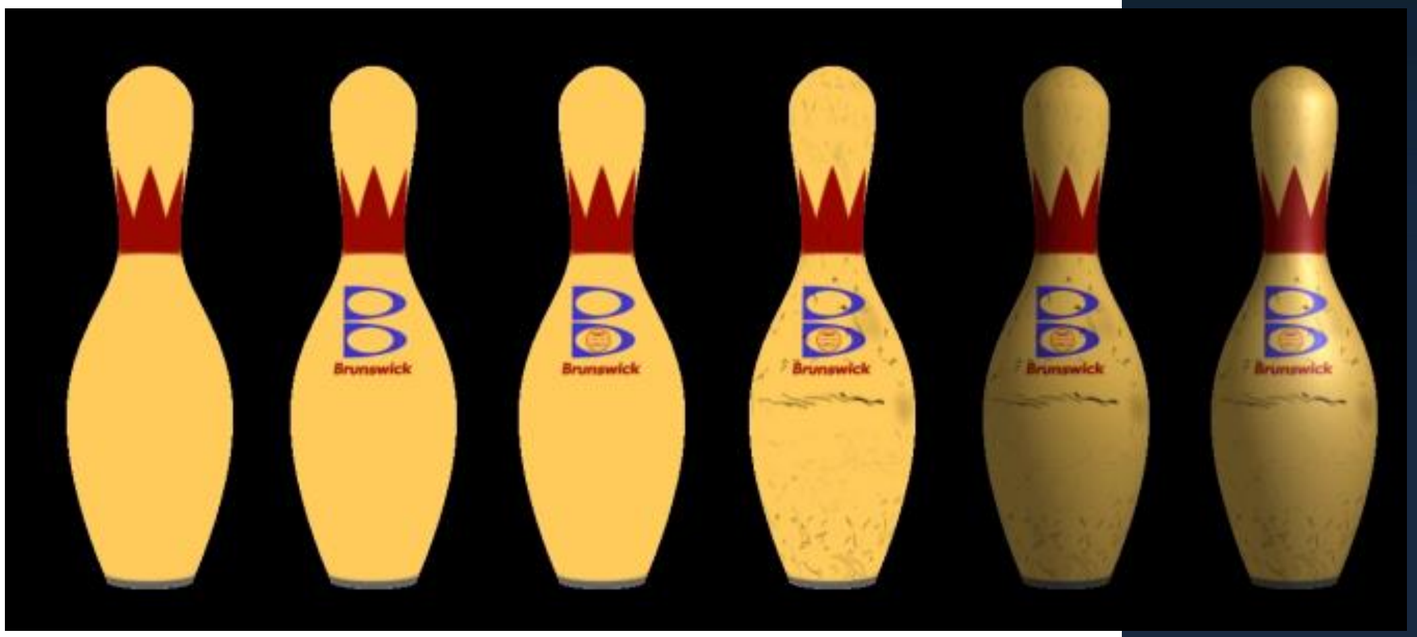
Many uses of texture mapping

Define variation in surface reflectance

Pattern on ball

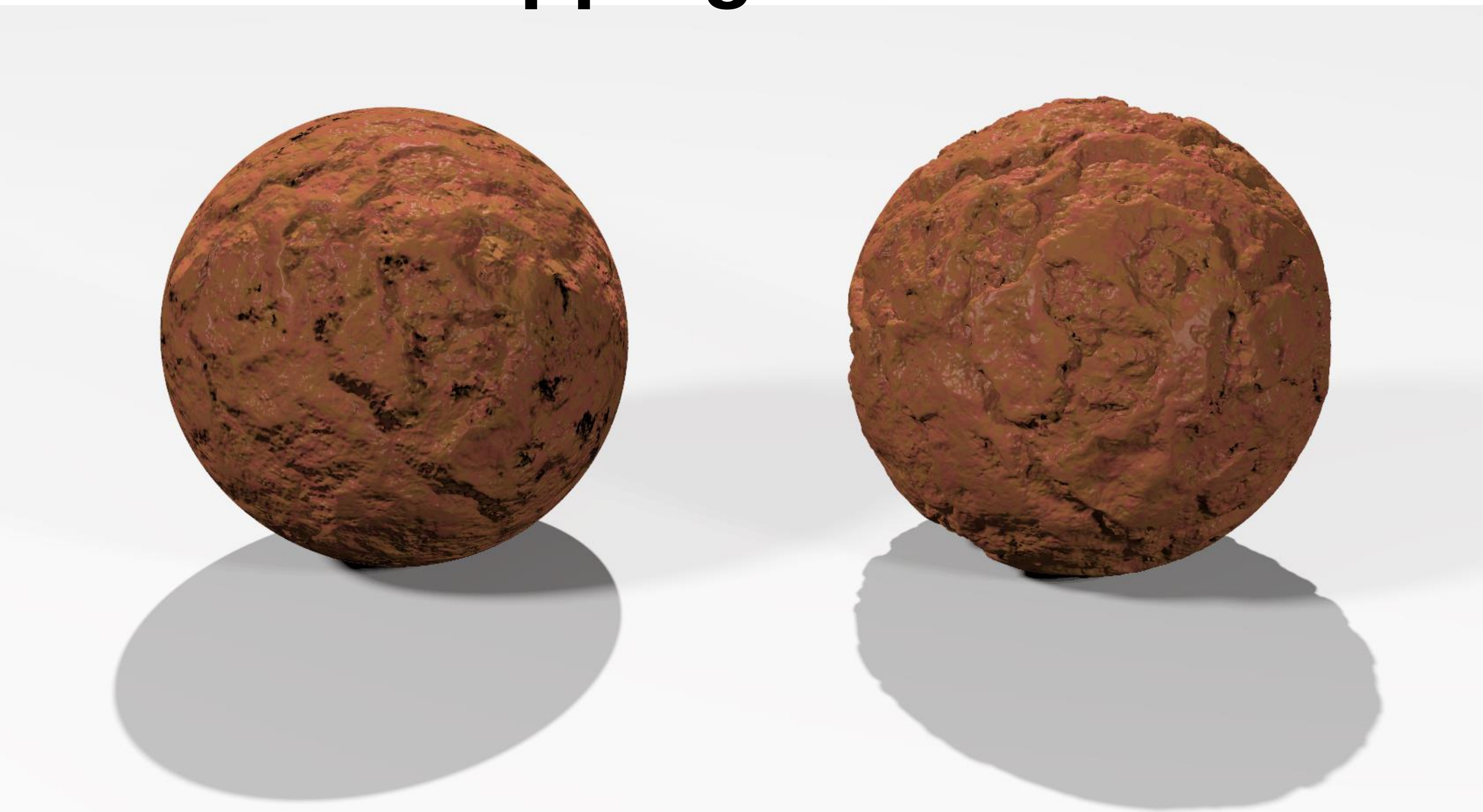
Wood grain on floor

Describe surface material properties



Multiple layers of texture maps for color, logos, scratches, etc.

Normal mapping



Use texture value to perturb surface normal to give appearance of a bumpy surface
Observe: smooth silhouette and smooth shadow boundary indicates surface geometry is not bumpy

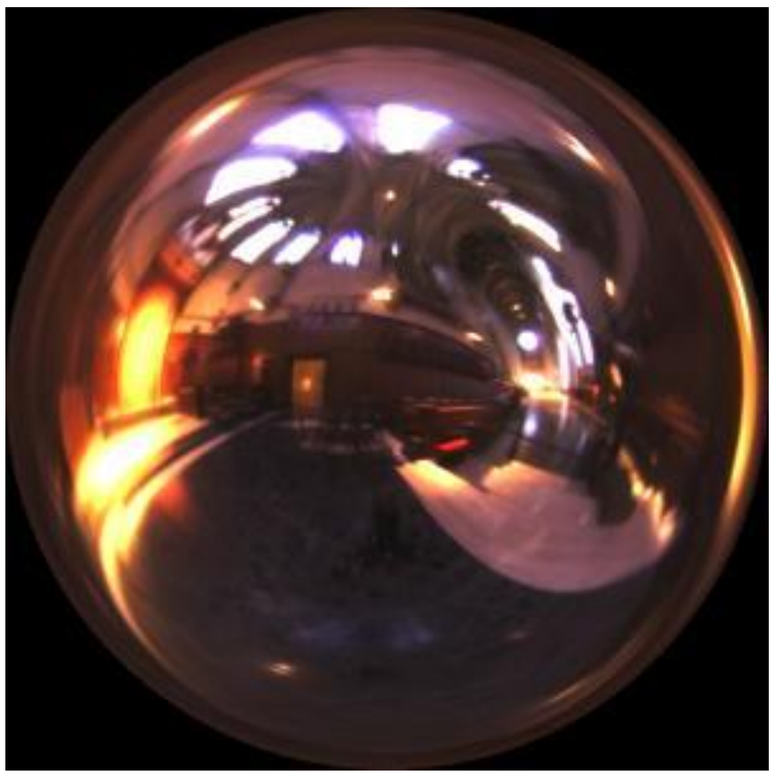
Rendering using high-resolution surface geometry (note bumpy silhouette and shadow boundary)

Represent precomputed lighting and shadows

Original model

With ambient occlusion

Extracted ambient occlusion map

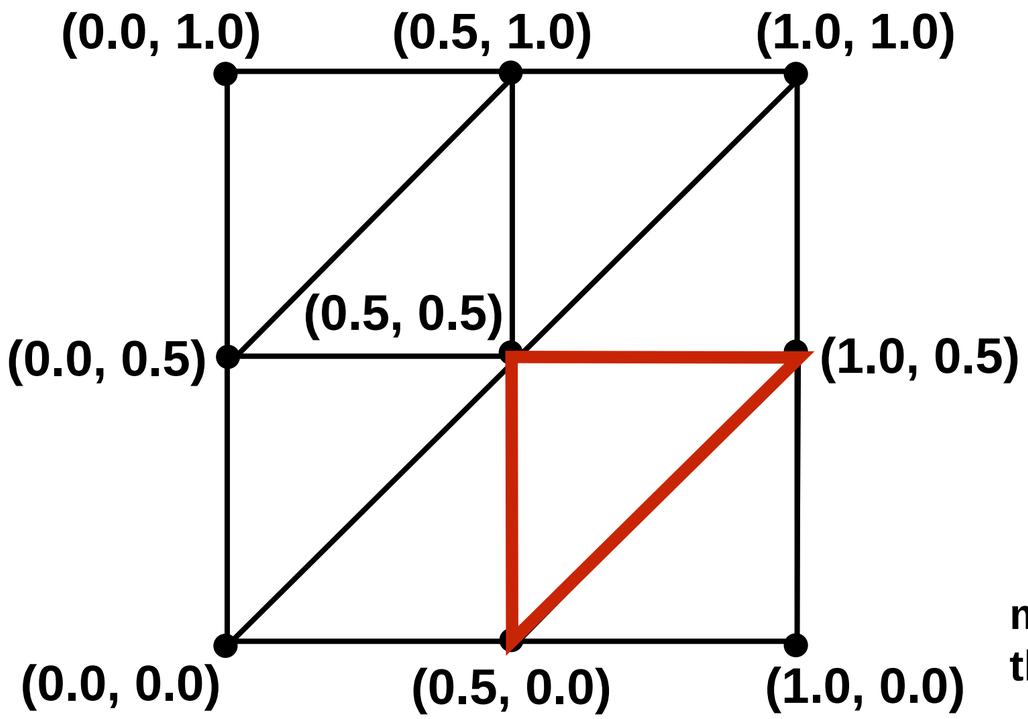


Grace Cathedral environment map

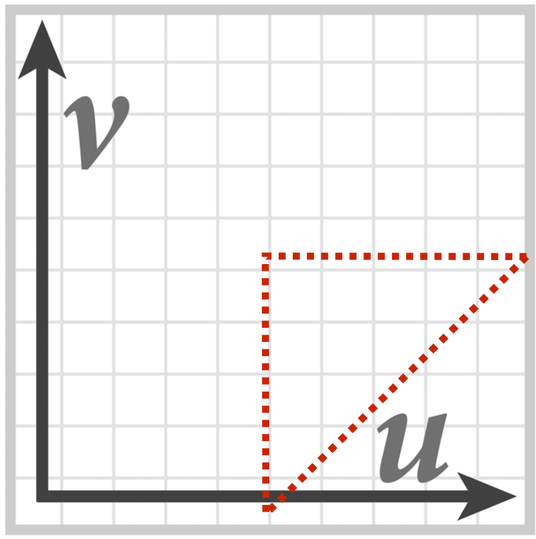
Environment map used in rendering

Texture coordinates

“Texture coordinates” define a mapping from surface coordinates (points on triangle) to points in texture domain.



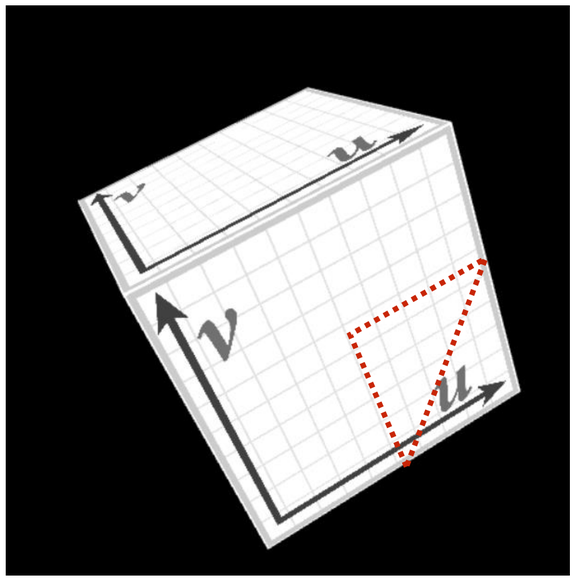
Eight triangles (one face of cube) with surface parameterization provided as per-vertex texture coordinates.



myTex(u,v) is a function defined on the $[0,1]^2$ domain:

myTex : $[0,1]^2 \rightarrow$ float3
(represented by 2048x2048 image)

Location of highlighted triangle in texture space shown in red.



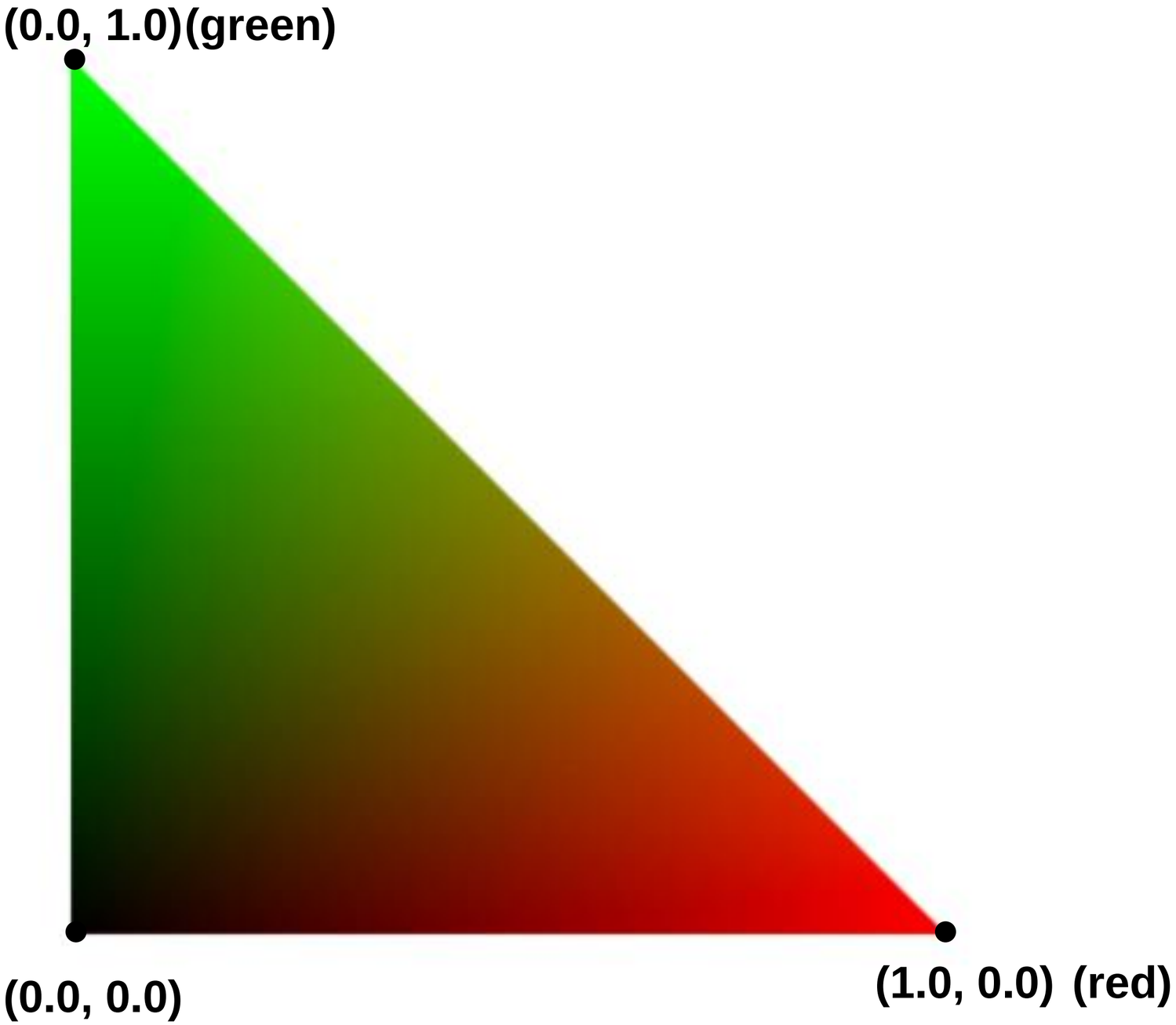
Final rendered result (entire cube shown).

Location of triangle after projection onto screen shown in red.

Surface-to-texture space mapping is provided as per vertex attributes.

