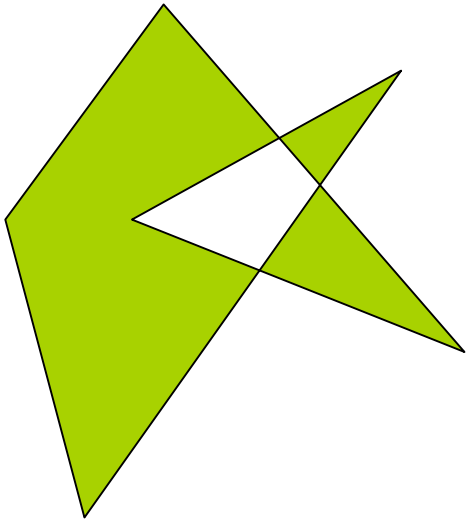
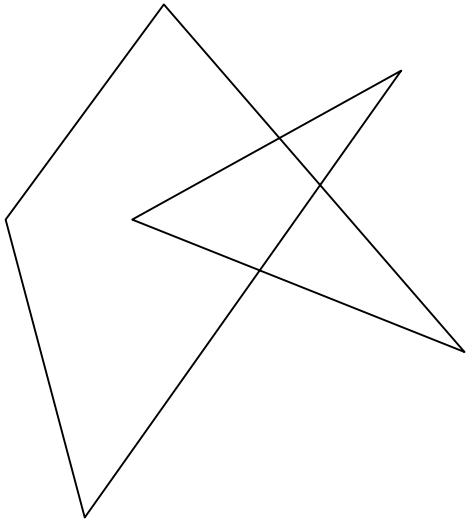
COMP175: Computer Graphics

Lecture 6
Coloring and Texturing Polygons

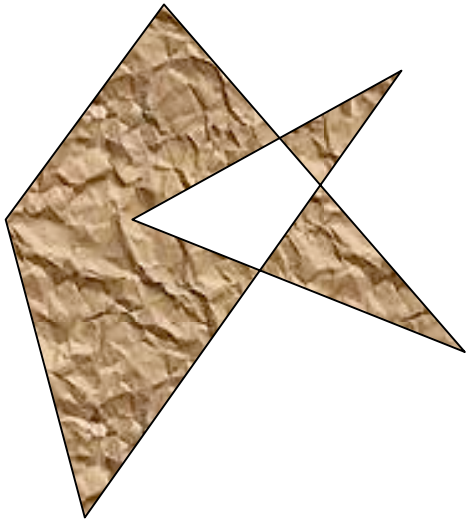
Filling 2D shapes



Solid fill

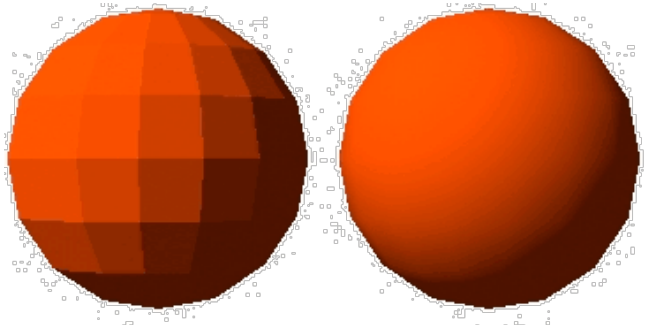


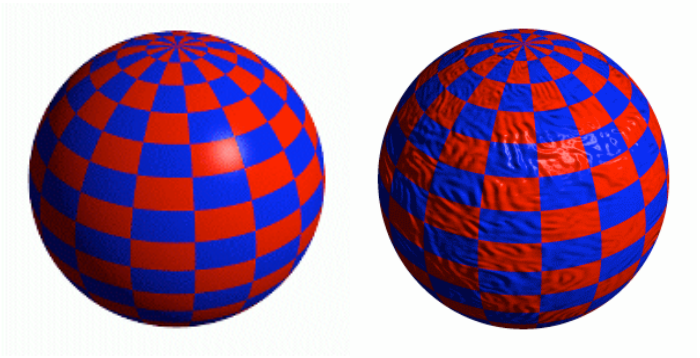
Patterned fill

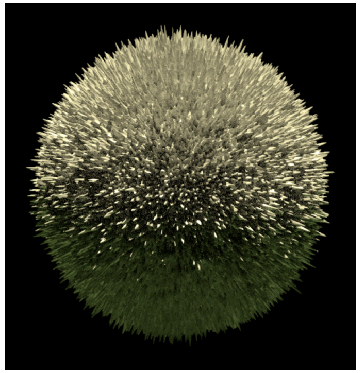


Textured fill

Coloring and texturing geometry





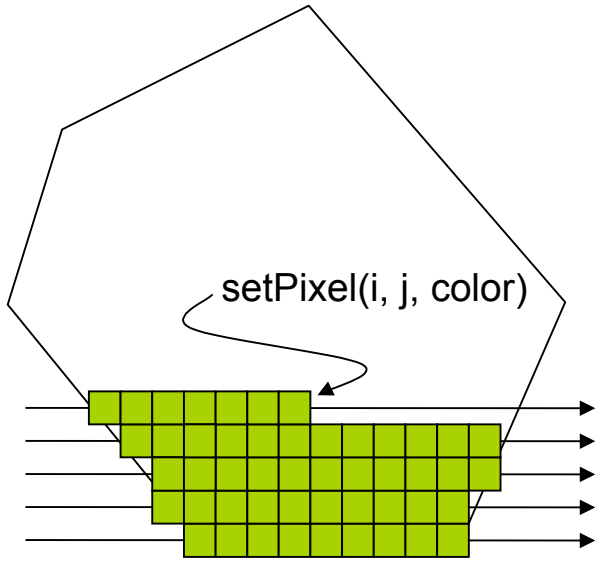


Solid fill

Set the pixels inside a polygon to a solid color

Rasterize a polygon

1. Determine which pixels along a scan line are inside the polygon
2. Set the color of these pixels using a function setPixel(i, j, color)

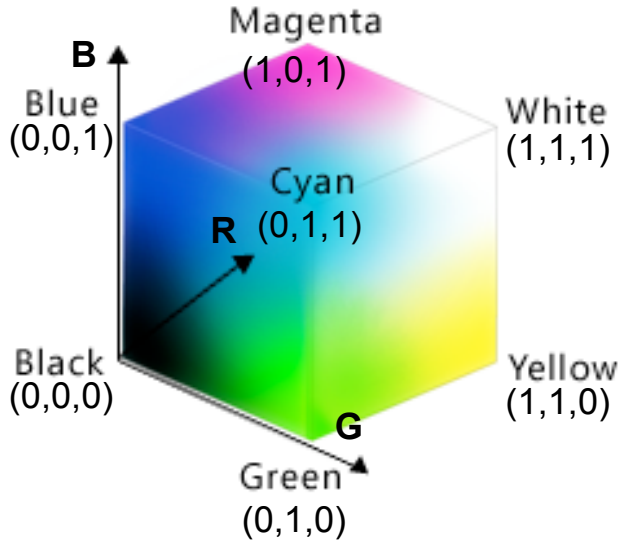


Representing color

We will use the RGB color space

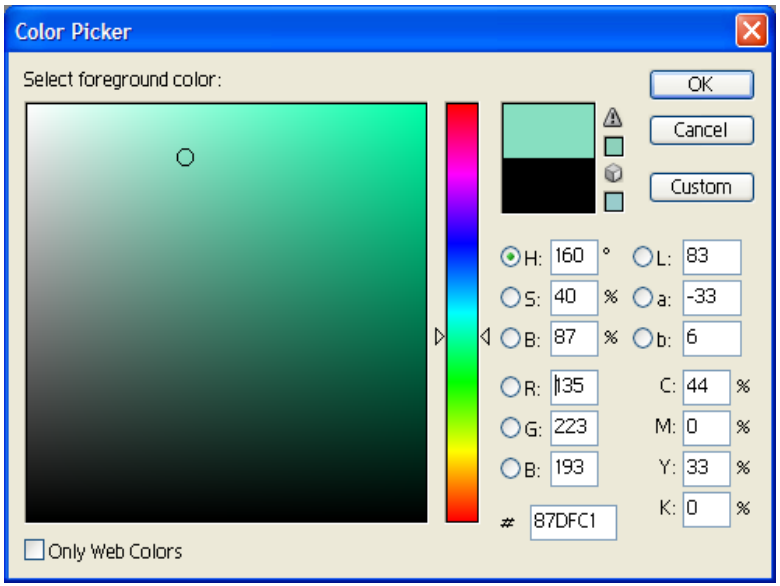
Each pixel has three floating point color components

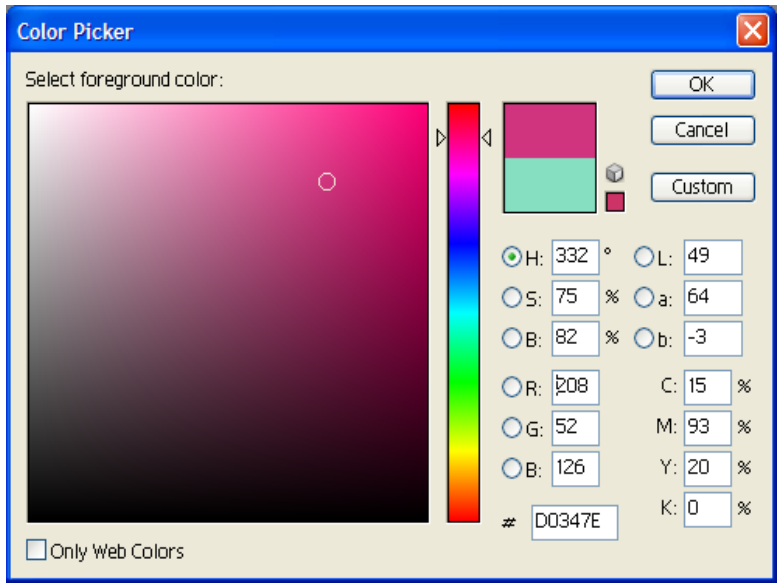
Color	R	G	B
Black	0.0	0.0	0.0
White	1.0	1.0	1.0
Red	1.0	0.0	0.0
Yellow	1.0	1.0	0.0
Green	0.0	1.0	0.0
Cyan	0.0	1.0	1.0
Blue	0.0	0.0	1.0
Magenta	1.0	0.0	1.0