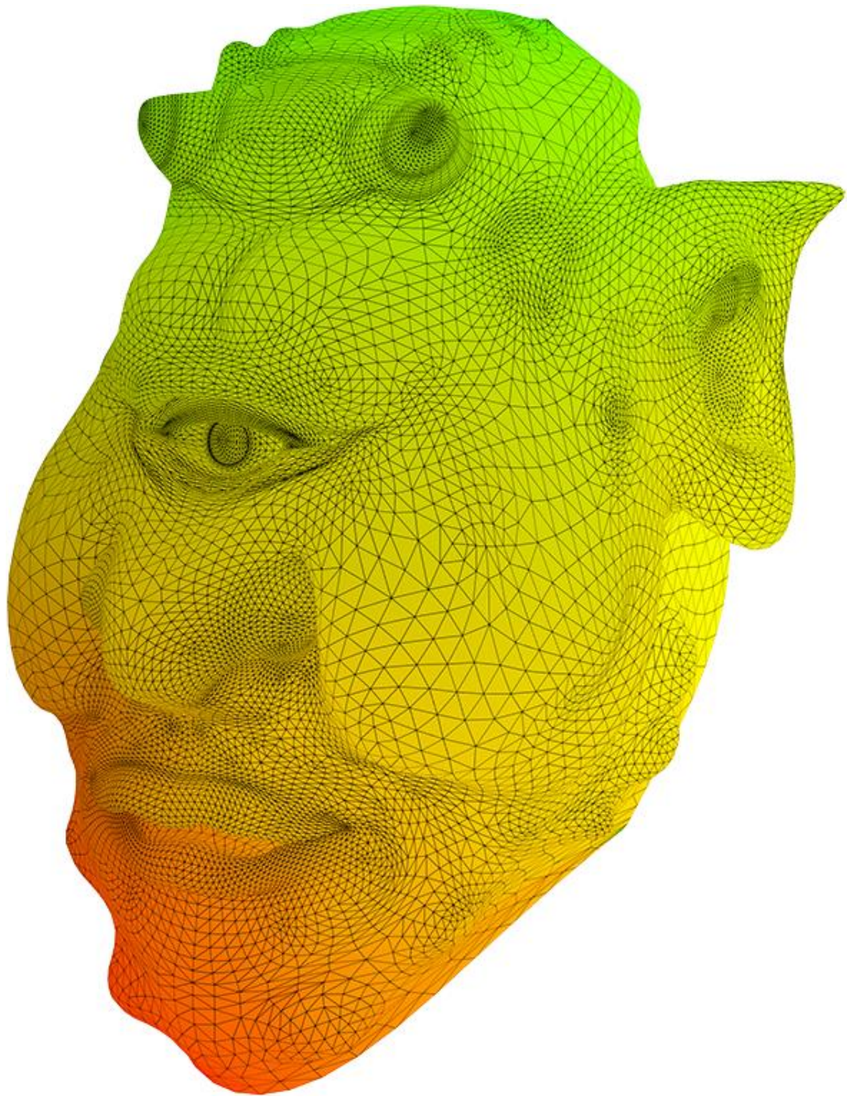
Visualization of texture coordinates

Texture coordinates linearly interpolated over triangle

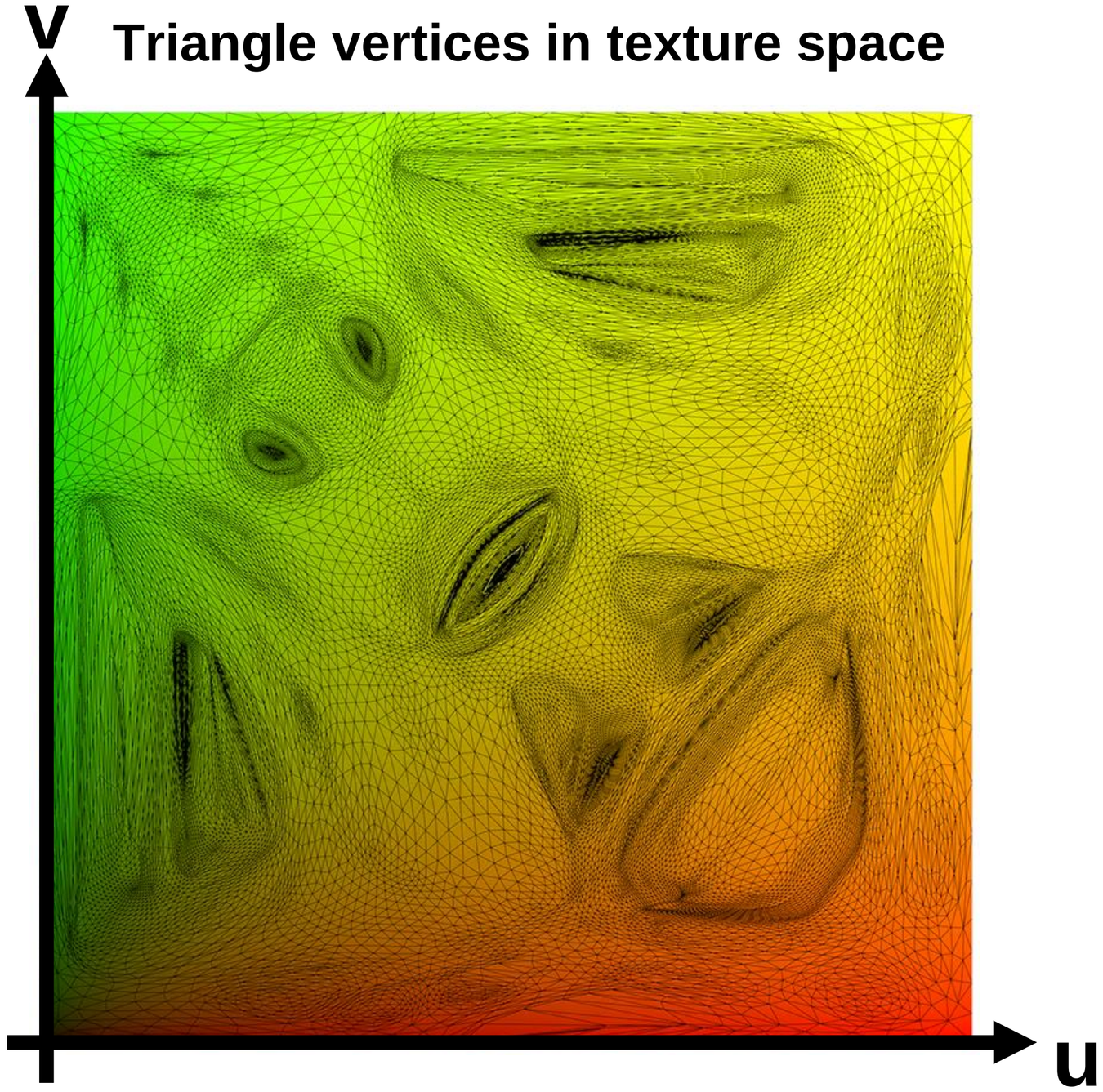


More complex mapping

Visualization of texture coordinates



Triangle vertices in texture space



Each vertex has a coordinate (u,v) in texture space.
(Actually coming up with these coordinates is another story!)

Simple texture mapping operation

for each covered screen sample (x,y):

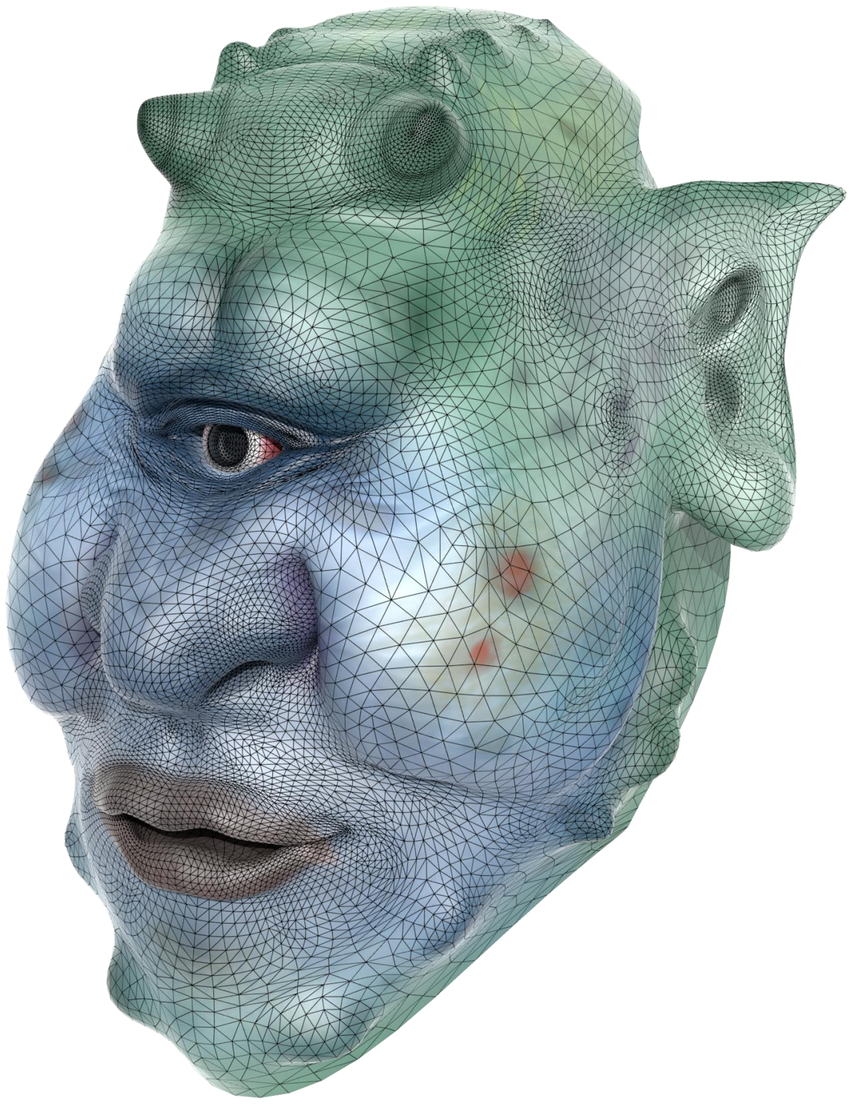
(u,v) = evaluate texcoord value at (x,y)

float3 texcolor = texture.sample(u,v); ← “just” an image lookup…

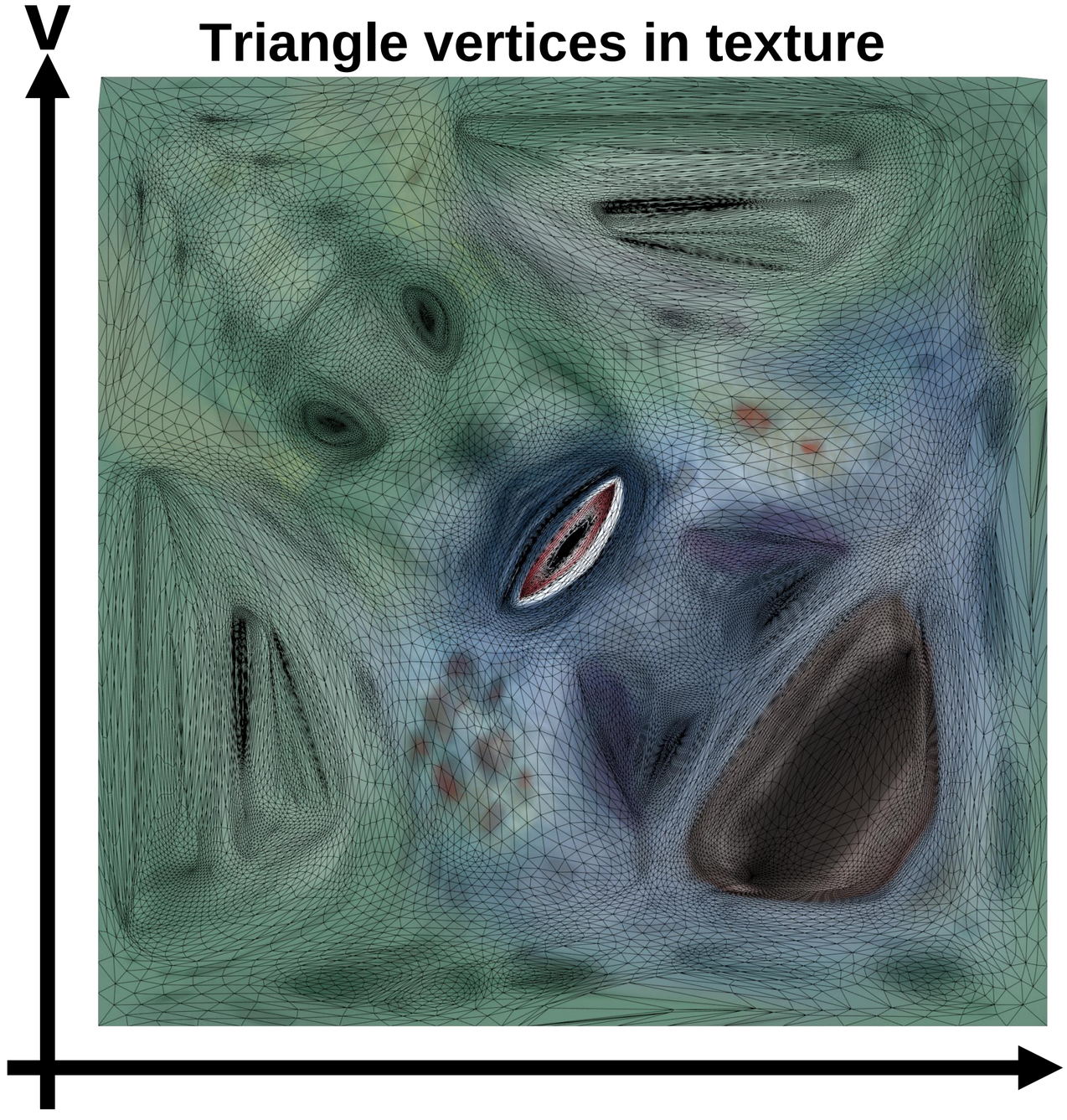
set sample’s color to texcolor;