Source: Adapted from Windows User Experience Interaction Guidelines

Representing color

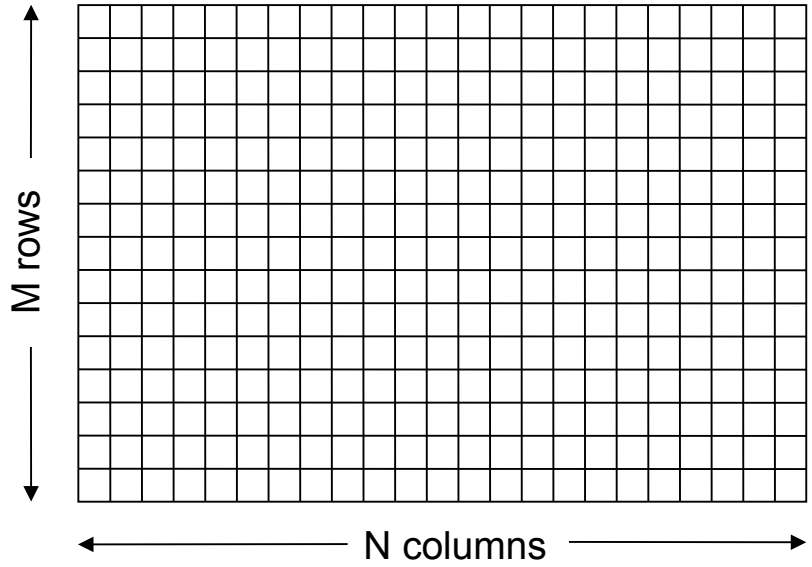




Representing images

Images are represented in a single linear array

- Each image pixel consists of three floating point values
- An M*N image requires 3*M*N floats



- If we know the size in advance:

```
float image[3*W*H]
```

- If we don't know the size in advance:

```
float *image=new float[3*W*H];
...
delete [] image;
```

Solid fill: Rasterization

To fill with a solid color (R, G, B), set the color of each pixel[i][j] inside the polygon during rasterization

image[3(i+j*W)] = R
image[3(i+j*W)+1] = G
image[3(i+j*W)+2] = B

Gradient fill

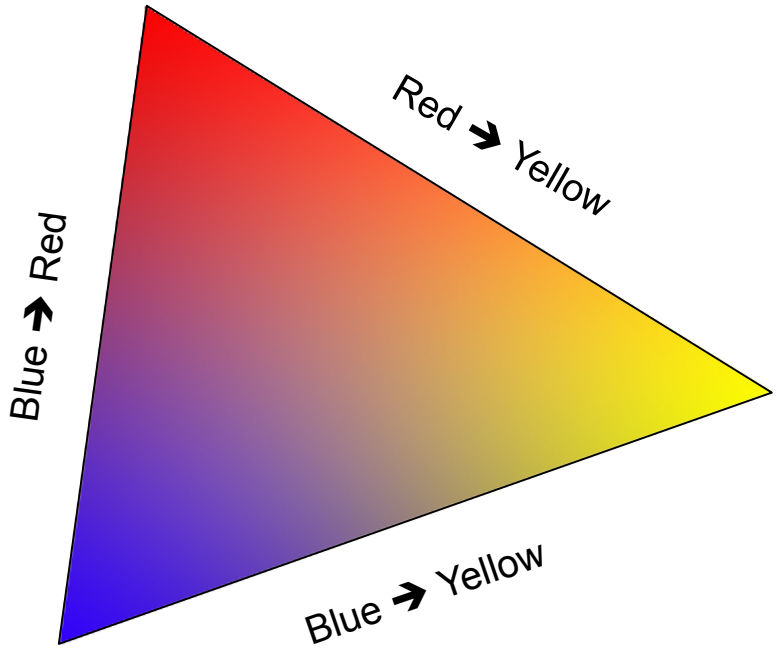
Define a color for each polygon vertex,
e.g., for a triangle with vertices V_0, V_1, and V_2

$$\text{color}(V_0) = (R_0, G_0, B_0)$$
$$\text{color}(V_1) = (R_1, G_1, B_1)$$
$$\text{color}(V_2) = (R_2, G_2, B_2)$$

Interpolate the vertex colors across the face of the polygon

Interpolating colors

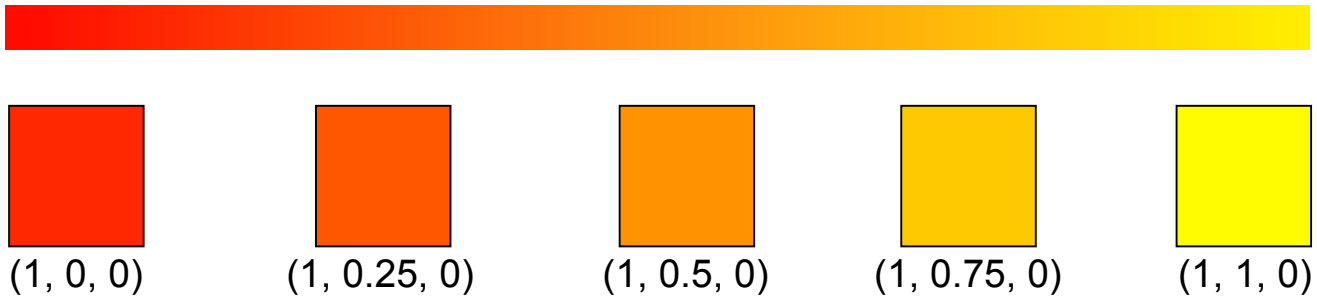
Color changes across the face of the polygon between vertices
Colors between vertices are linearly interpolated



Linear interpolation

How do we interpolate colors?

e.g., From red to yellow along an edge



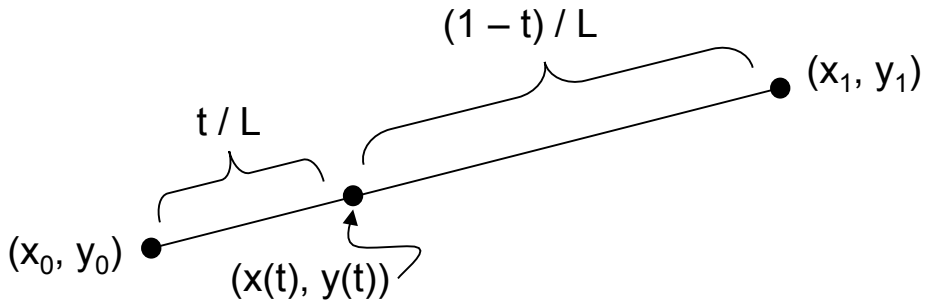
Depends on the color space

In RGB, interpolate from (1, 0, 0) to (1, 1, 0), so half way between red and yellow is (1, 0.5, 0)

Linear interpolation

$$(x(t), y(t)) = (1-t)(x_0, y_0) + t(x_1, y_1)$$

Linearly interpolates x and y-coordinates of the endpoints
Produces points on the straight line between the two endpoints



Linear interpolation

Same approach to linearly interpolate other properties

$$\text{Color } C(t) = (1-t)C_0 + tC_1$$

$$R(t) = (1-t)R_0 + tR_1$$
$$G(t) = (1-t)G_0 + tG_1$$
$$B(t) = (1-t)B_0 + tB_1$$

RGB does not always produce the best color interpolation

May be better to interpolate colors in HSV or other color spaces

Gradient fill: Rasterization

For each edge of the polygon, linearly interpolate the color along the edge to determine the color at each x-intersection point:

$$\text{Color } C(j) = (1-t)C_0 + tC_1$$

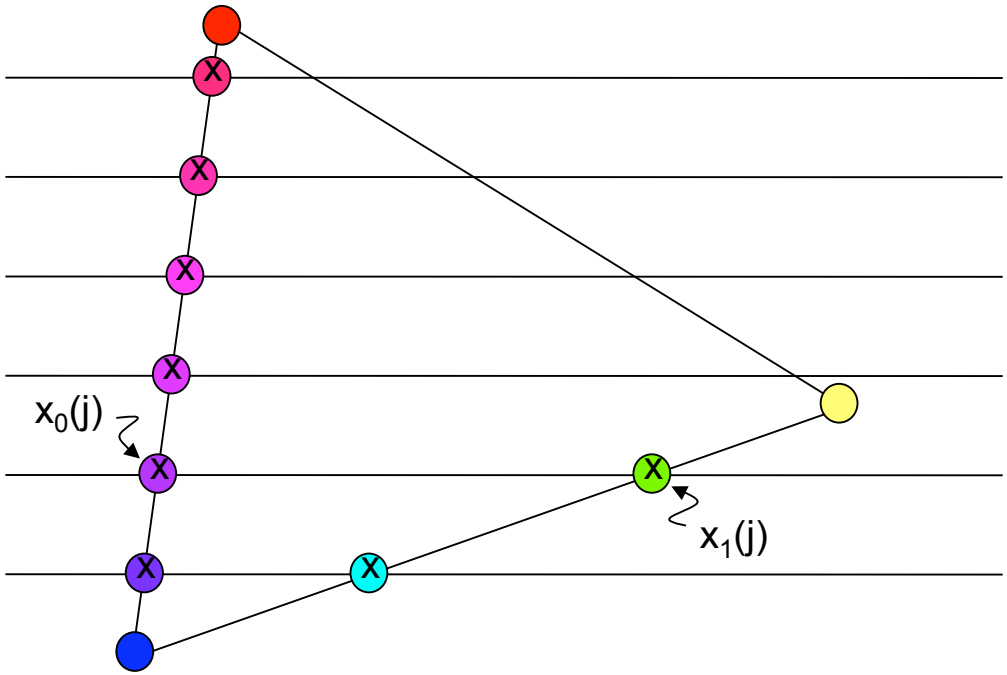
Where

- j is a scan line
- edge runs from (x_0, y_0) to (x_1, y_1)
- C_0 is the color at (x_0, y_0)
- C_1 is the color at (x_1, y_1)
- $t = (j – y_0) / (y_1 – y_0)$