Texture mapping adds detail

Rendered result



Triangle vertices in texture

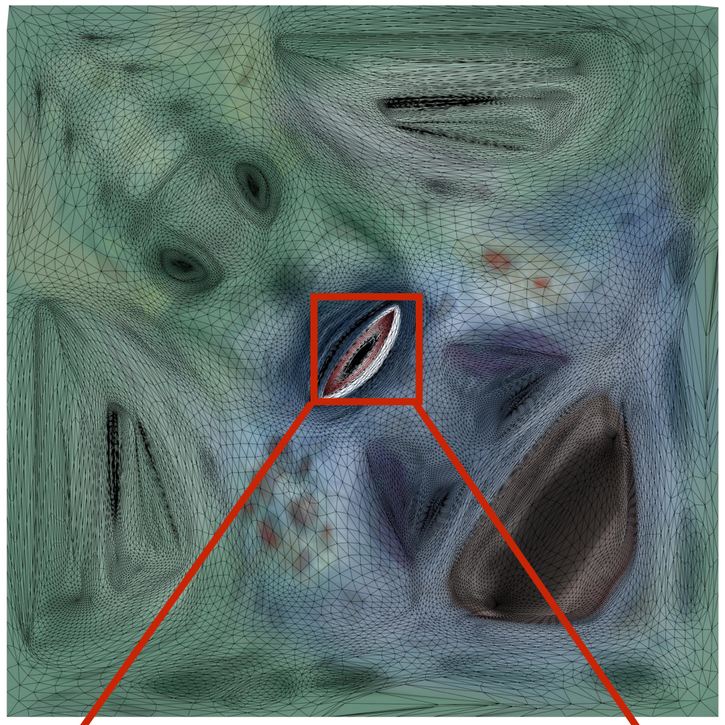


Texture mapping adds detail

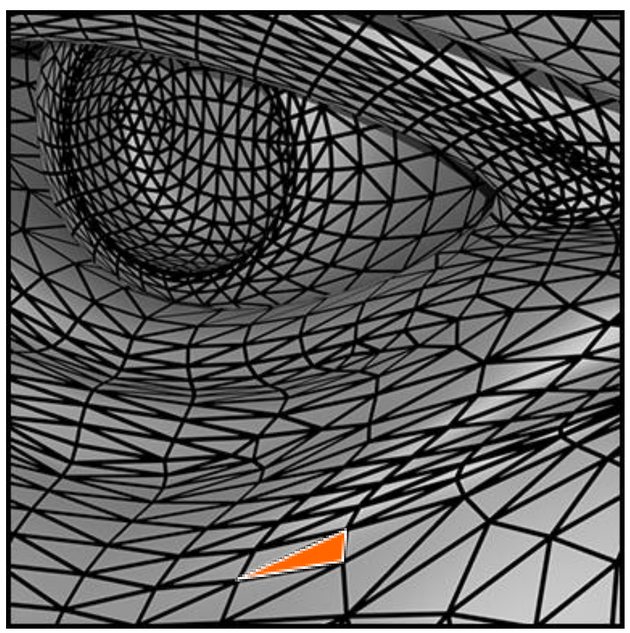
rendering without

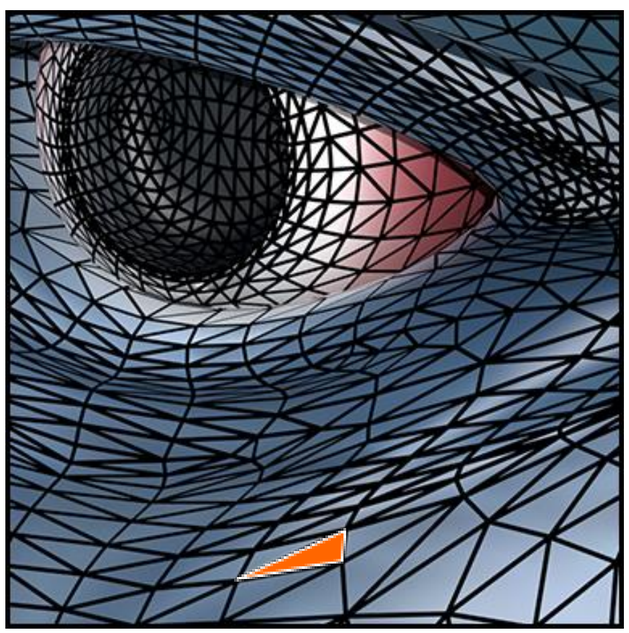
rendering with texture

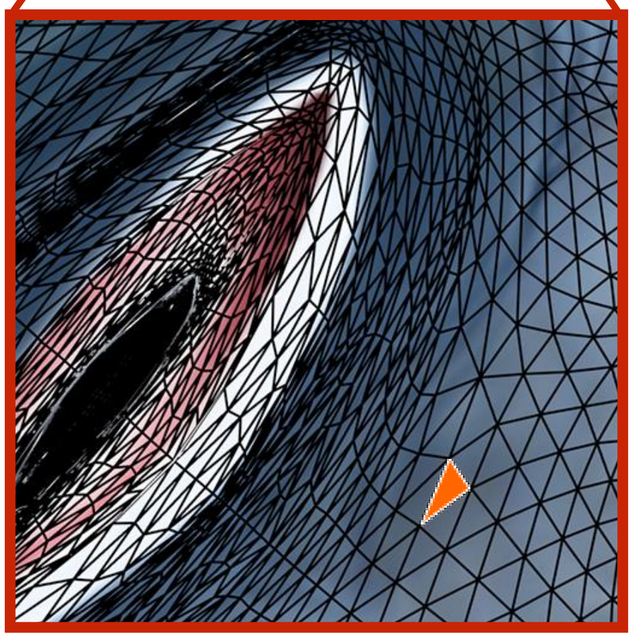
texture image



zoom

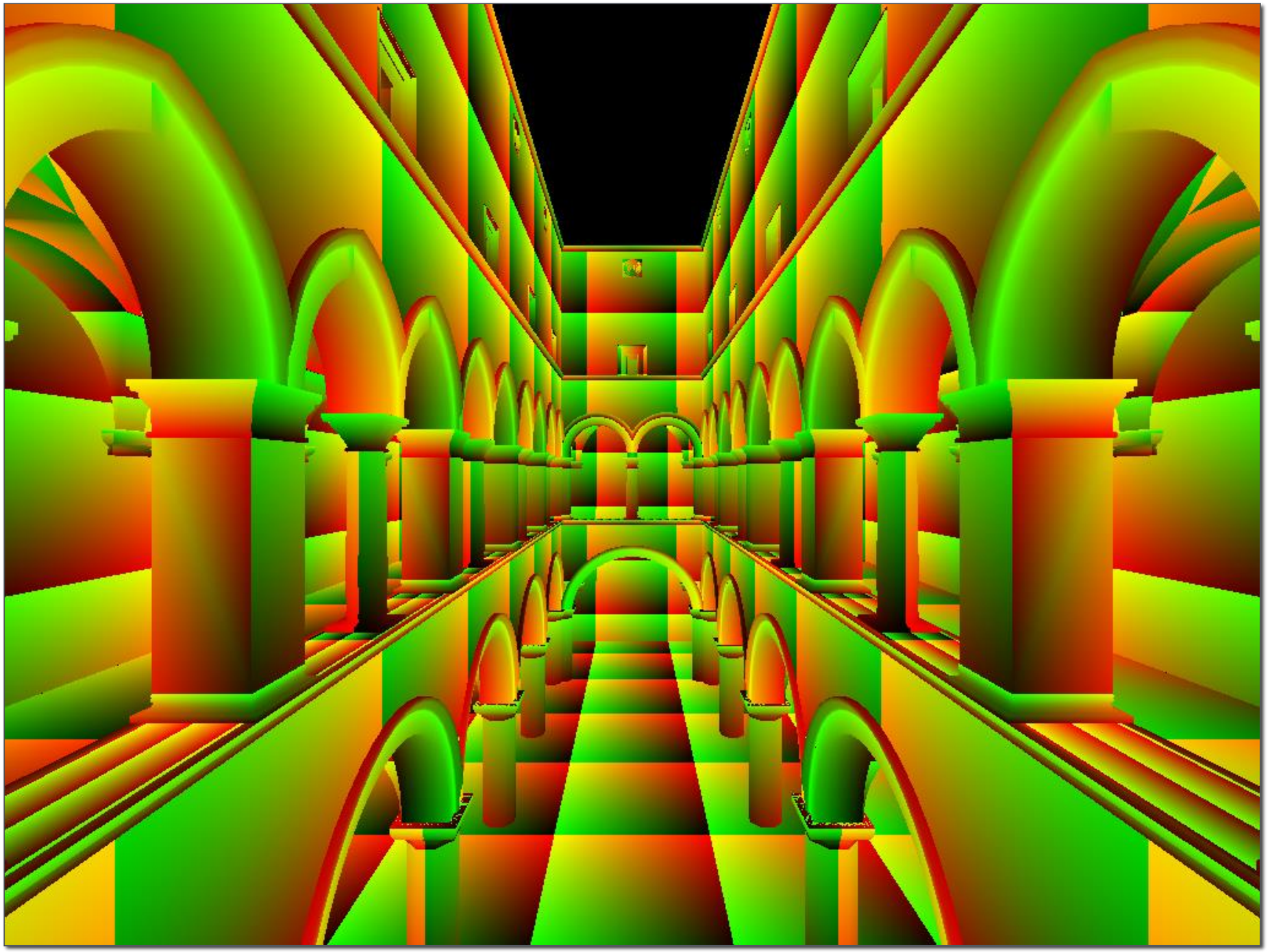






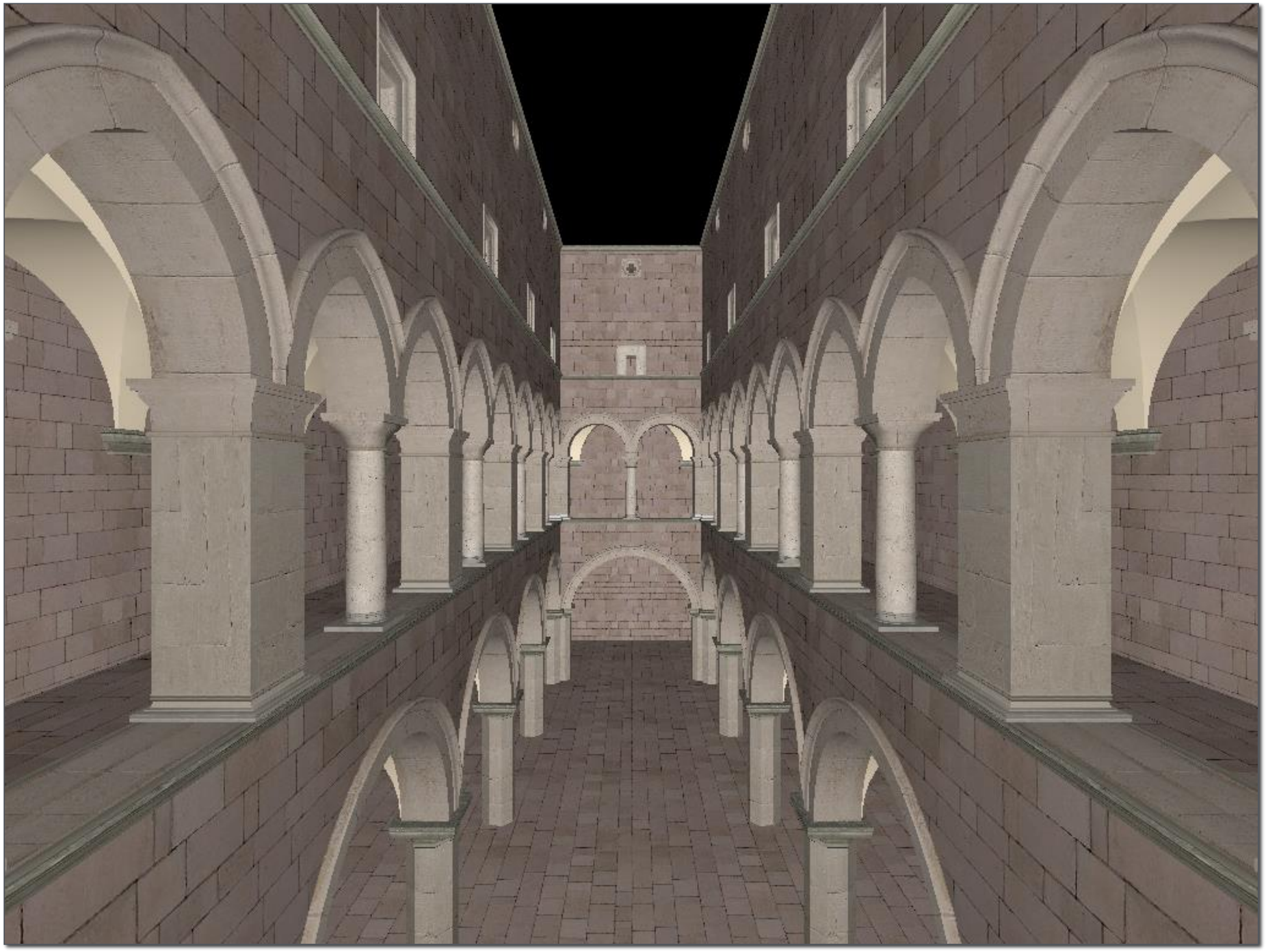
Each triangle “copies” a piece of the image back to the surface.

Another example: Sponza

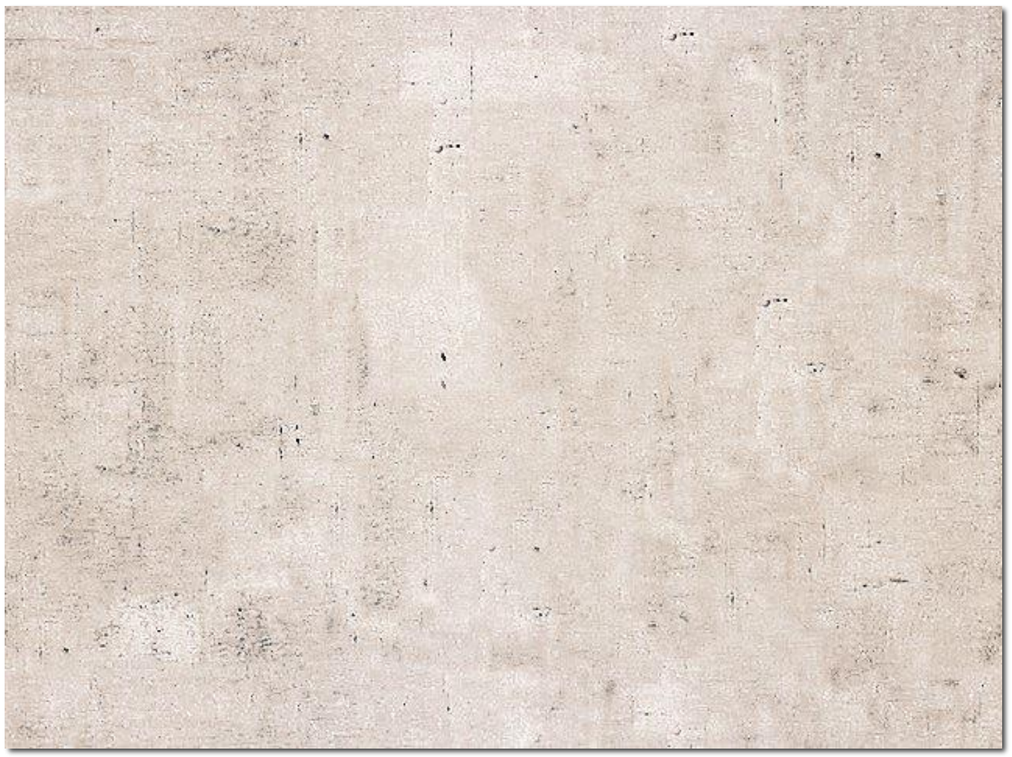


Notice texture coordinates repeat over surface.

Textured Sponza

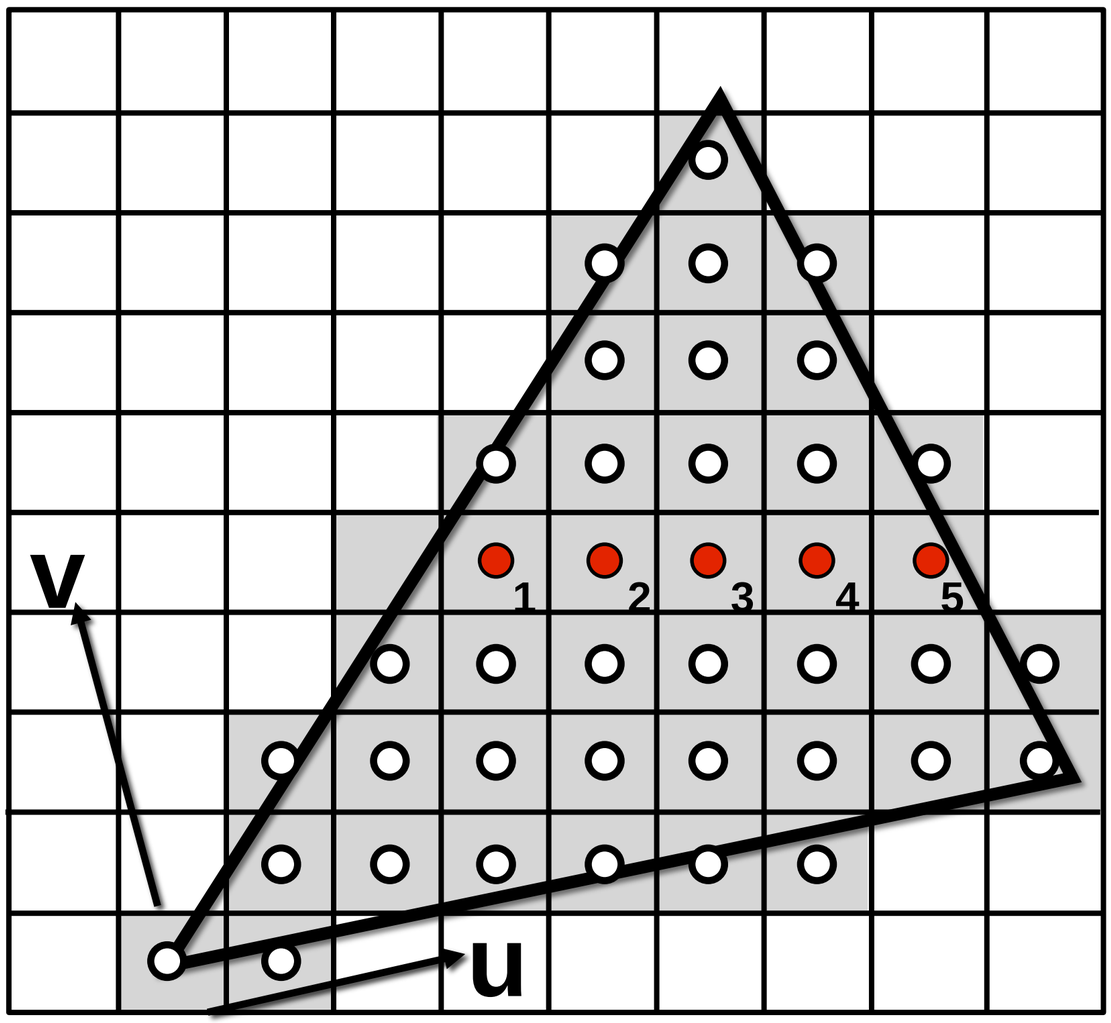


Example textures used in Sponza



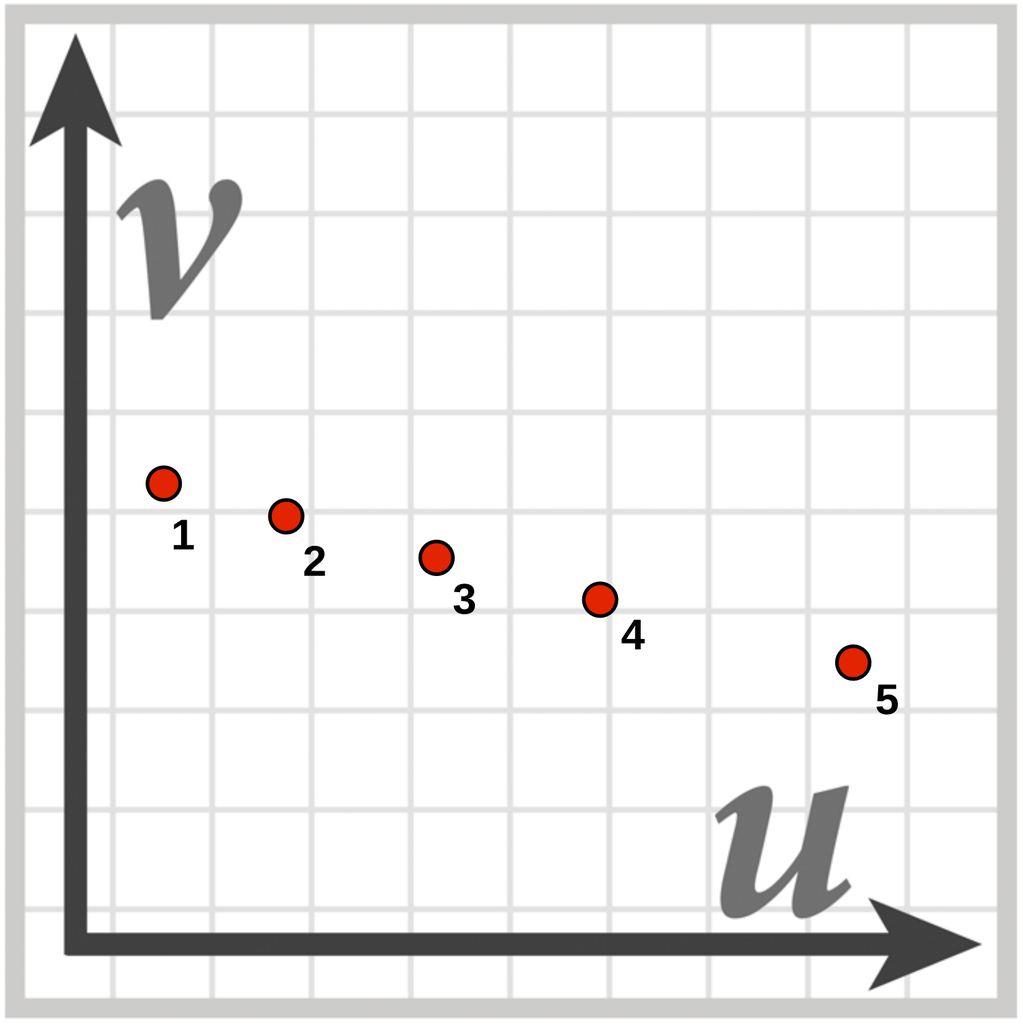
Texture space samples

Sample positions in XY screen space



Sample positions are uniformly distributed in screen space (rasterizer samples triangle's appearance at these locations)