Gradient fill: Rasterization

For each edge of the polygon, linearly interpolate the color along the edge to determine the color at each x-intersection point:

Color $C(j) = (1-t)C_0 + tC_1$



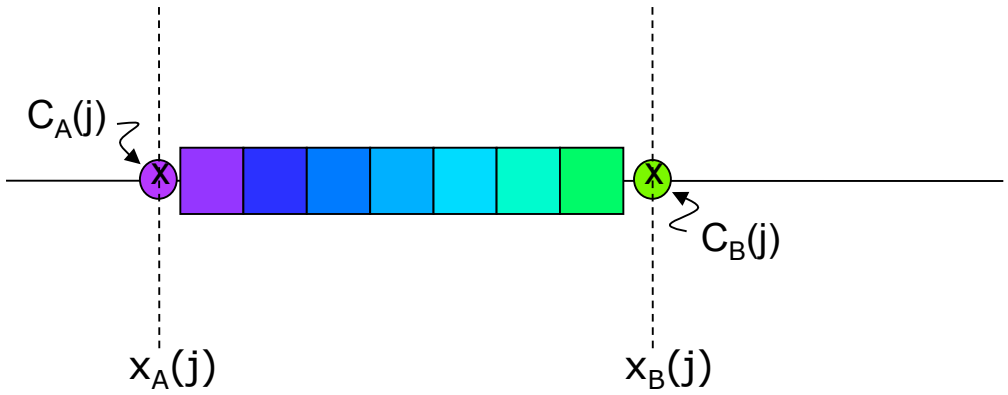
Gradient fill: Rasterization

Fill spans of pixels between pairs of x-intersections

Given a pair of interpolated colors $C_A(j)$ and $C_B(j)$ at intersections on j, fill the span of pixels between them with colors

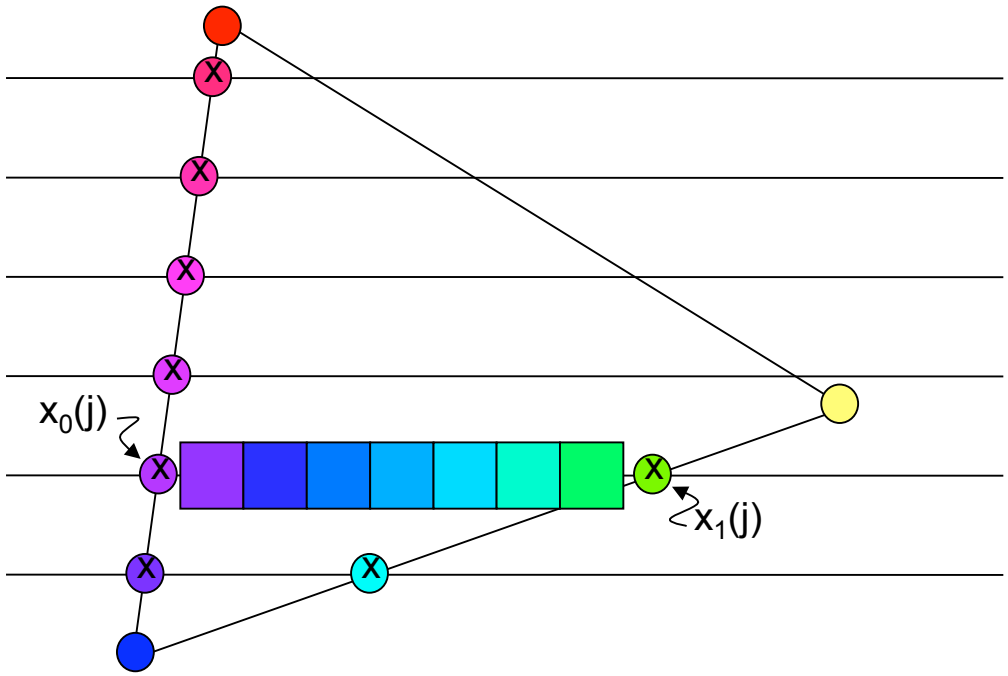
$$C(i, j) = (1\text{-}s)\, C_A(j) + s\, C_B(j)$$

Where $s = (i – x_A(j)) / (x_B(j)\text{-}x_A(j))$



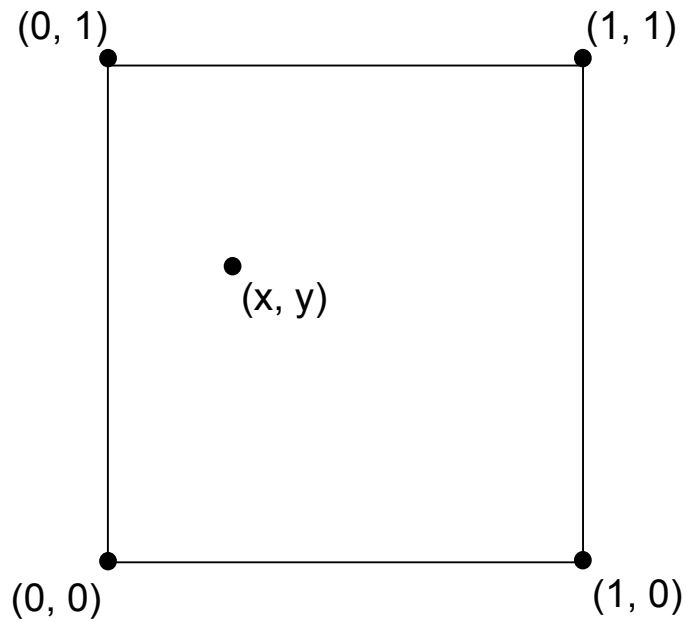
Gradient fill: Rasterization

Fill spans of pixels between pairs of x-intersections



Bilinear interpolation

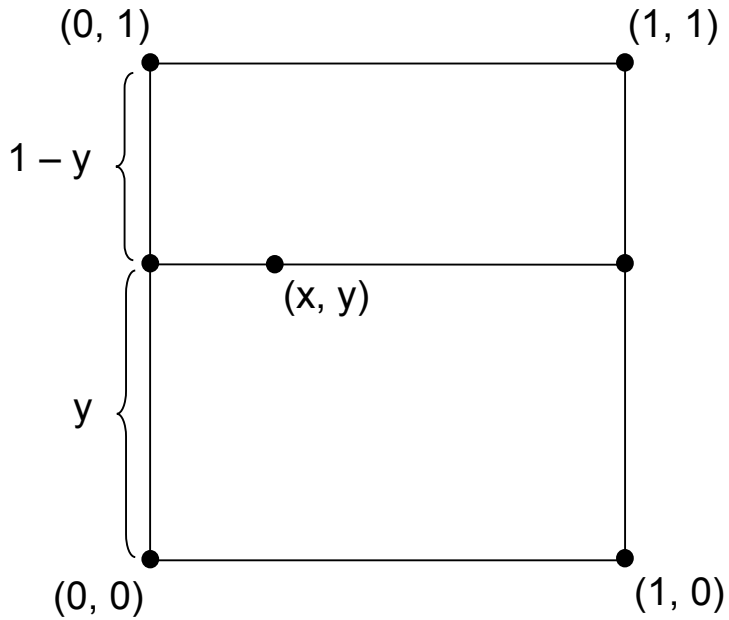
Consider the special case of a unit square polygon
Given C_{00}, C_{10}, C_{01}, C_{11}, determine C_{xy}



Bilinear interpolation

Determine C_{0y} and C_{1y} using linear interpolation along vertical edges

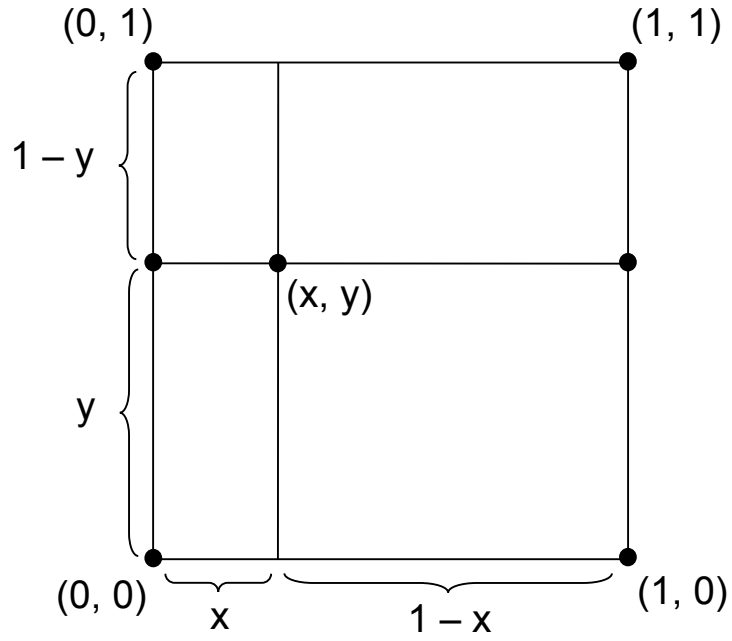
$$C_{0y} = (1 - y)C_{00} + yC_{01}$$
$$C_{1y} = (1 - y)C_{10} + yC_{11}$$