Sample positions in texture space

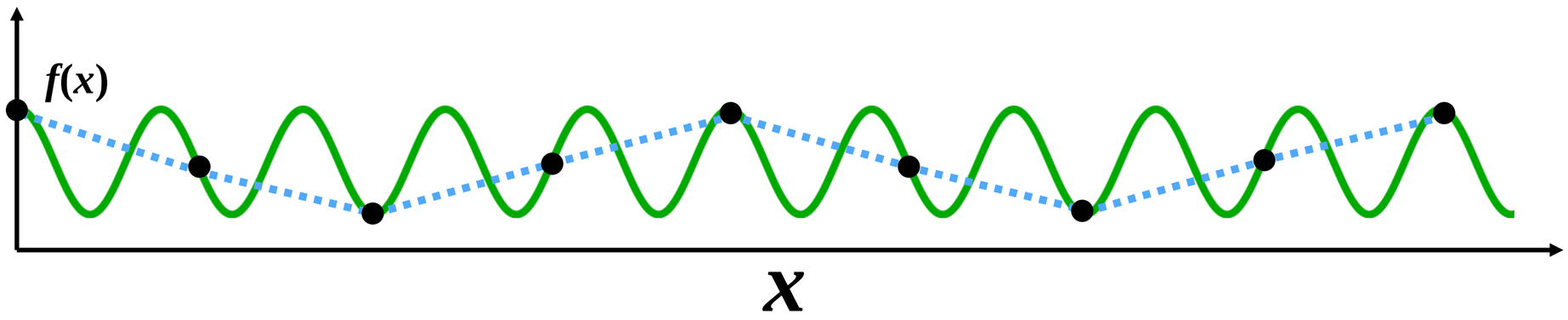


Texture sample positions in texture space (texture function is sampled at these locations)

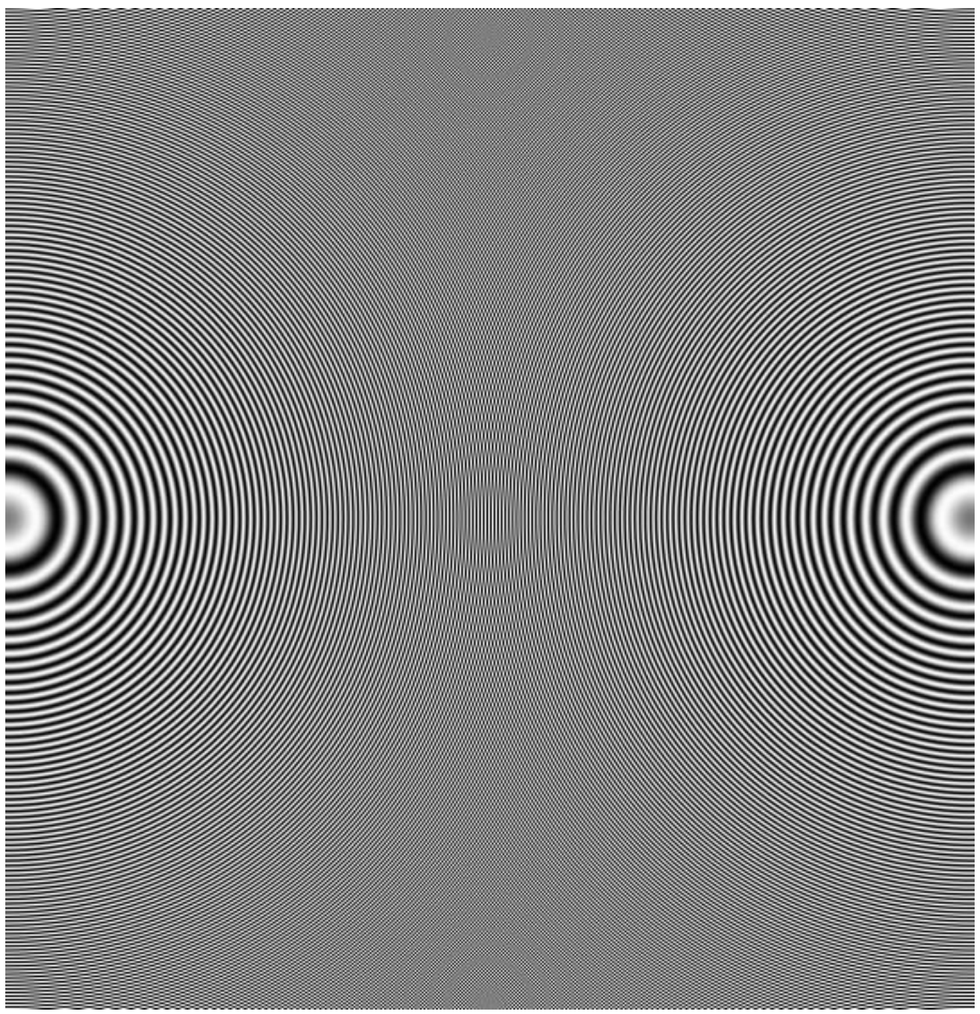
Q: what does it mean that equally-spaced points in screen space move further apart in texture space?

Recall: aliasing

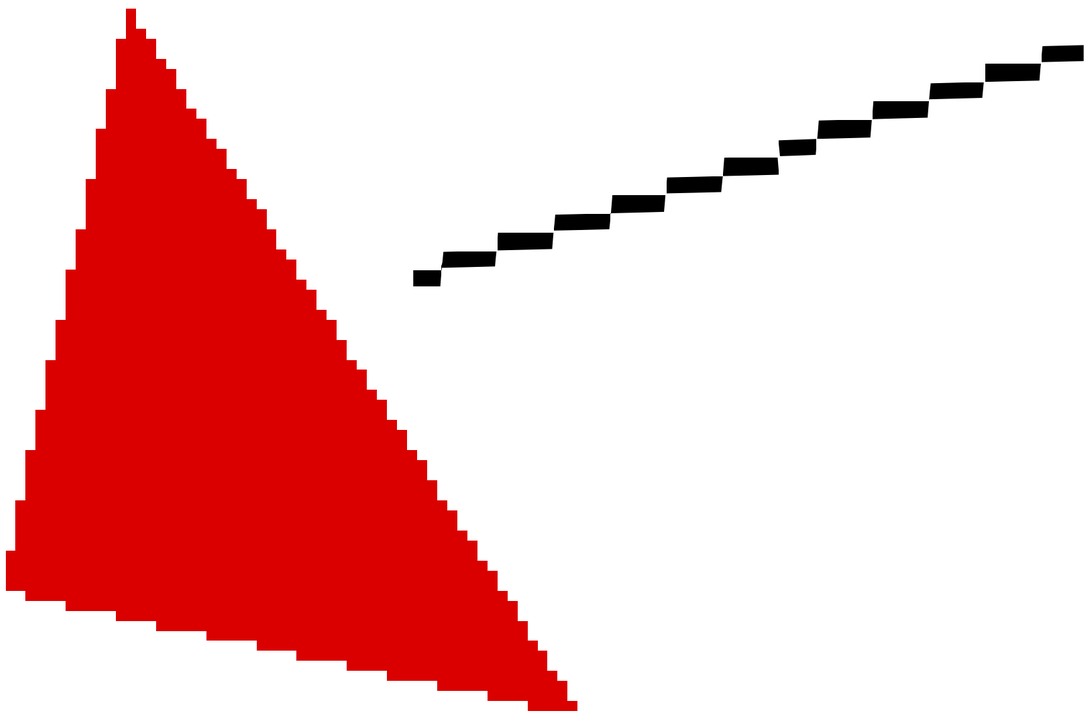
Undersampling a high-frequency signal can result in aliasing



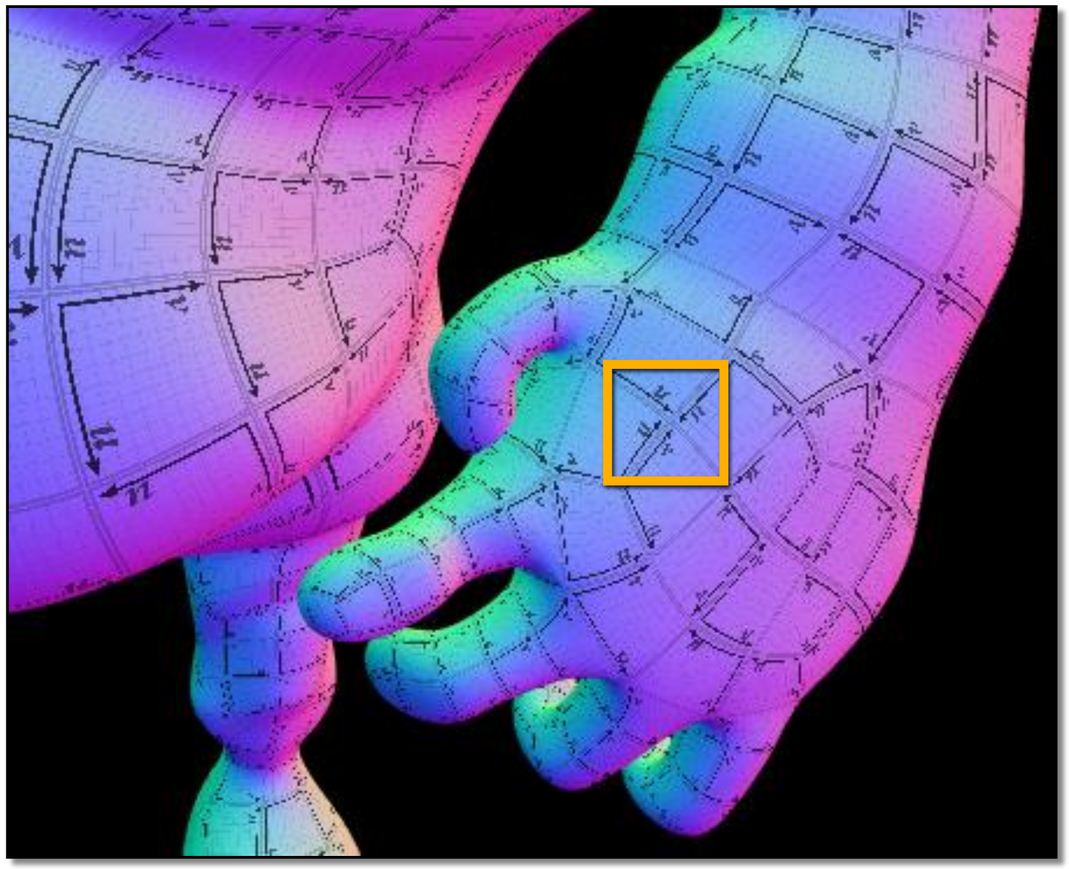
1D example



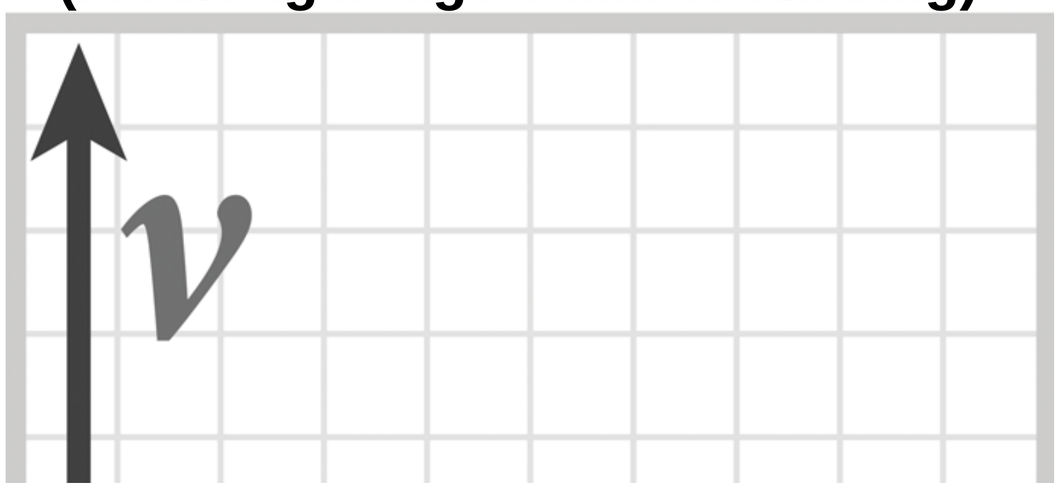
2D examples:
Moiré patterns,

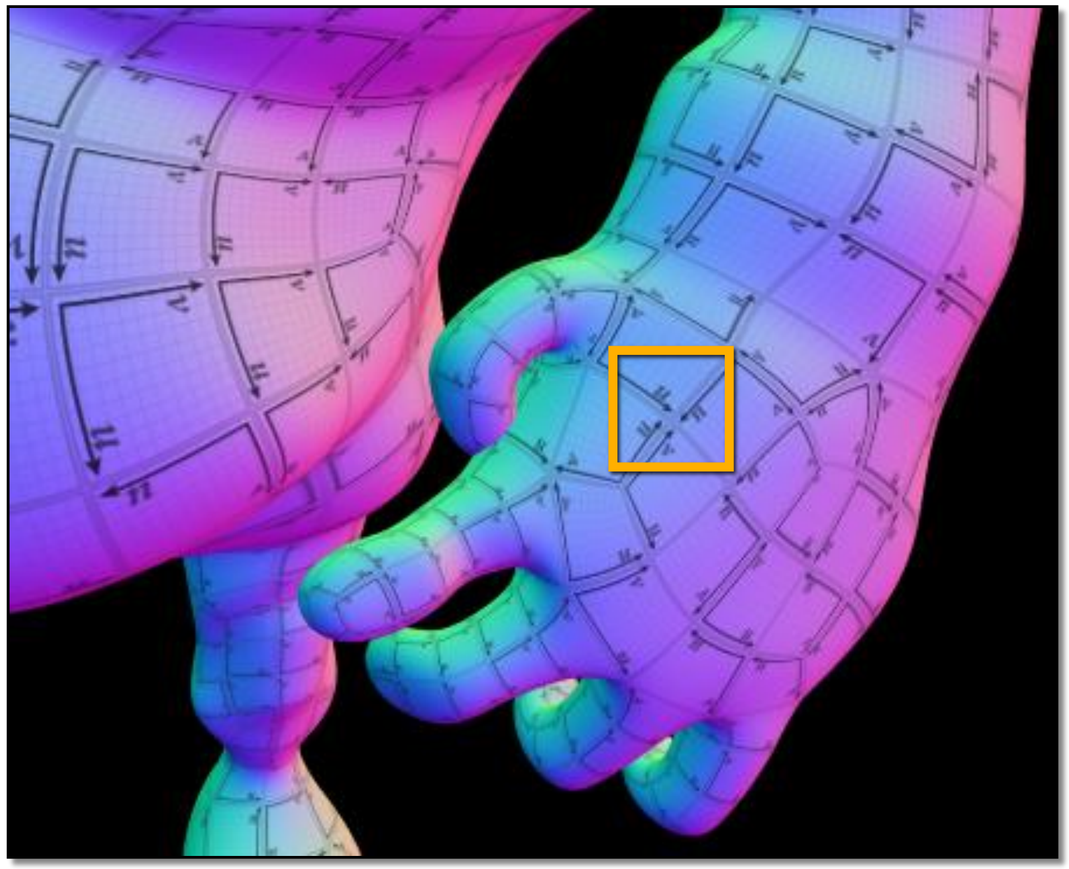


Aliasing due to undersampling texture



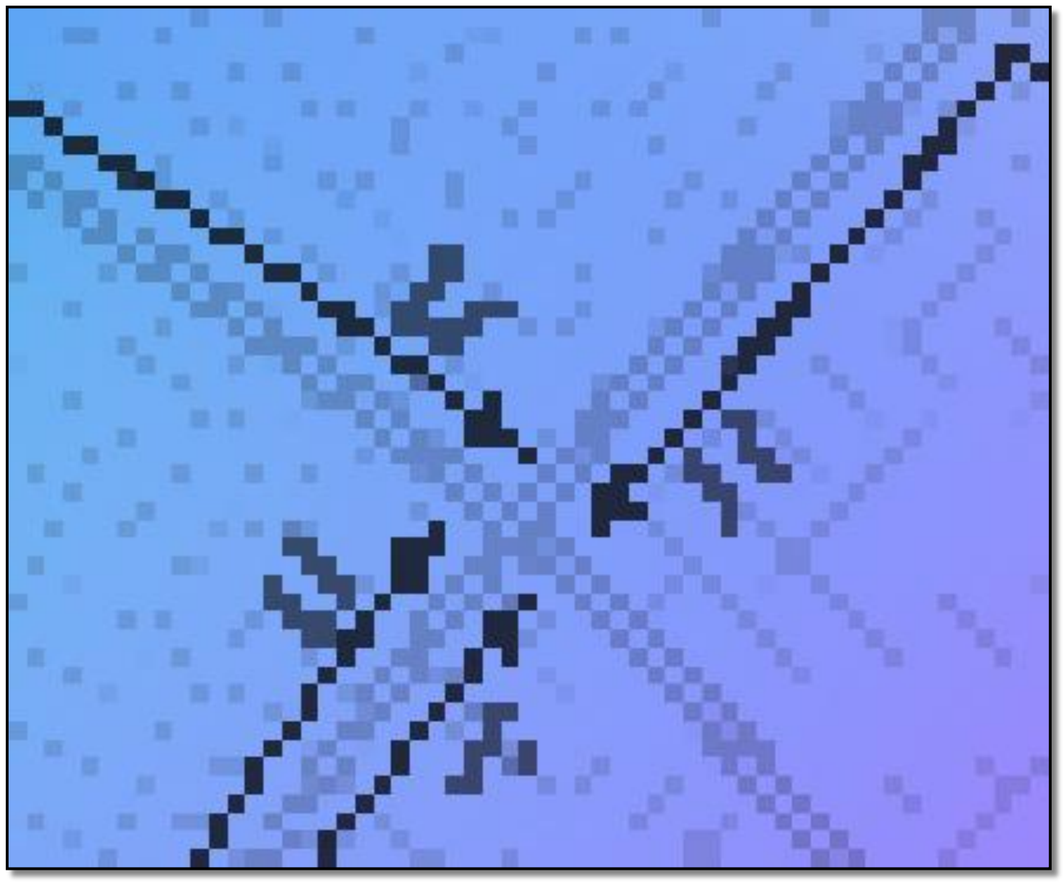
No pre-filtering of texture data
(resulting image exhibits aliasing)