Bilinear interpolation

Determine C_{xy} using interpolation along the horizontal scan line

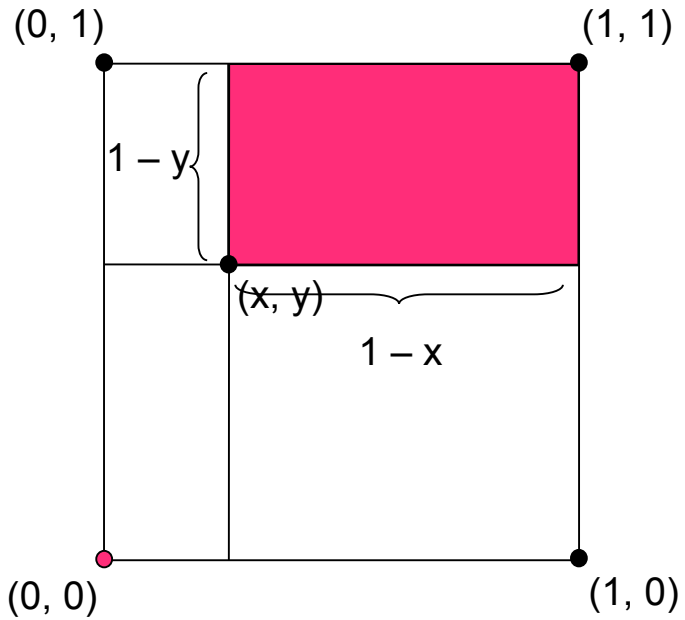
$$C_{xy} = (1 - x)C_{0y} + xC_{1y}$$
$$= (1 - x)(1 - y)C_{00} + (1 - x)yC_{01} + x(1 - y)C_{10} + xyC_{11}$$



Bilinear interpolation

The color contribution of each vertex is proportional to the area of the region opposite it

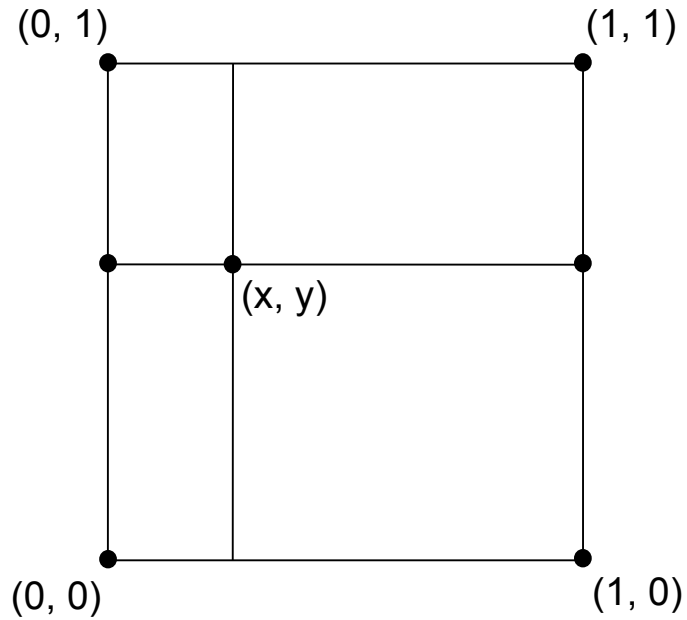
$$C_{xy} = (1 - x)(1 - y)C_{00} + (1 - x)yC_{01} + x(1 - y)C_{10} + xyC_{11}$$



Bilinear interpolation

This **bilinear interpolation** is commonly used to interpolate colors between integer pixel locations

$$C_{xy} = (1 - x)(1 - y)C_{00} + (1 - x)yC_{01} + x(1 - y)C_{10} + xyC_{11}$$

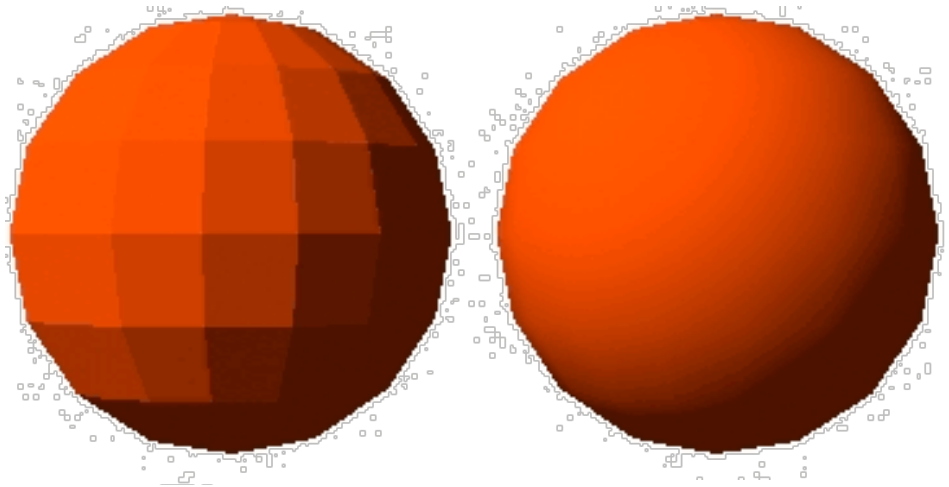


Shading in 3D

Flat shading assigns a single color per polygon

Gouraud shading interpolates between vertex colors

- a.k.a. **per-vertex shading**

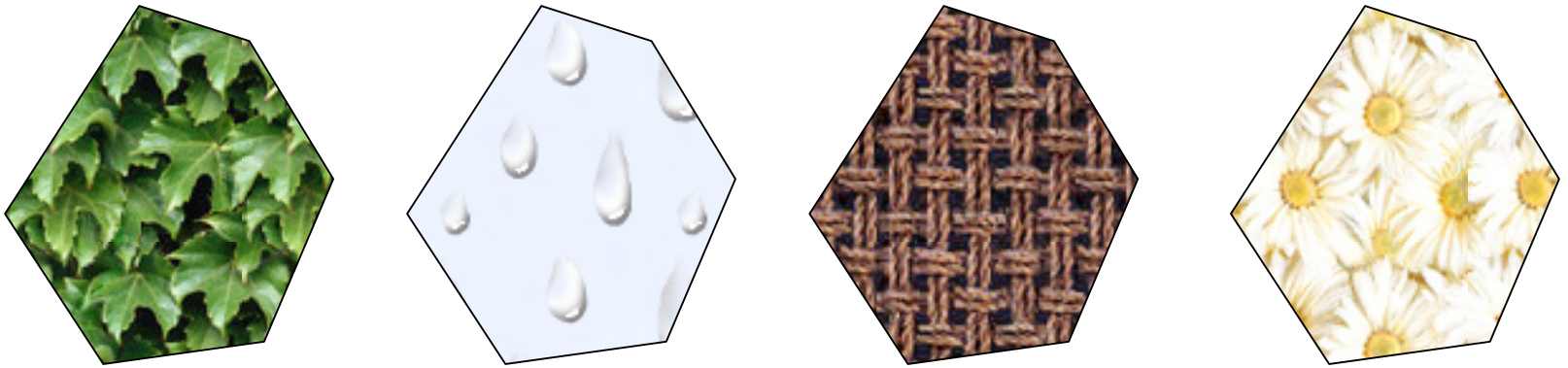


Flat shading Gouraud shading

These two spheres have the same geometry (number of polygons)

Patterned fill

Polygon is filled with a repeated pattern



Patterned fill

Polygon is filled with a repeated pattern
Pattern is defined by a small pattern tile

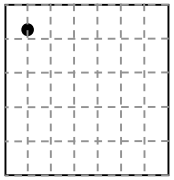


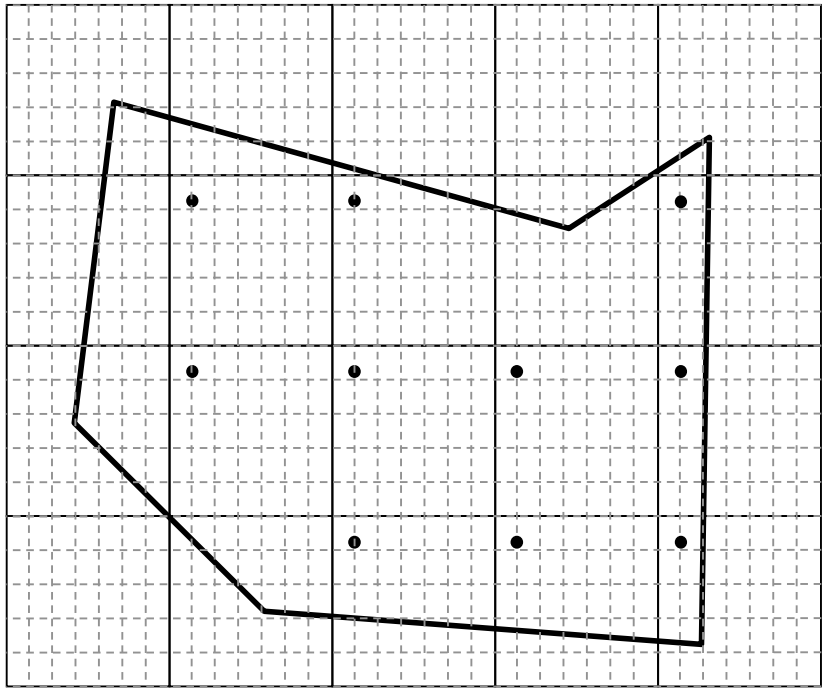

During rasterization, pixels are mapped into the tile using mod()

color[i][i] = tile[i%M, j%N]

where MxN is the size of the tile

Patterned fill



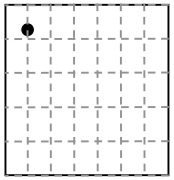


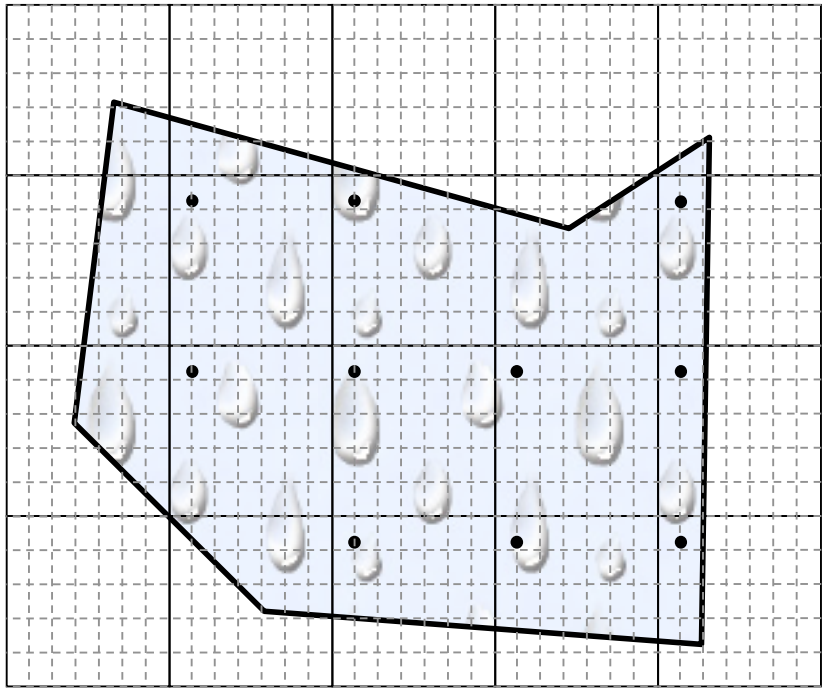
During rasterization, pixels are mapped into the tile using mod()

color[i][i] = tile[i%M, j%N]

where MxN is the size of the tile

Patterned fill





During rasterization, pixels are mapped into the tile using mod()

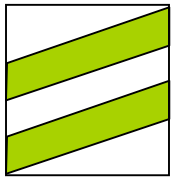
color[i][i] = tile[i%M, j%N]

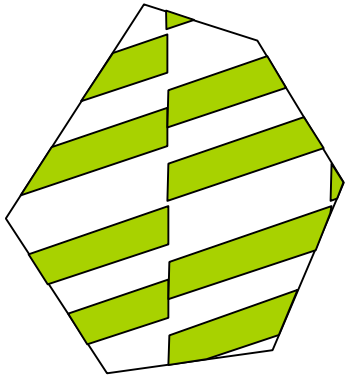
where MxN is the size of the tile

Patterned fill

If the tile wraps around, the pattern is seamless

Otherwise, tile boundaries are apparent



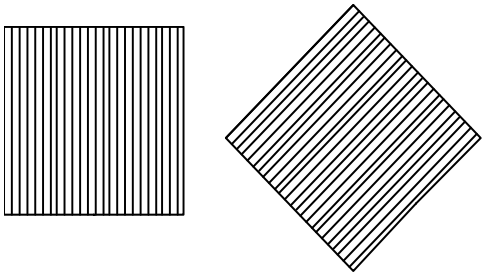


Pattern anchor

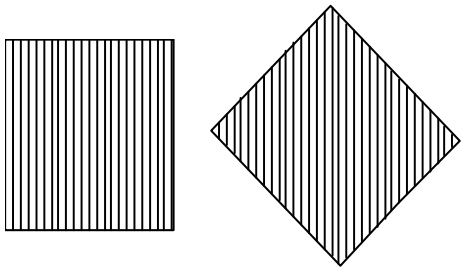
Determines the tile origin

Two common anchor points:

- Anchor with respect to the polygon
  - Pattern is fixed to the polygon



- Anchor with respect to the image origin
  - Pattern changes when the polygon moves



Limitations of pattern fill

Fixed pattern size (and sometimes orientation)

Not tightly correlated to size and orientation of polygon

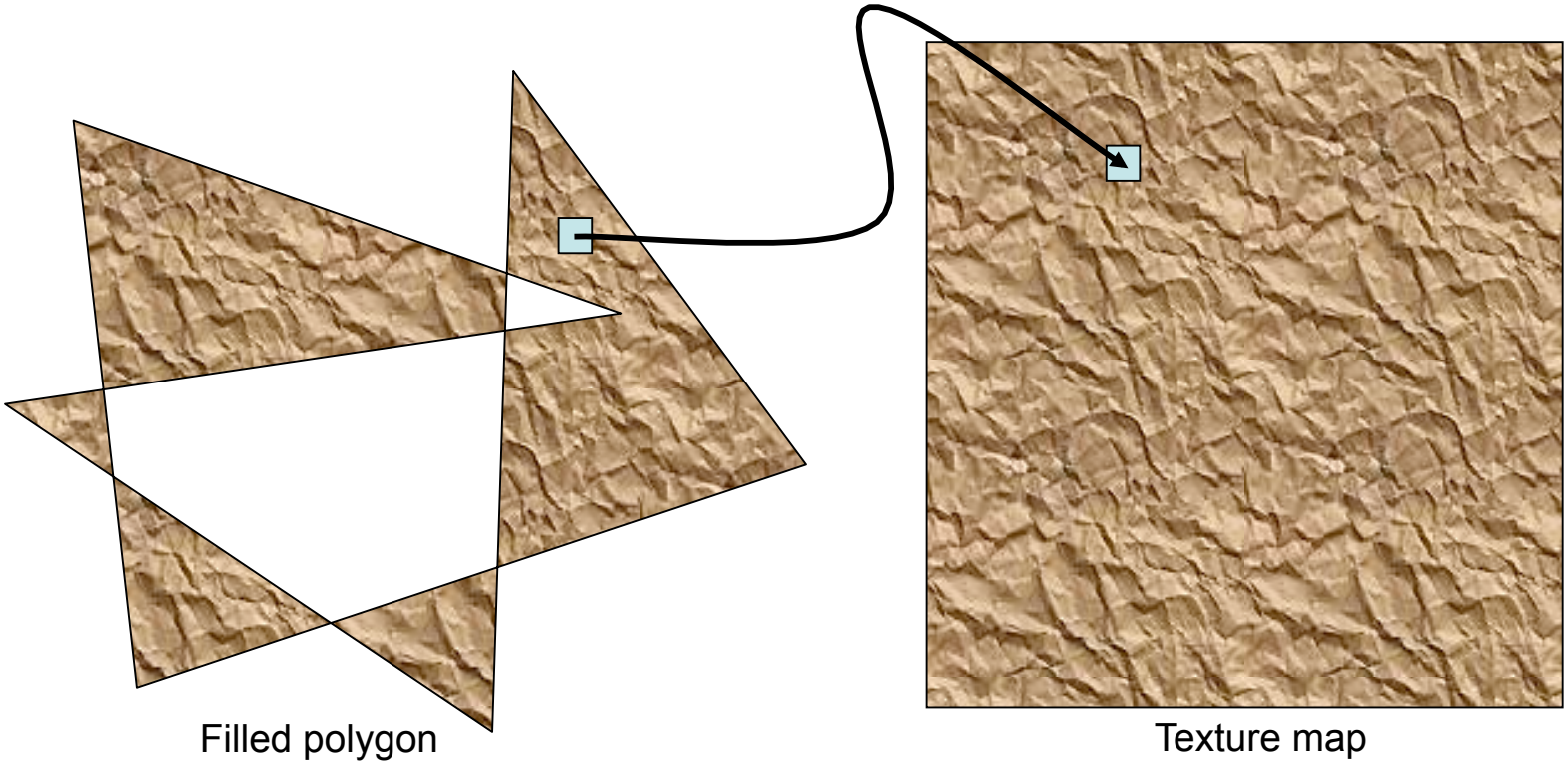
Polygon is “just a window” on the texture

Textured fill

Allows us to add much more detail

Use **texture maps** to determine pixel colors

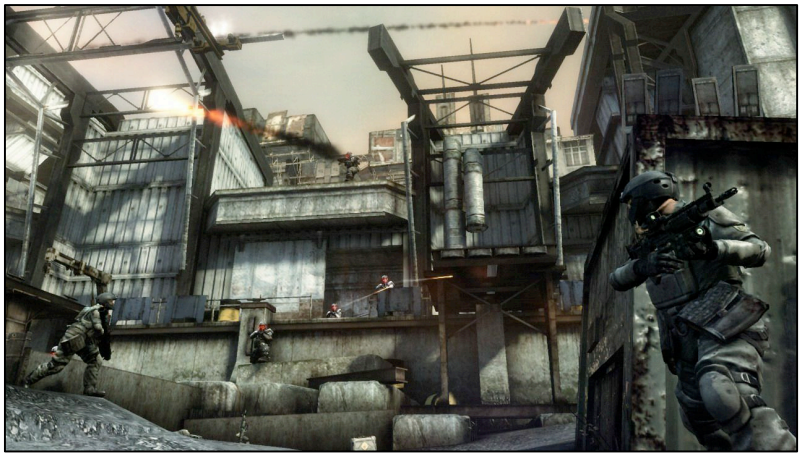
Pixels in the polygon are mapped onto the texture image to determine their color



Texture maps

Used to set the color of each point on a polygon
Give a simple polygon the appearance of something more complex

Source: Oliviera et al. 2000



Source: *Killzone 2*

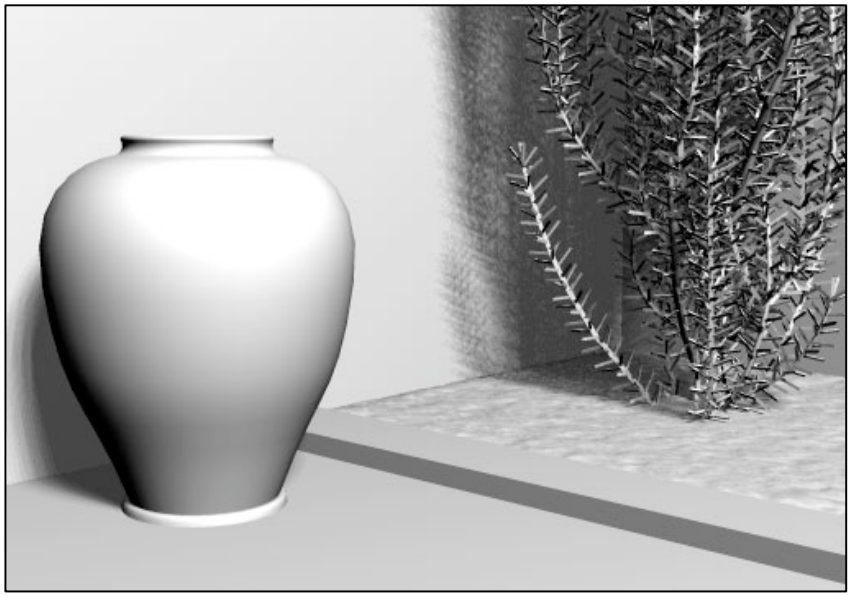
Can think of as painted paper that gets stretched over geometry
An inexpensive way to add color detail to an object