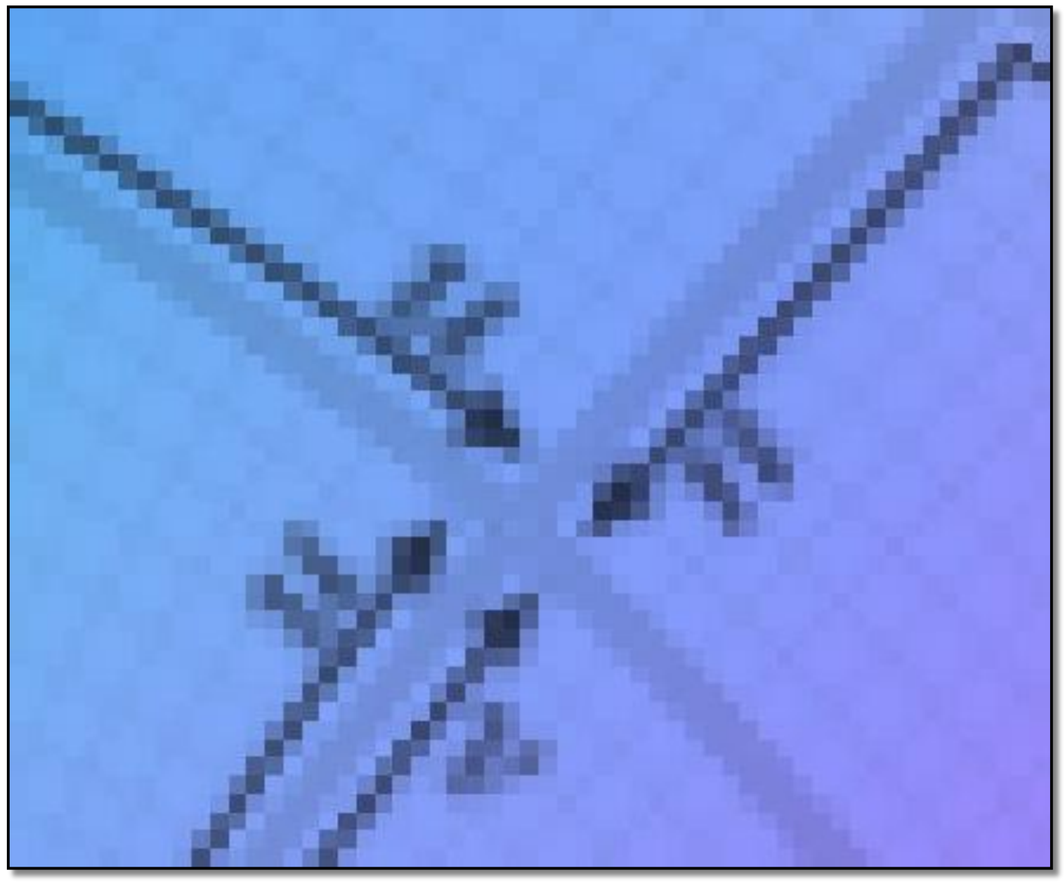


Rendering using pre-filtered
texture data

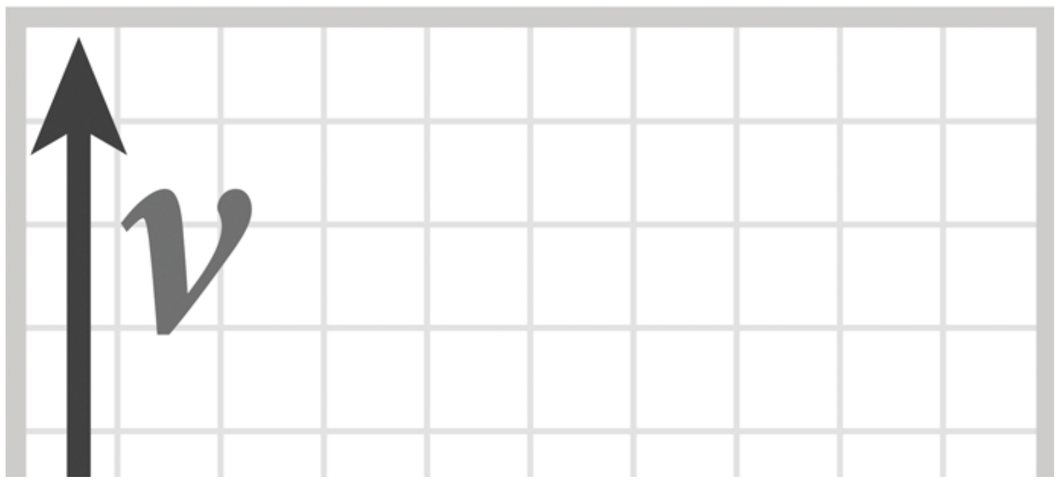
Aliasing due to undersampling (zoom)



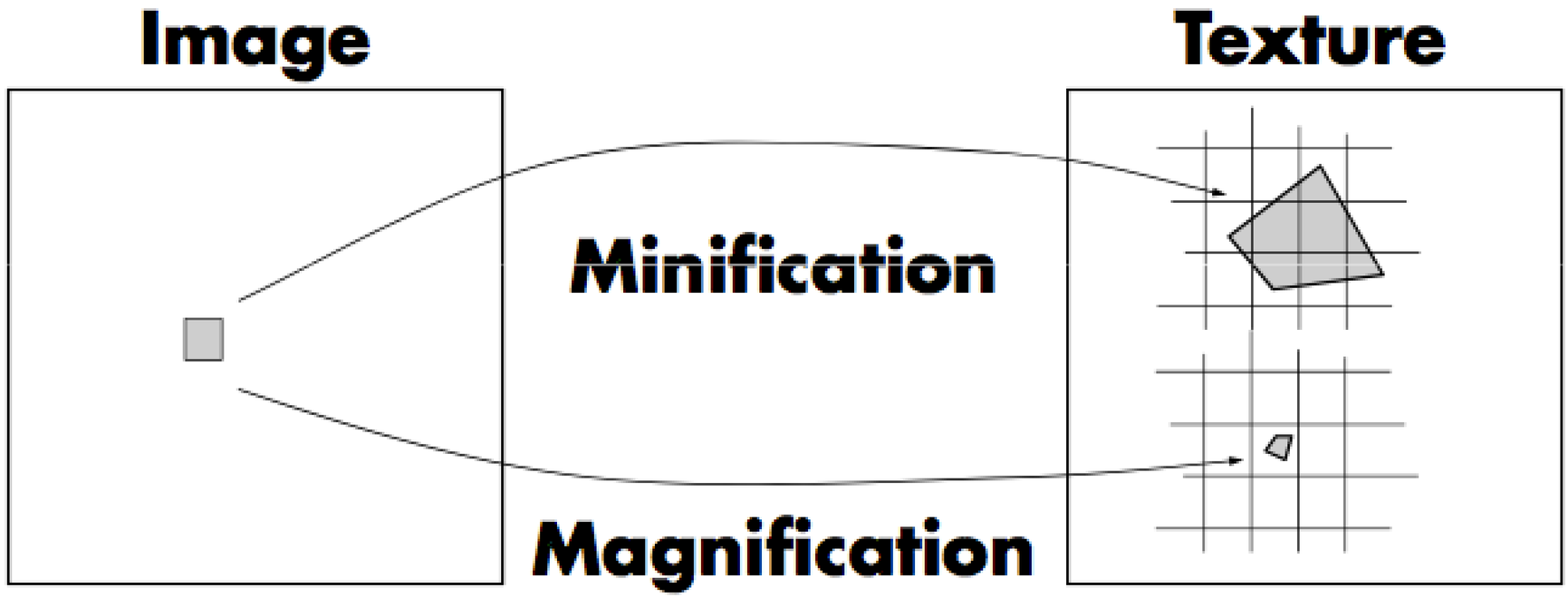
No pre-filtering of texture data



Rendering using pre-filtered texture data



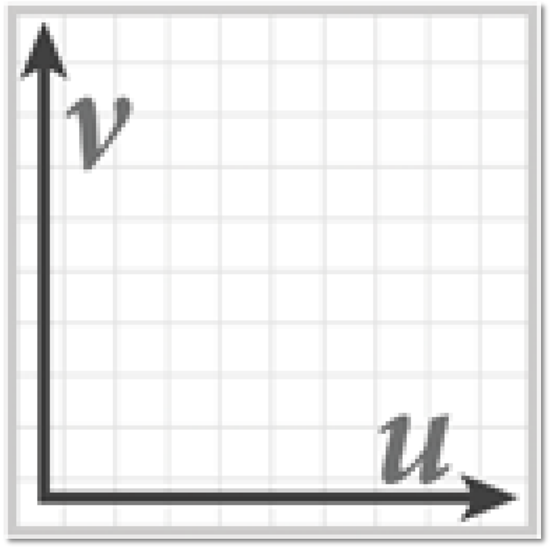
Filtering textures



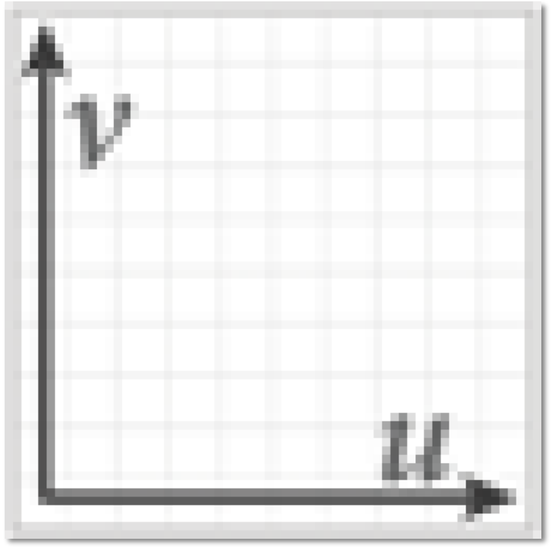
- **Minification:**
  - Area of screen pixel maps to large region of texture (filtering required -- averaging)
  - One texel corresponds to far less than a pixel on screen
  - Example: when scene object is very far away
  - Texture map is too large, it contains more details than screen can display
- **Magnification:**
  - Area of screen pixel maps to tiny region of texture (interpolation required)
  - One texel maps to many screen pixels
  - Example: when camera is very close to scene object
  - Texture map is too small

Figure credit: Akeley and Hanrahan

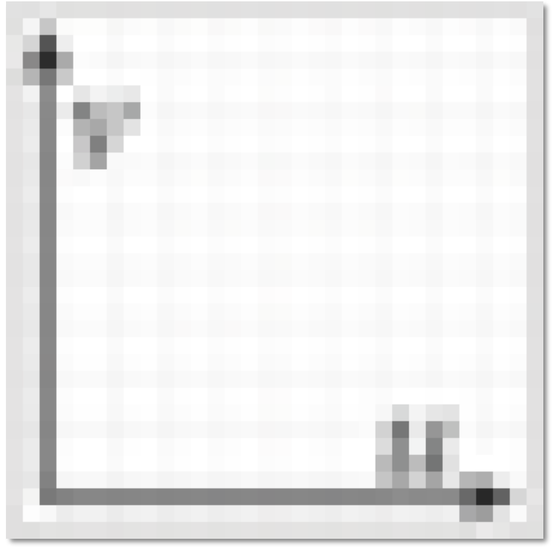
Mipmap (L. Williams 83)