Texture maps

Many sources

- Digital photographs
- Hand painted textures
- Procedurally generated textures

Add realism to 3D scenes



3D scene geometry

3D scene with textures applied

Parametric texture mapping

Separate **screen space** and **texture space**
Texture the polygon as before, but in texture space
Deform (render) the texture into screen space

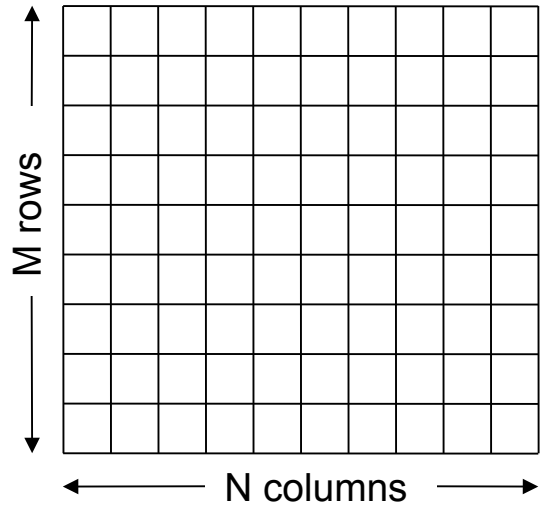
Define the texture as an image
Wrap it around a surface (defined by polygons)
Texture size and orientation are tied to the polygons
When the surface moves or deforms, the texture goes with it

Texture maps

Store an MxN color image

Data values in the texture map are called **texels** (texture elements)

Texture coordinates (u, v) ∈ [0, 1] x [0, 1]

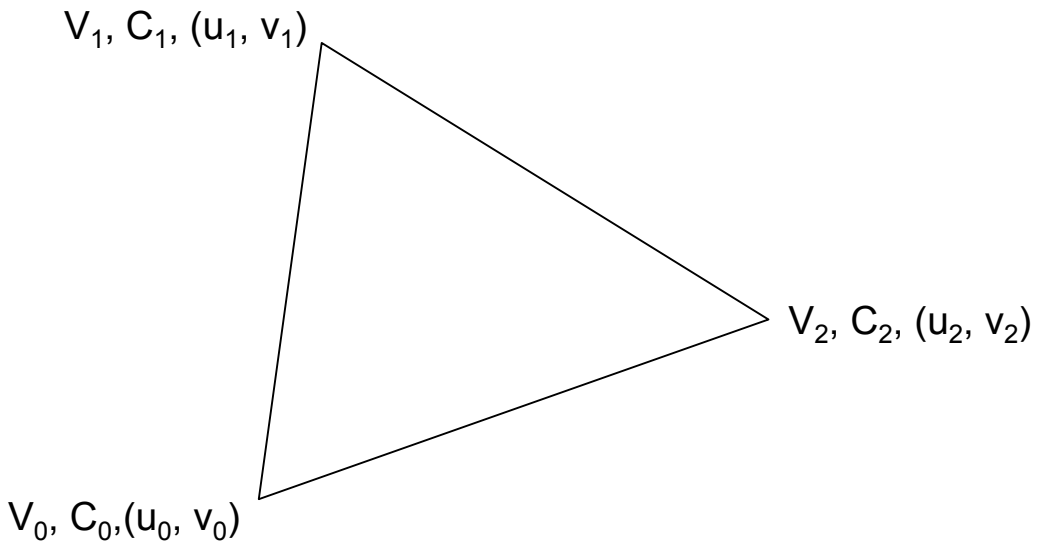


Texture coordinates

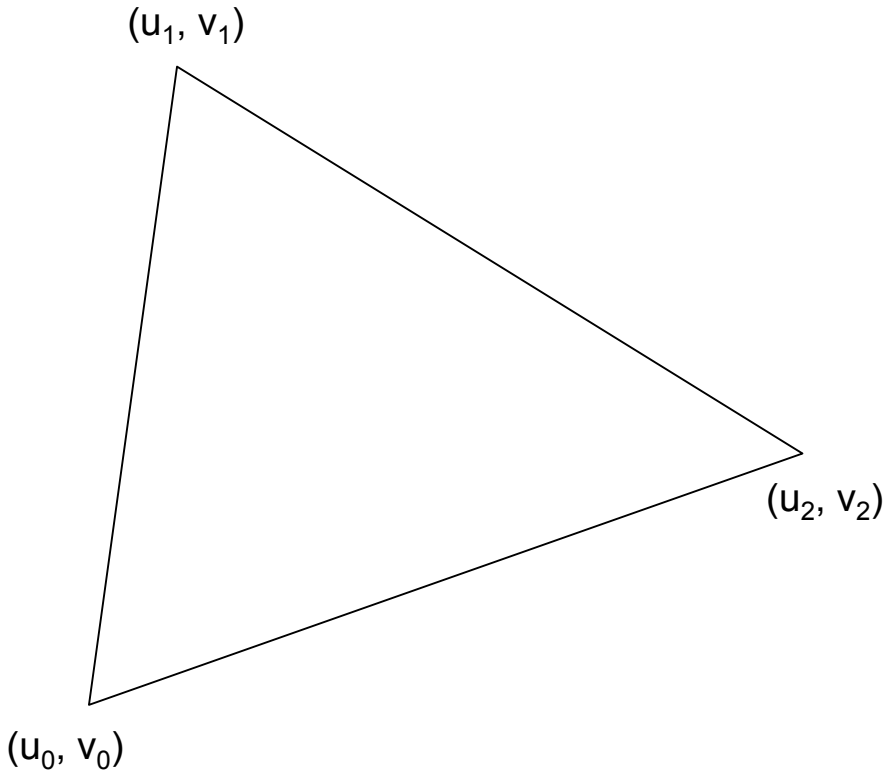
For texture mapped polygons

- Specify a pair of texture coordinates for each vertex
- e.g., for a triangle with vertices V_0, V_1, V_2 (and colors C_0, C_1, C_2)

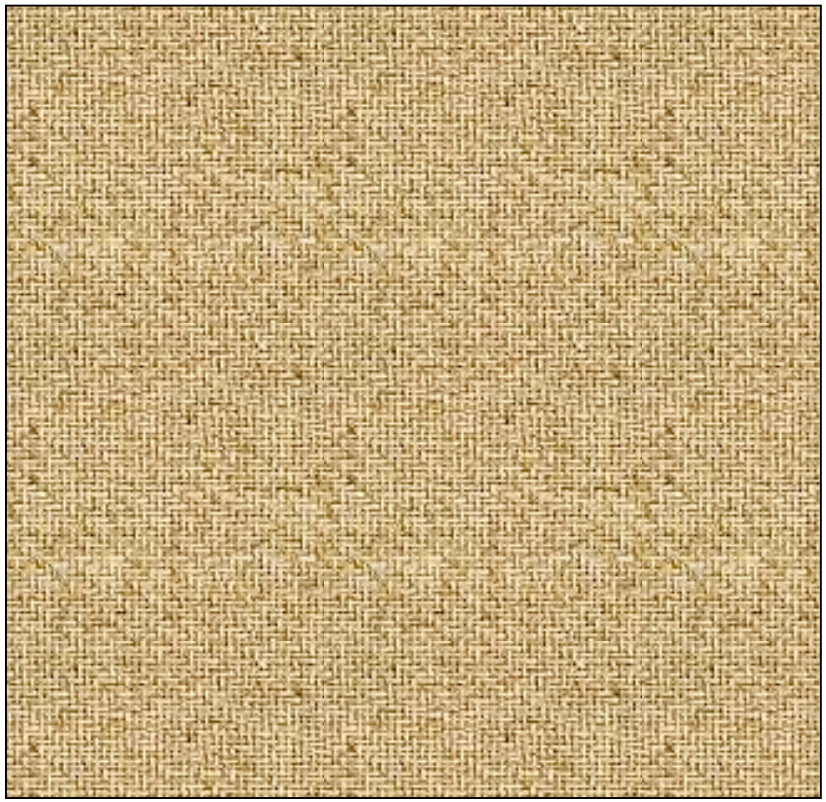
$$\text{texCoords}(V_0) = (u_0, v_0)$$
$$\text{texCoords}(V_1) = (u_1, v_1)$$
$$\text{texCoords}(V_2) = (u_2, v_2)$$