Level 0 = 128x128



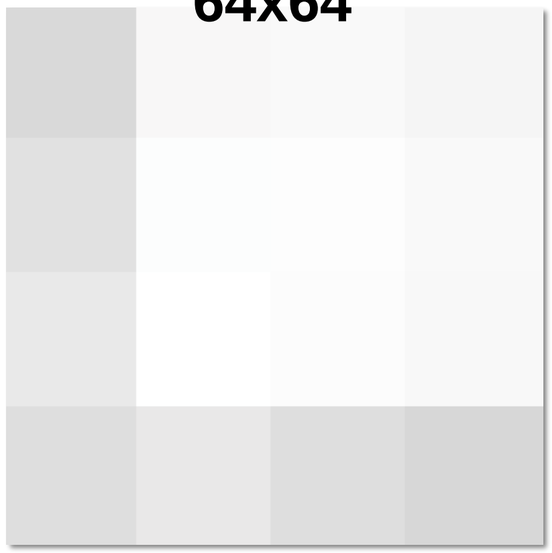
Level 1 = 64x64



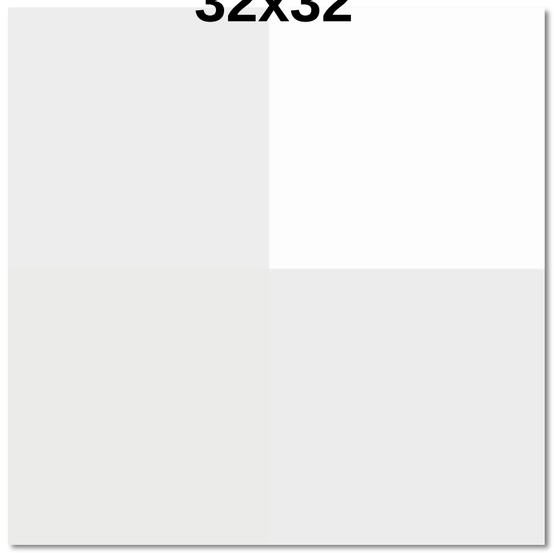
Level 2 = 32x32

Level 3 = 16x16

Level 4 = 8x8



Level 5 = 4x4



Level 6 = 2x2

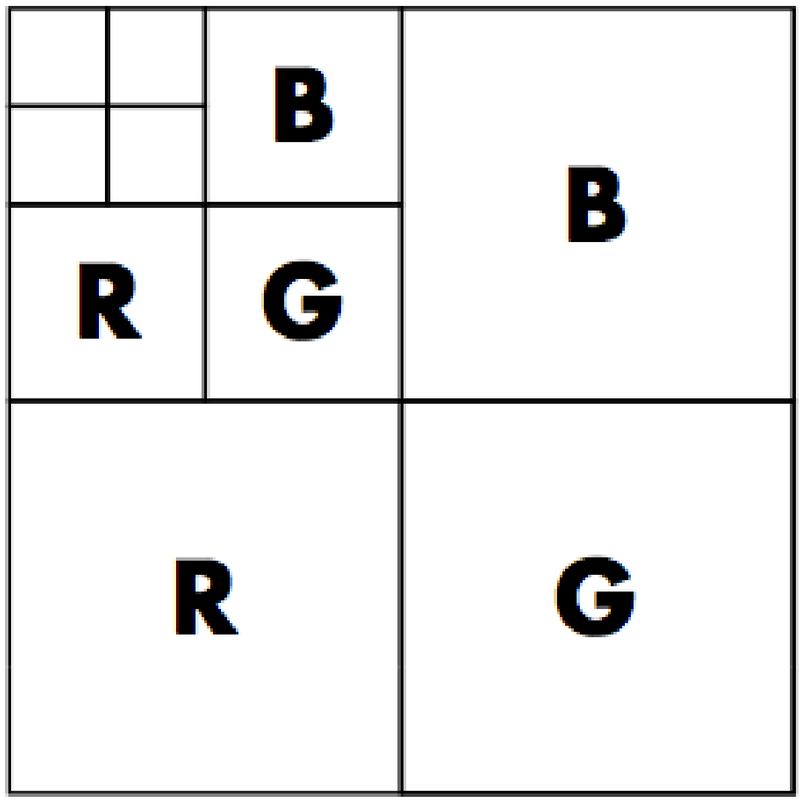
Level 7 = 1x1

Idea: prefilter texture data to remove high frequencies

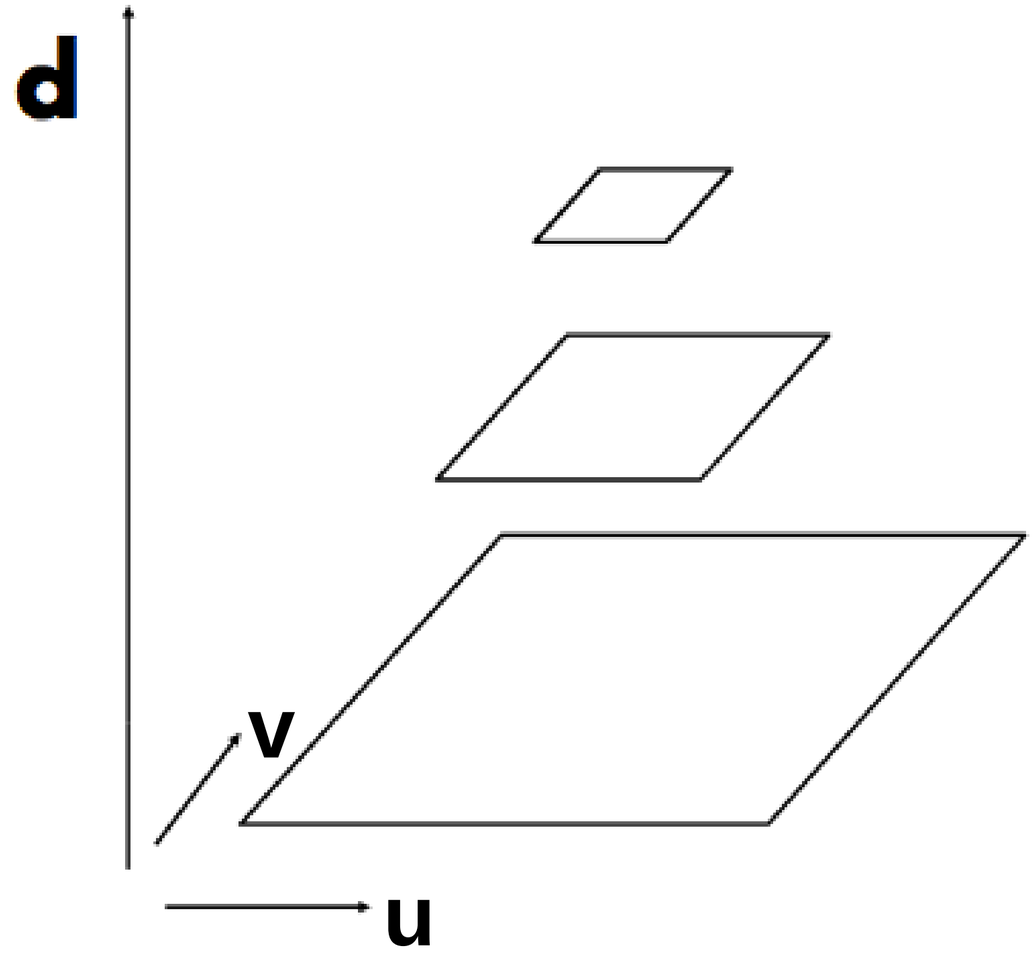
Texels at higher levels store integral of the texture function over a region of texture space (downsampled images)

Texels at higher levels represent low-pass filtered version of original texture signal

Mipmap (L. Williams 83)



Williams' original proposed mip-map layout



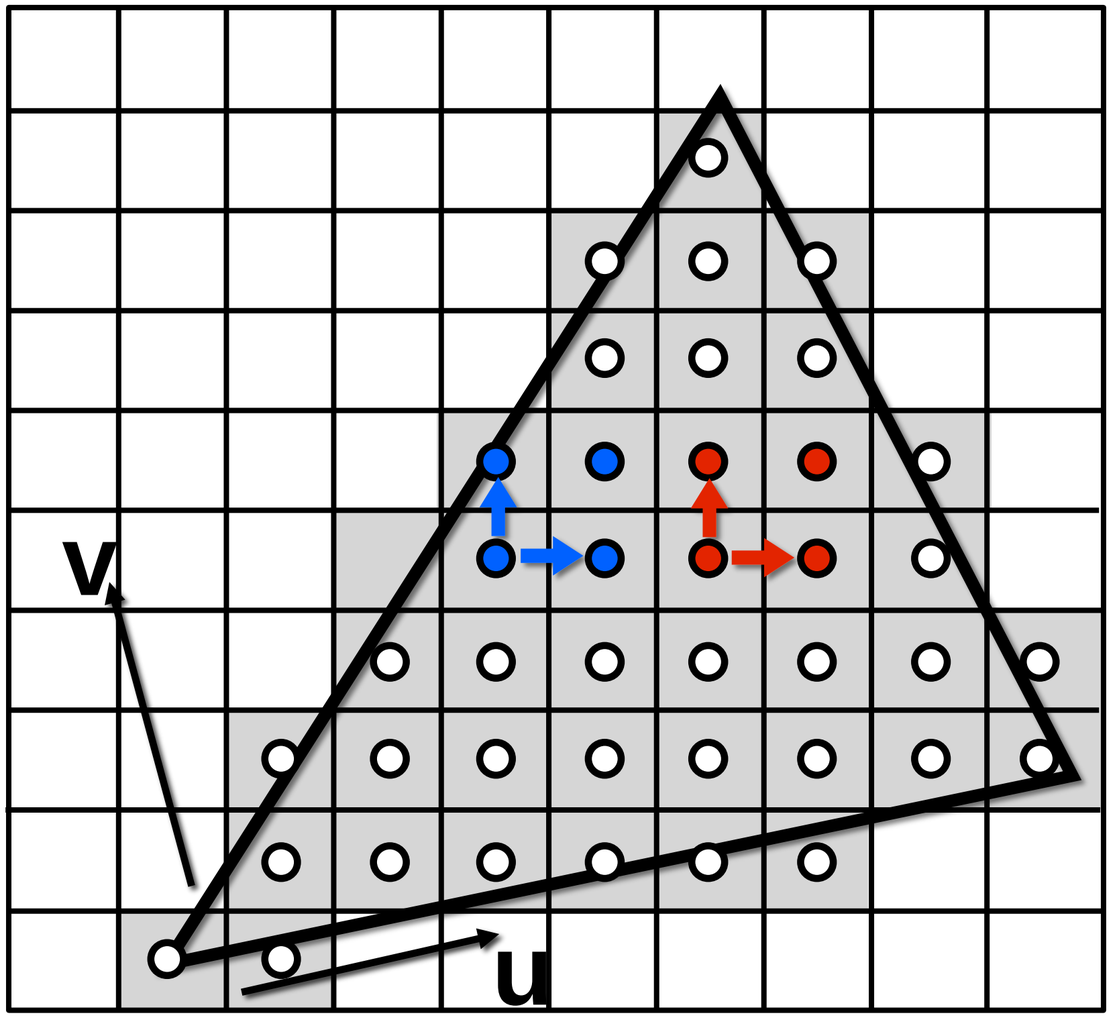
"Mip hierarchy" level = *d*

What is the storage overhead of a mipmap?

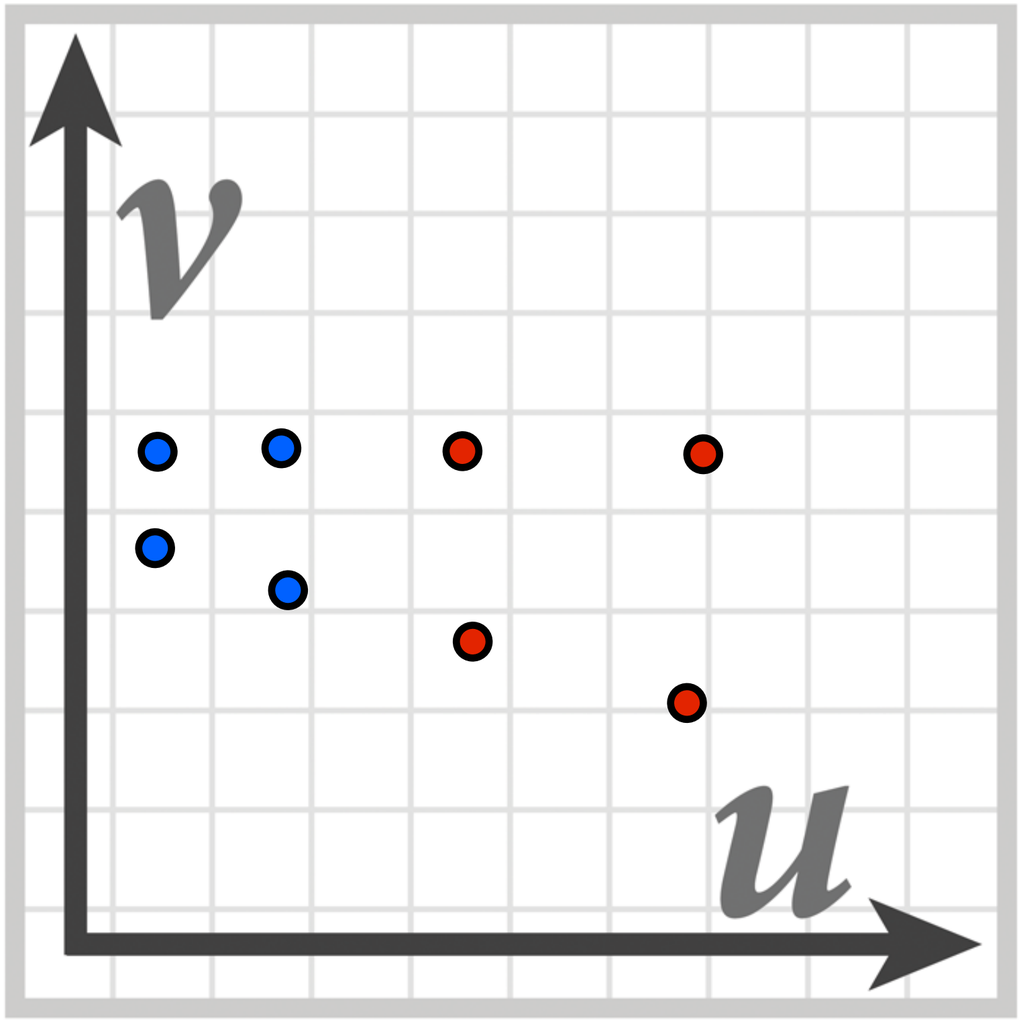
Slide credit: Akeley and Hanrahan

Computing *d*

Compute differences between texture coordinate values of neighboring screen samples



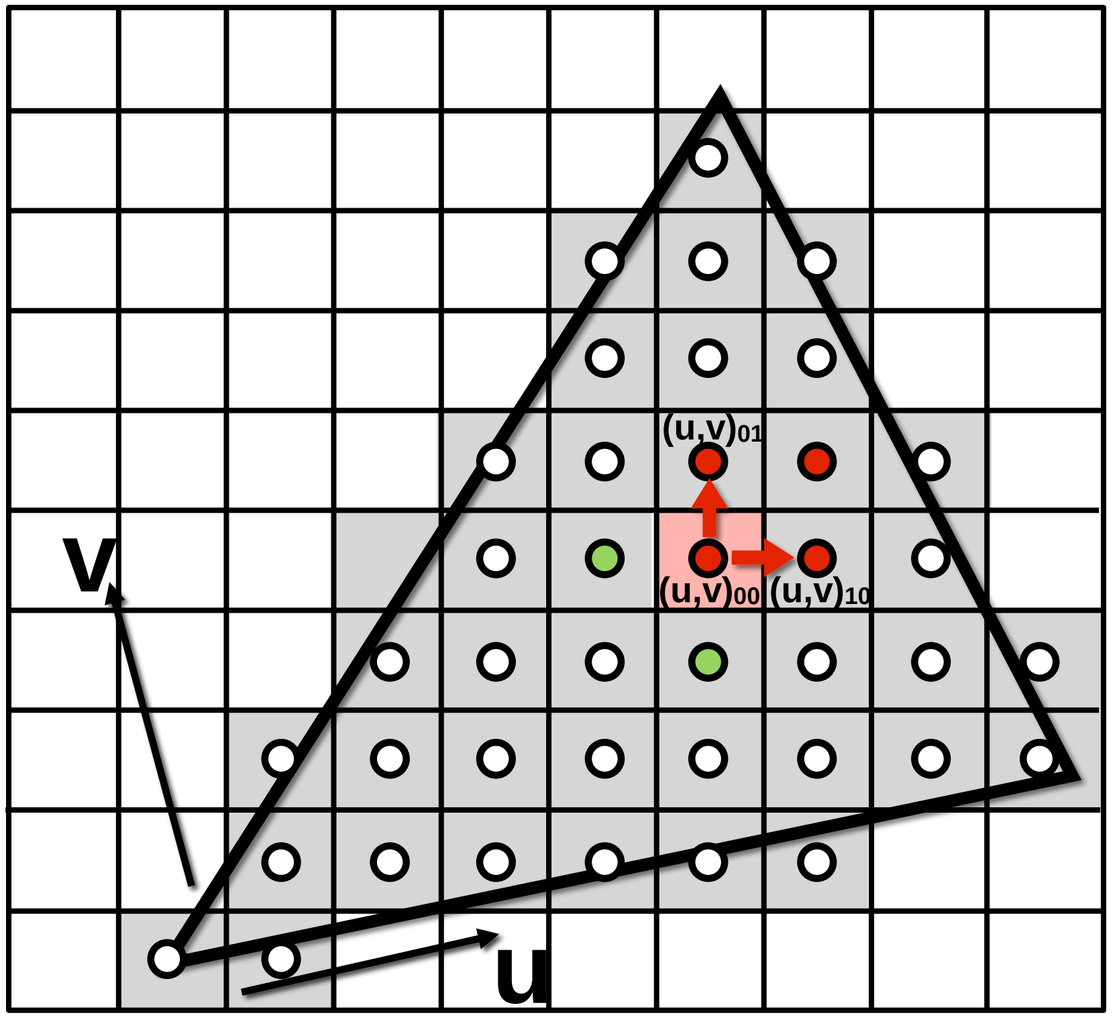
Screen space

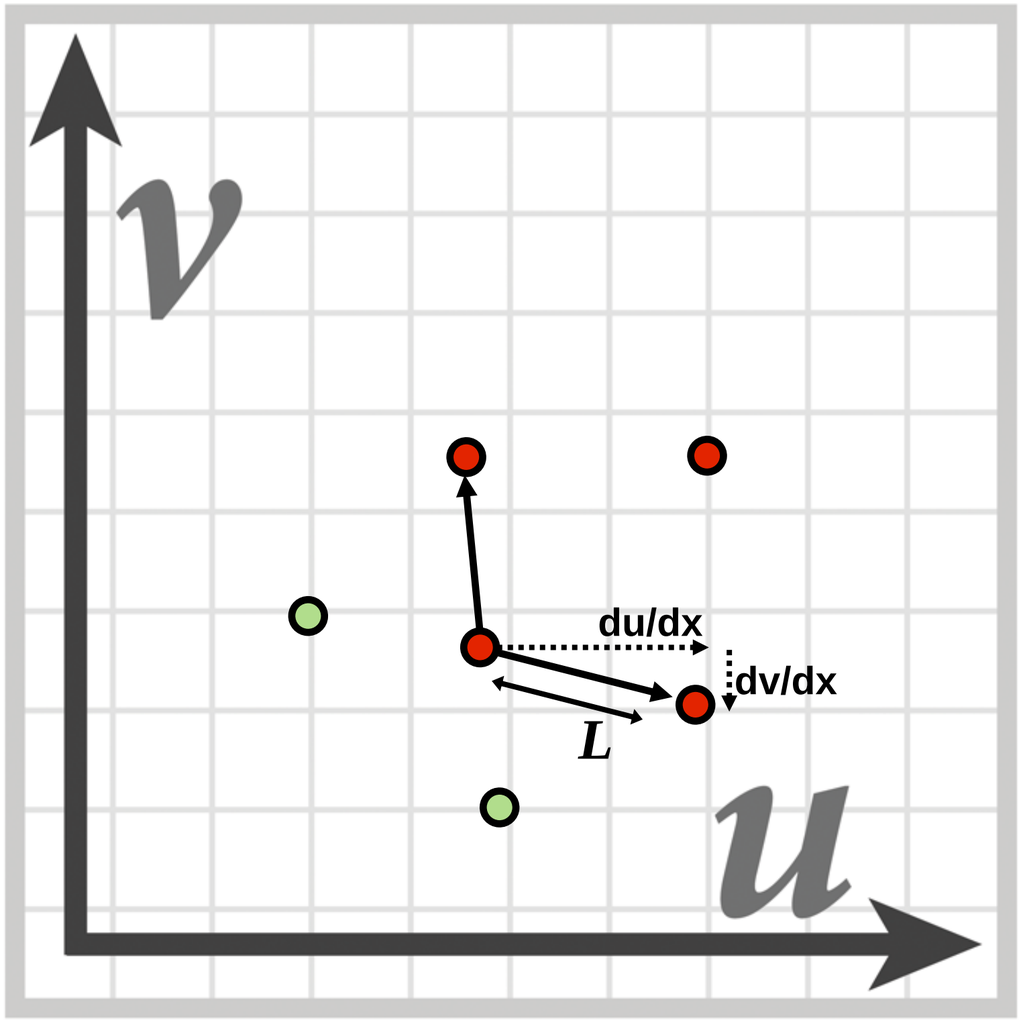


Texture space

Computing *d*

Compute differences between texture coordinate values of neighboring screen samples





$du/dx = u_{10}-u_{00}$ $dv/dx = v_{10}-v_{00}$

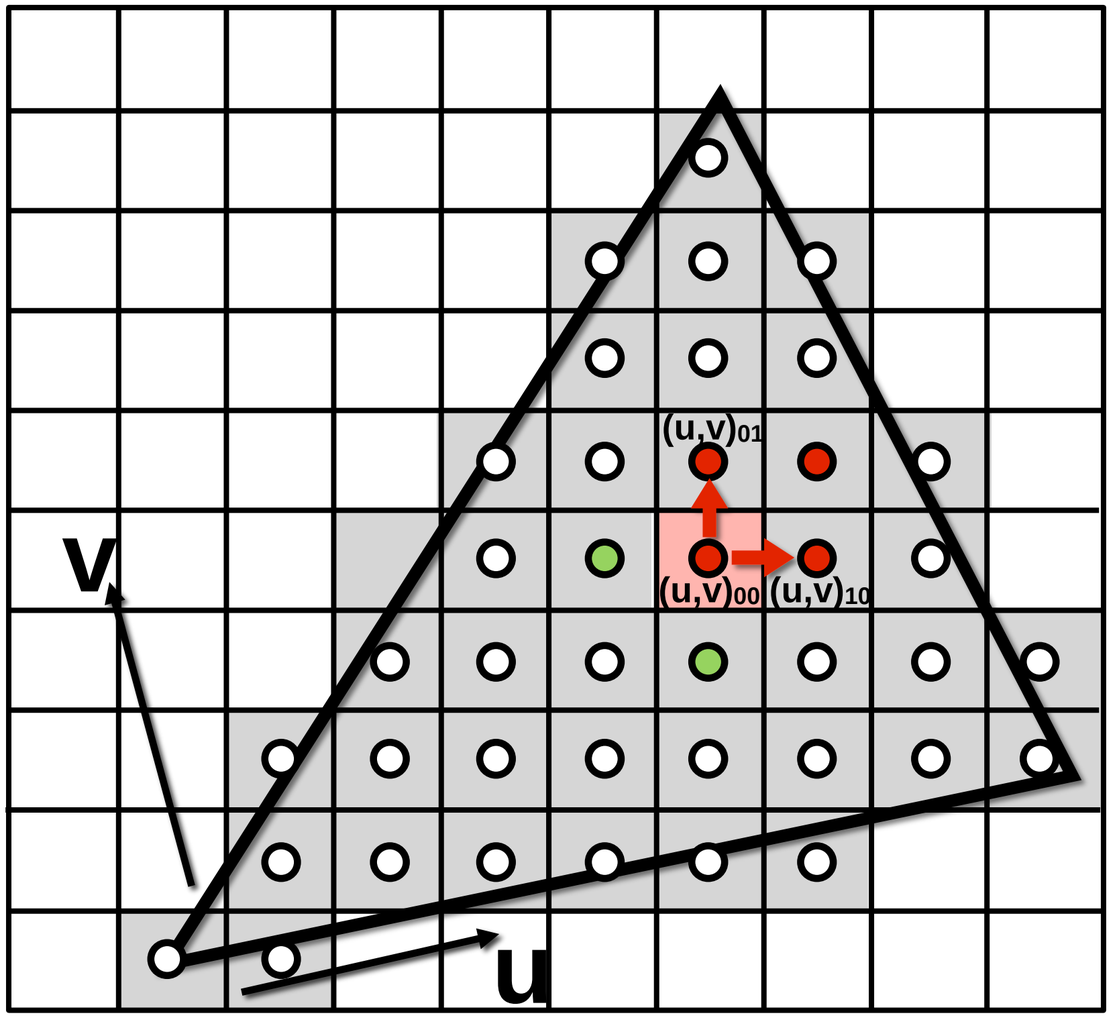
$du/dy = u_{01}-u_{00}$ $dv/dy = v_{01}-v_{00}$

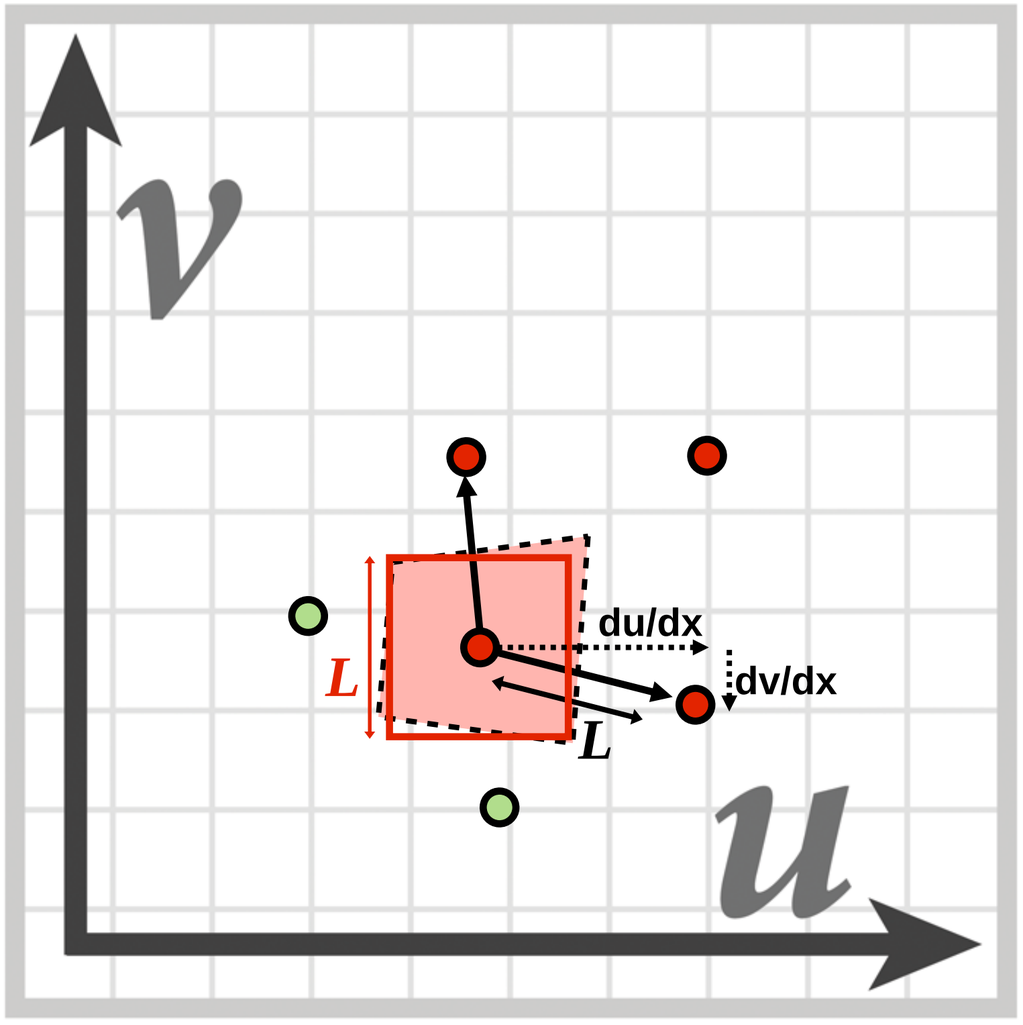
$$L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2+\left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2+\left(\frac{dv}{dy}\right)^2}\right)$$

mip-map d = log₂ L

Computing *d*

Compute differences between texture coordinate values of neighboring screen samples





$du/dx = u_{10}-u_{00}$ $dv/dx = v_{10}-v_{00}$

$du/dy = u_{01}-u_{00}$ $dv/dy = v_{01}-v_{00}$

$$L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2+\left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2+\left(\frac{dv}{dy}\right)^2}\right)$$

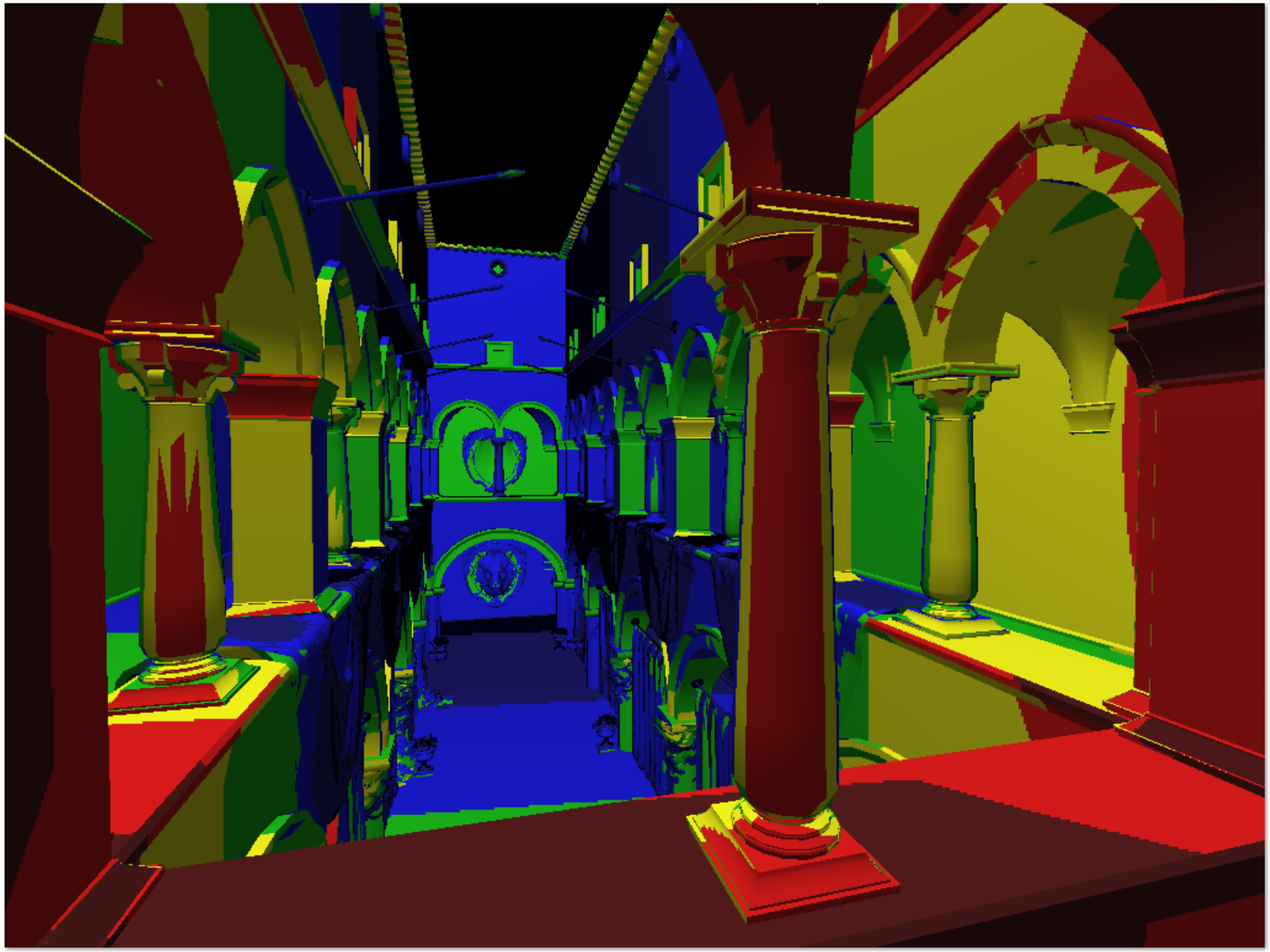
mip-map d = log₂ L

Sponza (bilinear resampling at level 0)

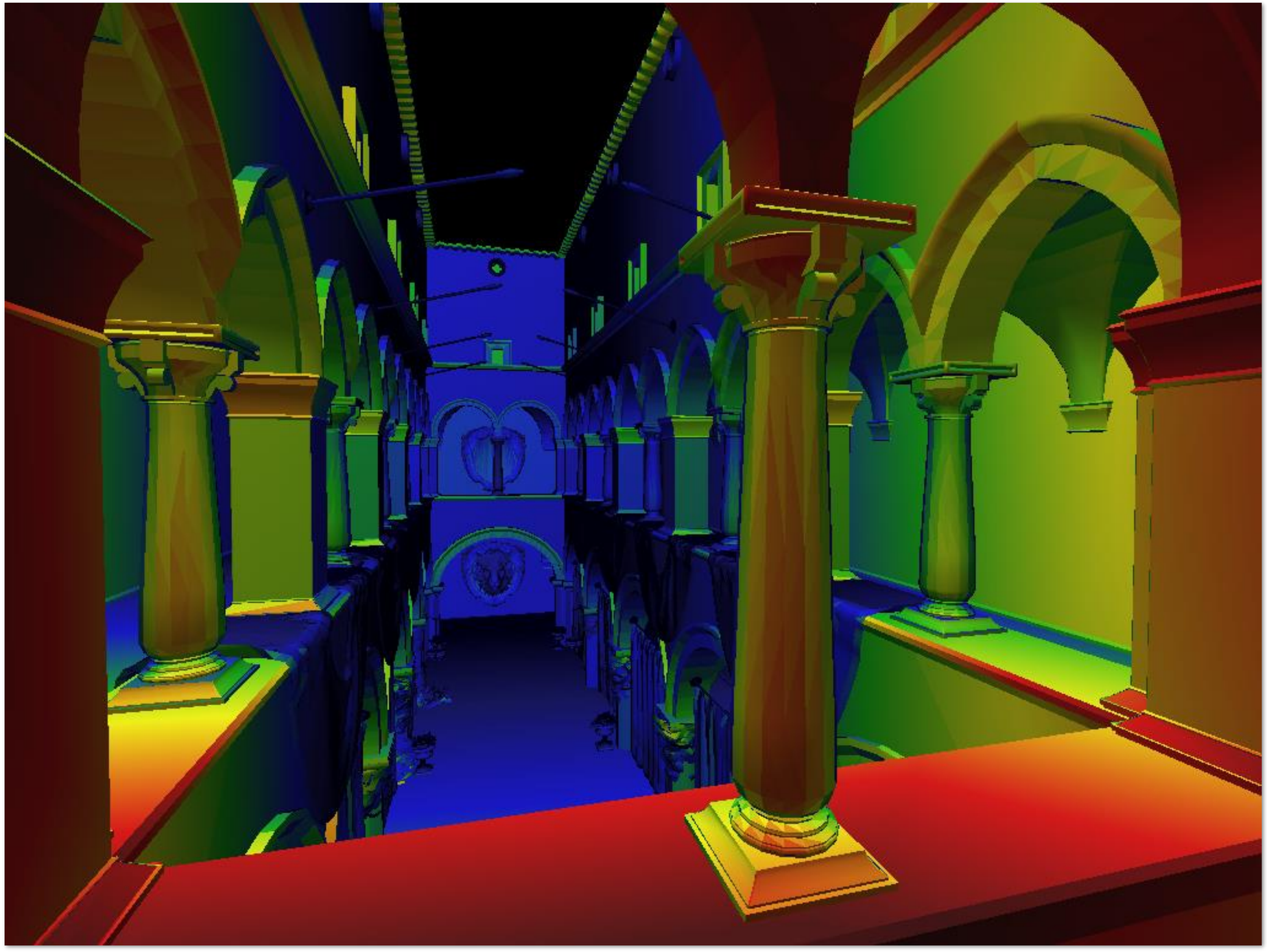
Sponza (bilinear resampling at level 2)

Sponza (bilinear resampling at level 4)

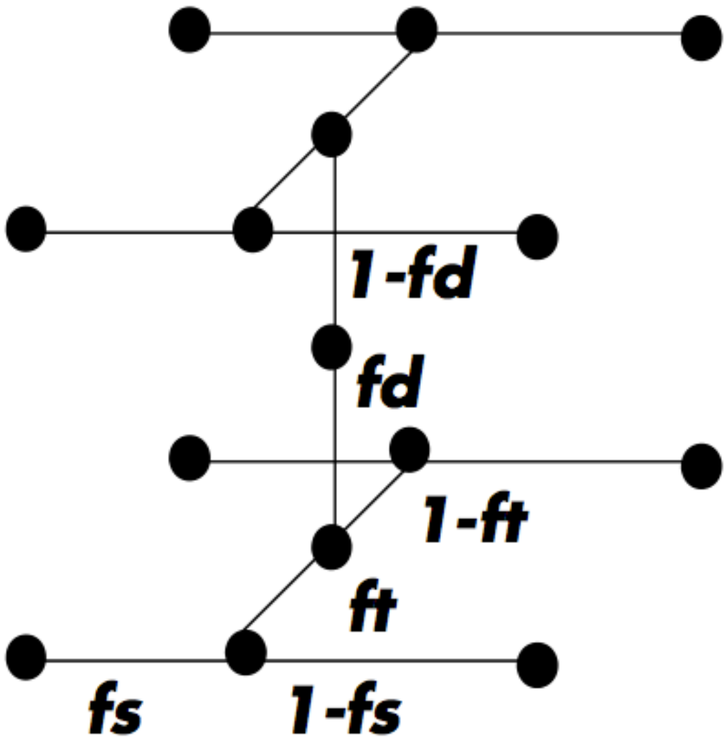
Visualization of mip-map level
(bilinear filtering only: *d* clamped to nearest level)



Visualization of mip-map level
(trilinear filtering: visualization of continuous *d*)



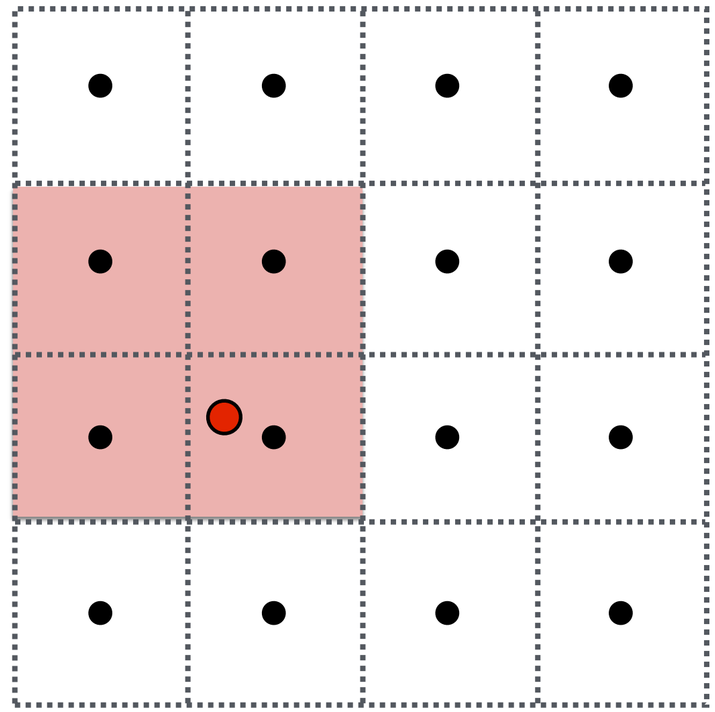
“Tri-linear” filtering



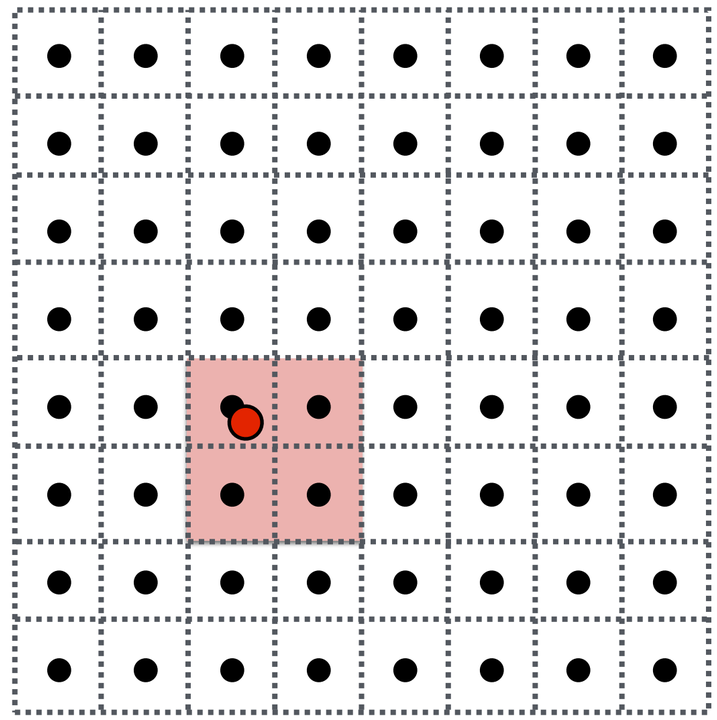
$$lerp(t, v_1, v_2) = v_1 + t(v_2 - v_1)$$

Bilinear resampling:
four texel reads
3 lerps (3 mul + 6 add)

Trilinear resampling:
eight telex reads
7 lerps (7 mul + 14 add)



mip-map texels: level d+1

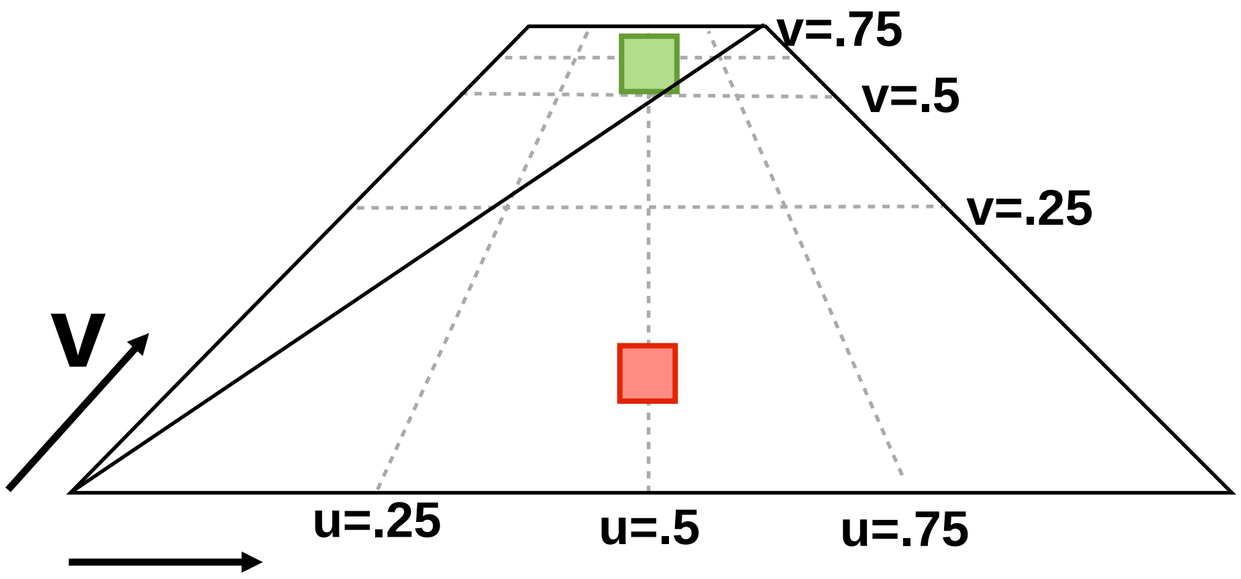


mip-map texels: level d

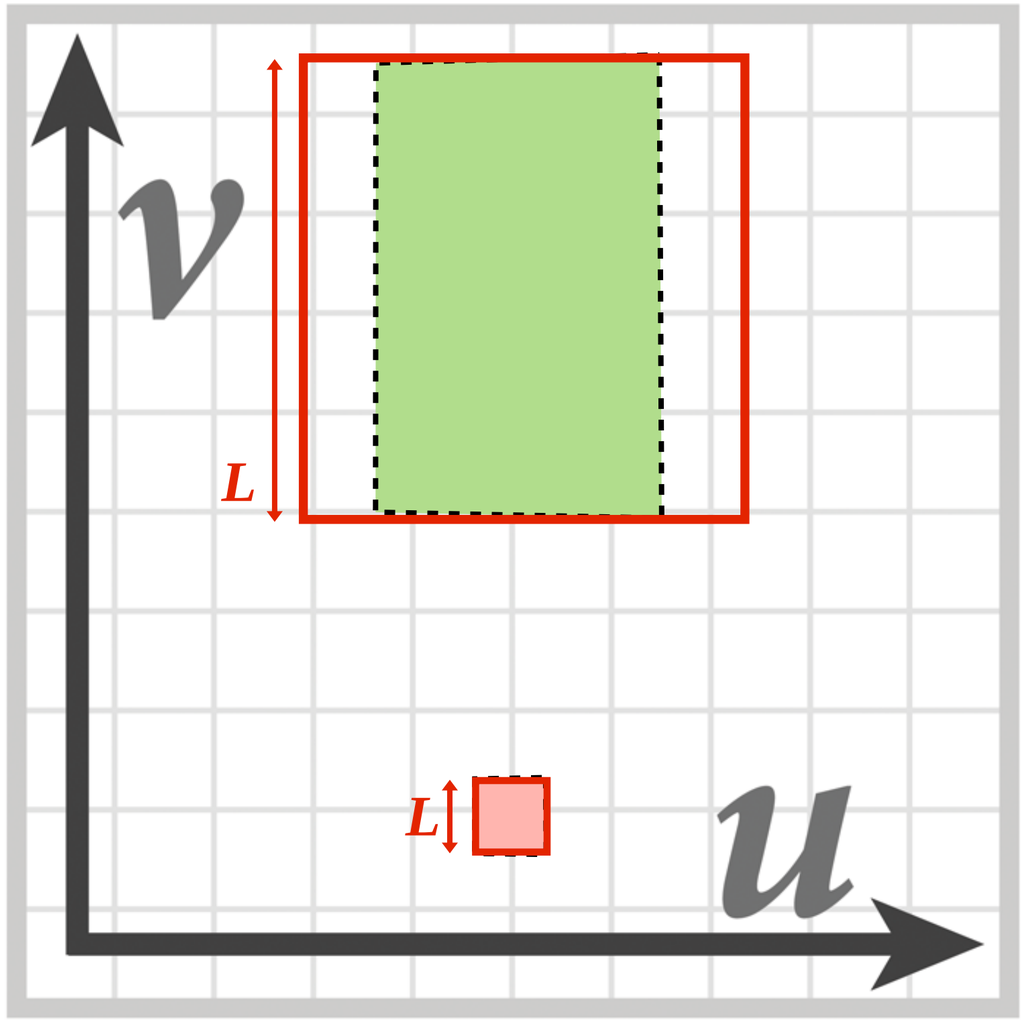
Figure credit: Akeley and Hanrahan

Pixel area may not map to isotropic region in texture space

Proper filtering requires anisotropic filter footprint

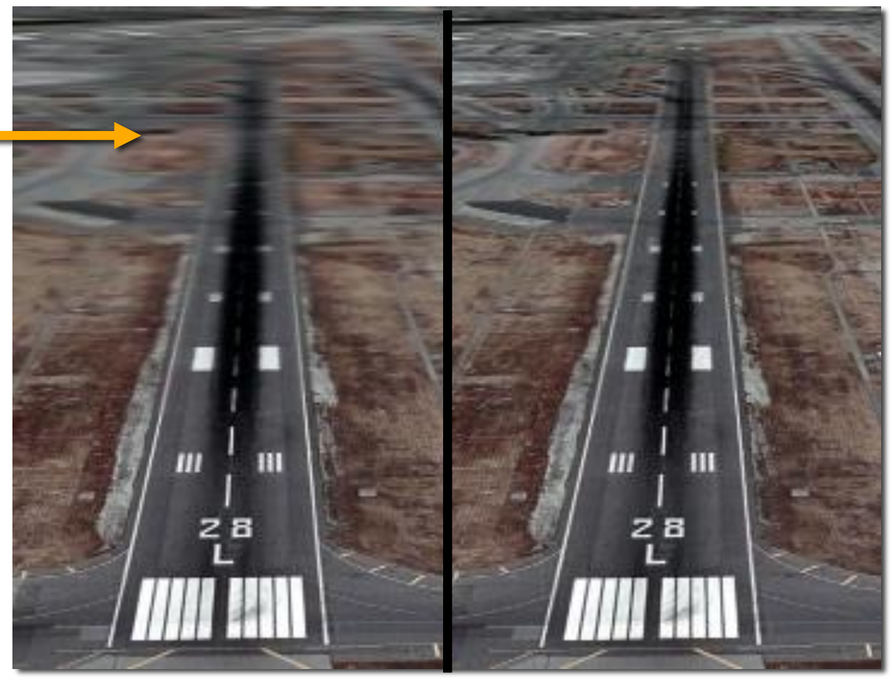


Texture space: viewed from camera with perspective



Modern solution: Combine multiple mipmap samples

Overblurring in *u* direction



Trilinear (Isotropic) Filtering

Anisotropic Filtering

Summary: texture filtering using the mip map

- **Small storage overhead (33%)**
  - **Mipmap is 4/3 the size of original texture image**
- **For each isotropically-filtered sampling operation**
  - **Constant filtering cost (independent of *d*)**
  - **Constant number of texels accessed (independent of *d*)**
- **Combat aliasing with prefiltering, rather than supersampling**
  - **Recall: we used supersampling to address aliasing problem when sampling coverage**
- **Bilinear/trilinear filtering is isotropic and thus will "overblur" to avoid aliasing**
  - **Anisotropic texture filtering provides higher image quality at higher compute and memory bandwidth cost**

Summary: a texture sampling operation

1. **Compute u and v from screen sample x,y (via evaluation of attribute equations)**
2. **Compute du/dx, du/dy, dv/dx, dv/dy differentials from screen-adjacent samples.**
3. **Compute *d***
4. **Convert normalized texture coordinate (u,v) to texture coordinates texel_u, texel_v**
5. **Compute required texels in window of filter**
6. **Load required texels (need eight texels for trilinear)**
7. **Perform tri-linear interpolation according to (texel_u, texel_v, d)**

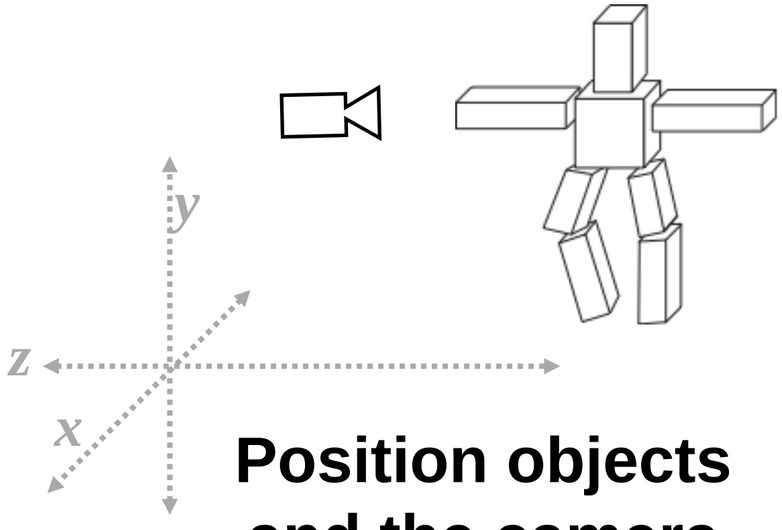
Takeaway: a texture sampling operation is not "just" an image pixel lookup! It involves a significant amount of math.

All modern GPUs have dedicated hardware support for performing texture sampling operations.

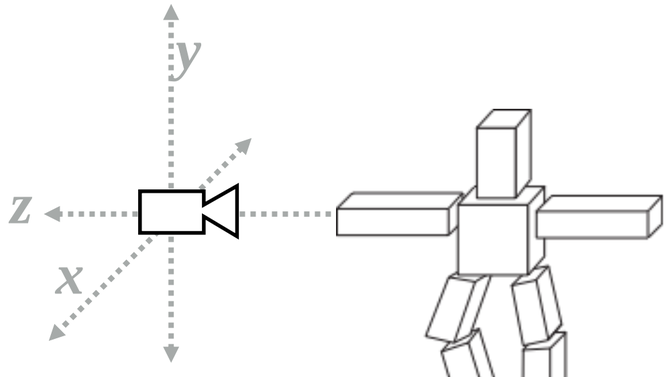
Texturing summary

- Texture coordinates: define mapping between points on triangle's surface (object coordinate space) to points in texture coordinate space
- Texture mapping is a sampling operation and is prone to aliasing
  - Solution: prefilter texture map to eliminate high frequencies in texture signal
  - Mip-map: precompute and store multiple resampled versions of the texture image (each with different amounts of low-pass filtering)
  - During rendering: dynamically select how much low-pass filtering is required based on distance between neighboring screen samples in texture space
    - Goal is to retain as much high-frequency content (detail) in the texture as possible, while avoiding aliasing

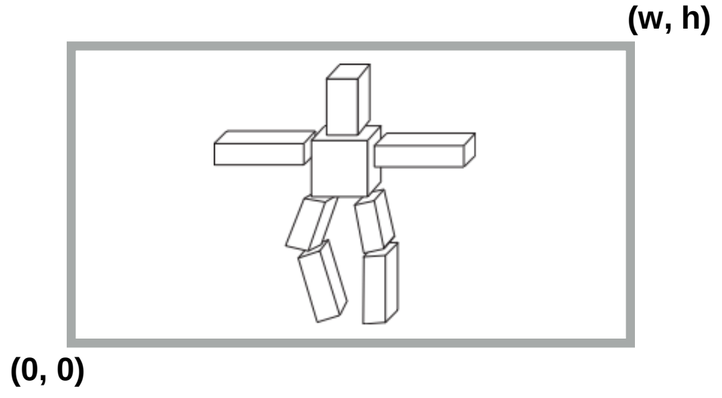
What you know how to do (at this point in the course)



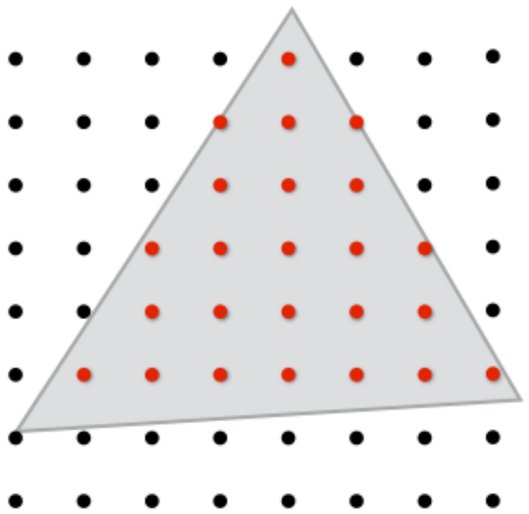
Position objects and the camera in the world



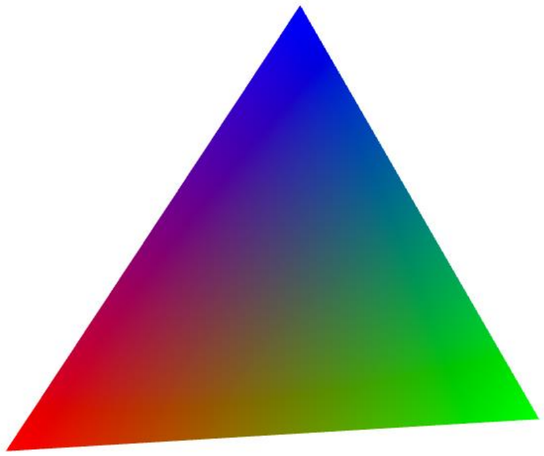
Determine the position of objects relative to the camera



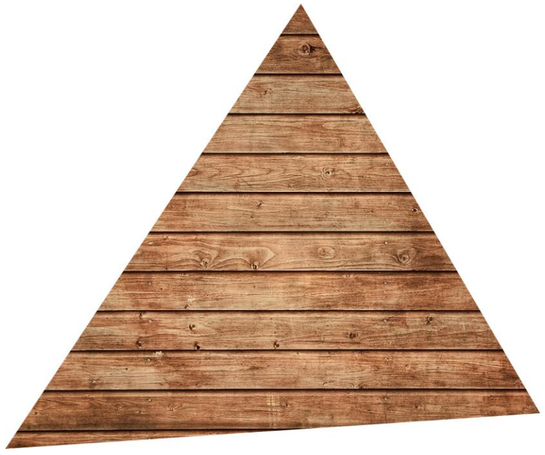
Project objects onto the screen



Sample triangle coverage



Compute triangle attribute values at covered sample points



Sample texture maps