Texture coordinates

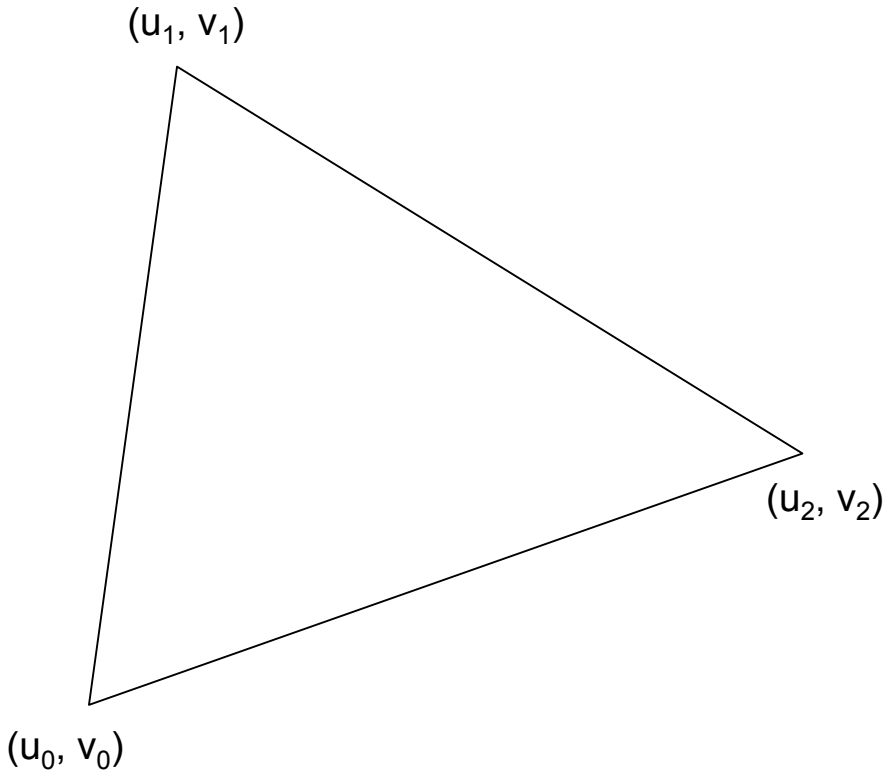


Triangle labeled with texture coordinates

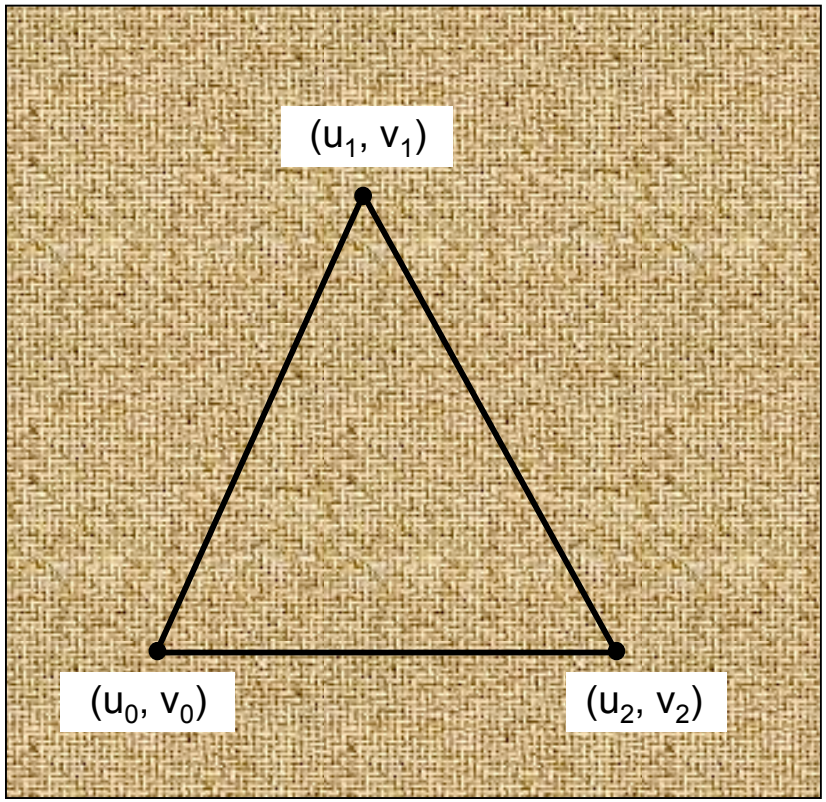


Texture image

Texture coordinates

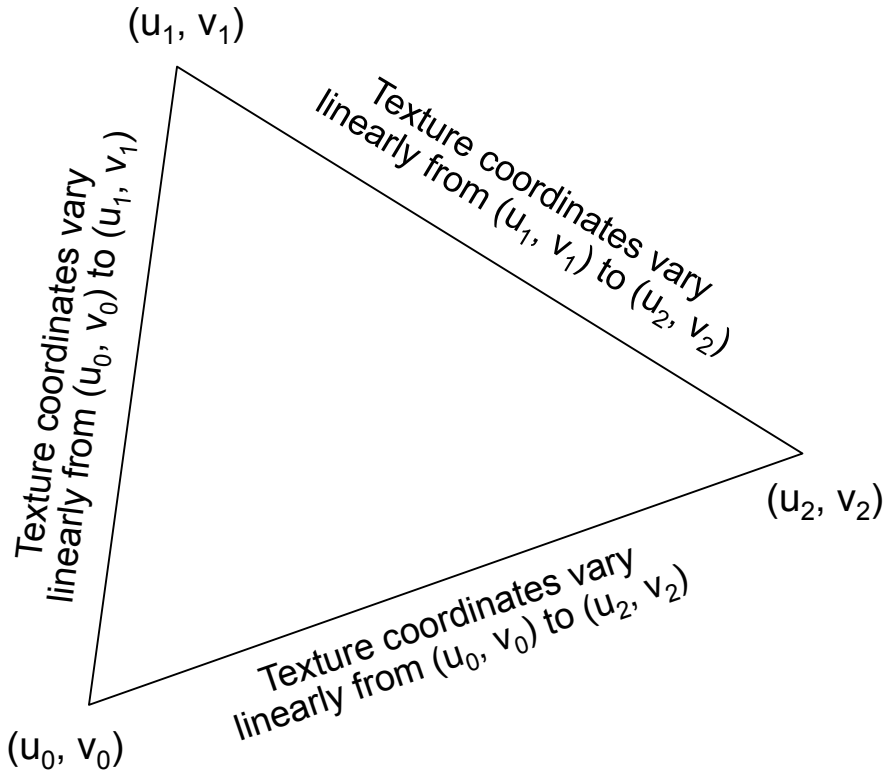


Triangle labeled with texture coordinates

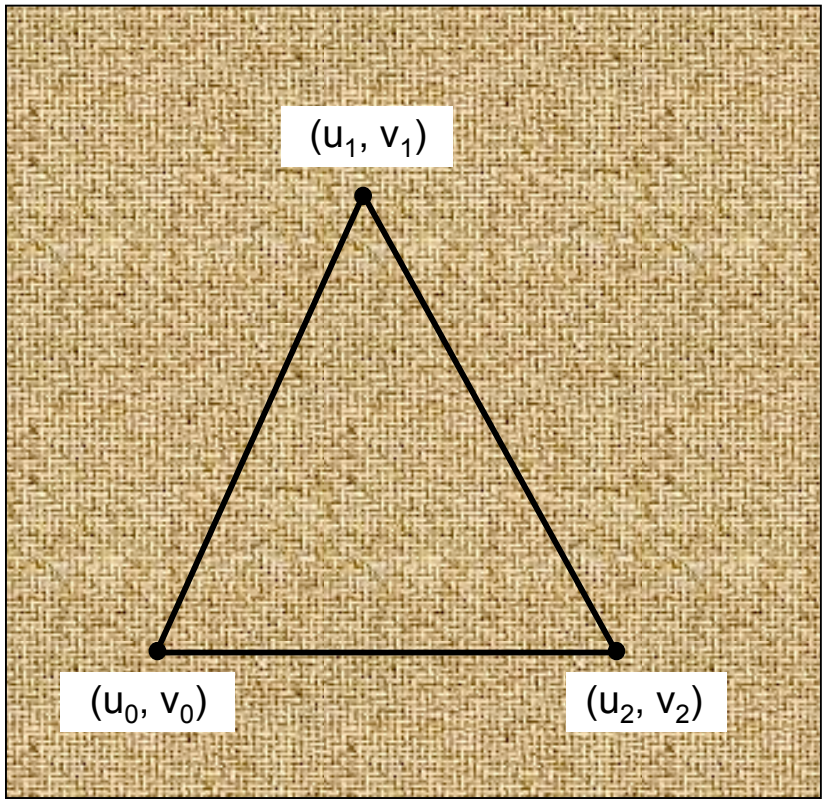


Texture image

Texture coordinates



Triangle labeled with texture coordinates

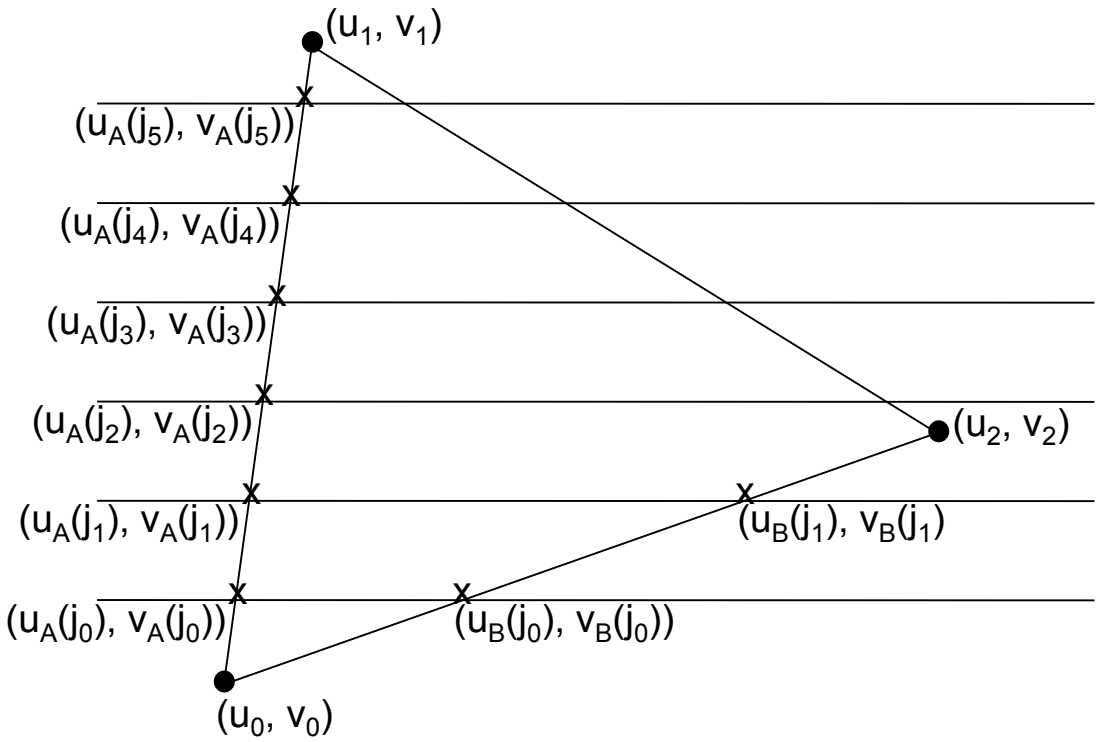


Texture image

Interpolating texture coordinates

To determine the texture color for pixel[i][j]:

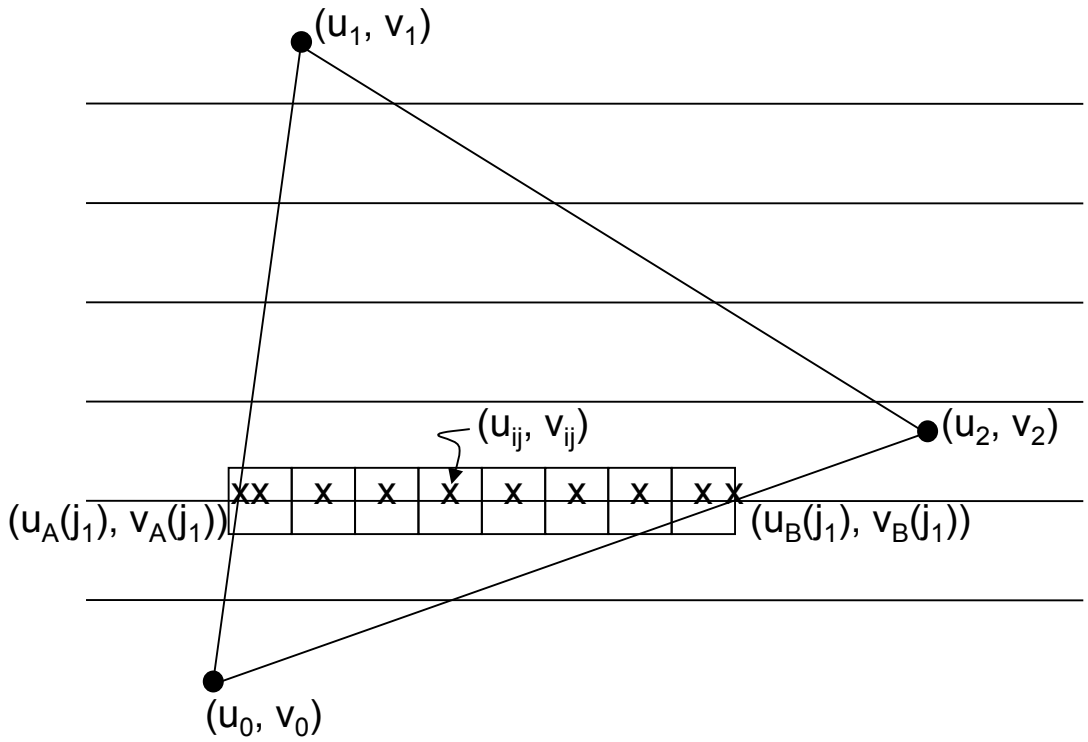
1. Interpolate vertex texture coordinates along each edge



Interpolating texture coordinates

To determine the texture color for pixel[i][j]:

1. Interpolate vertex texture coordinates along each edge
2. Interpolate vertex texture coordinates across spans



Interpolating texture coordinates

To determine the texture color for pixel[i][j]

Use the same linear interpolation that we used for interpolating color along edges and between spans

Along edges:

$$(u(j), v(j)) = (1 - t)(u_0, v_0) + t(u_1, v_1)$$

where $t = (j - y_0) / (y_1 - y_0)$

Along spans:

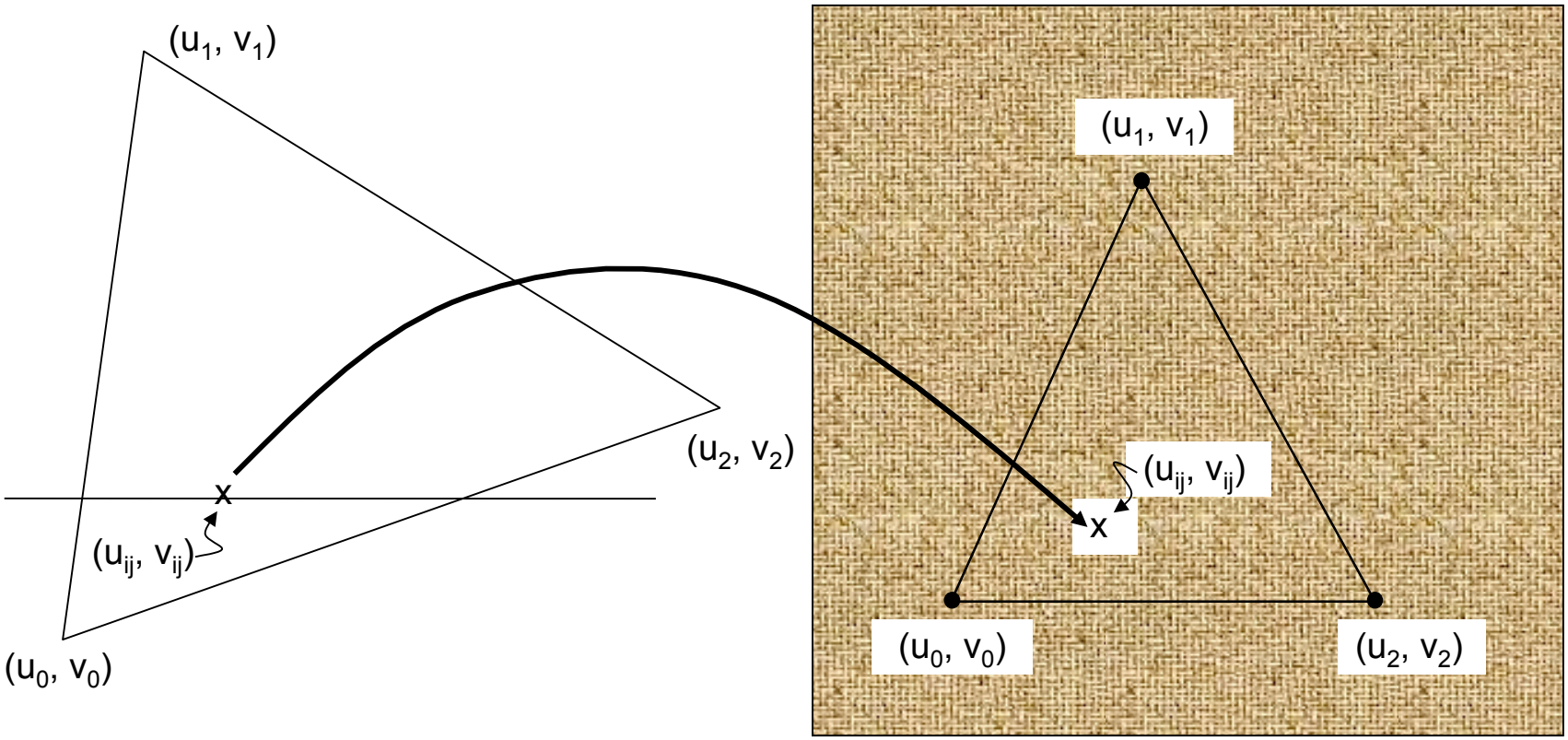
$$(u(i, j), v(i, j)) = (1 - s)(u_A(j), v_A(j))) + s(u_B(j), v_B(j))$$

where $s = (i - x_A(j)) / (x_B(j) - x_A(j))$

Interpolating texture coordinates

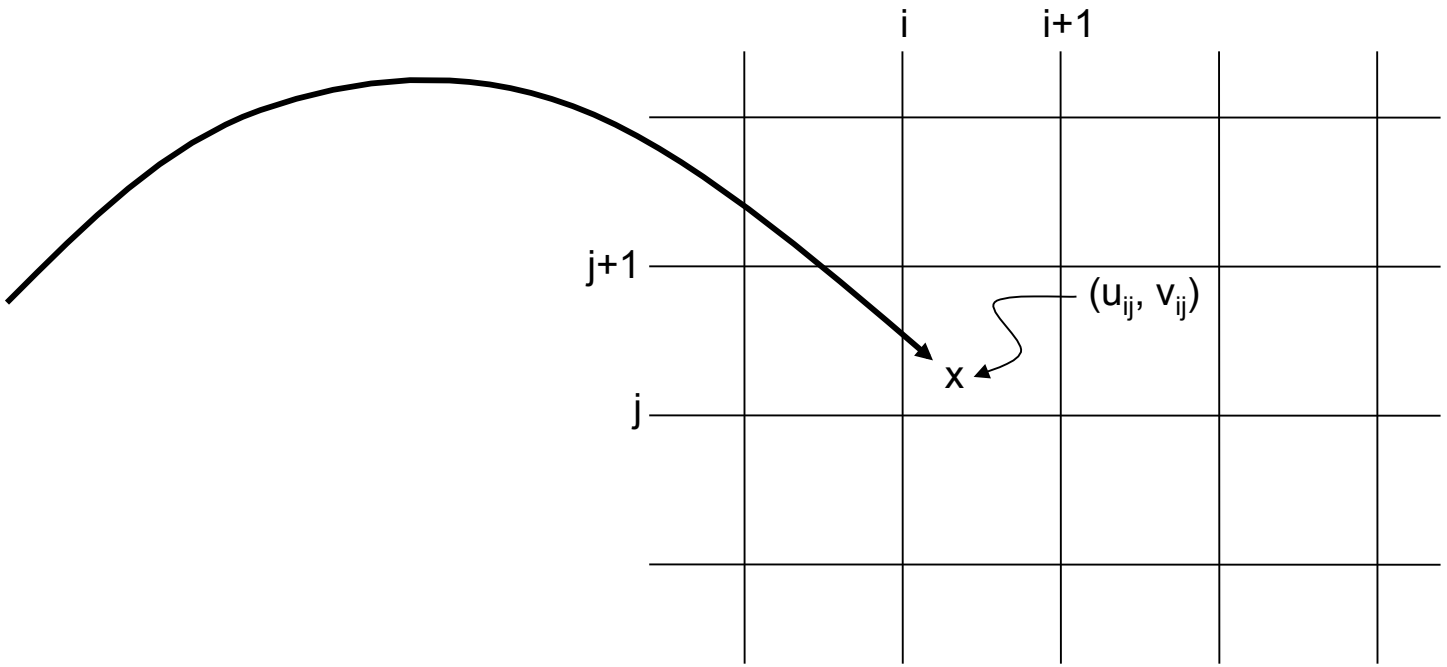
To determine the texture color for pixel[i][j]

3. Map the interpolated texture coordinates into the texture map
4. Determine the texture color from the texture map



Interpolating texel colors

What do you do when the texture sample you want does not lie on an integer texel location?

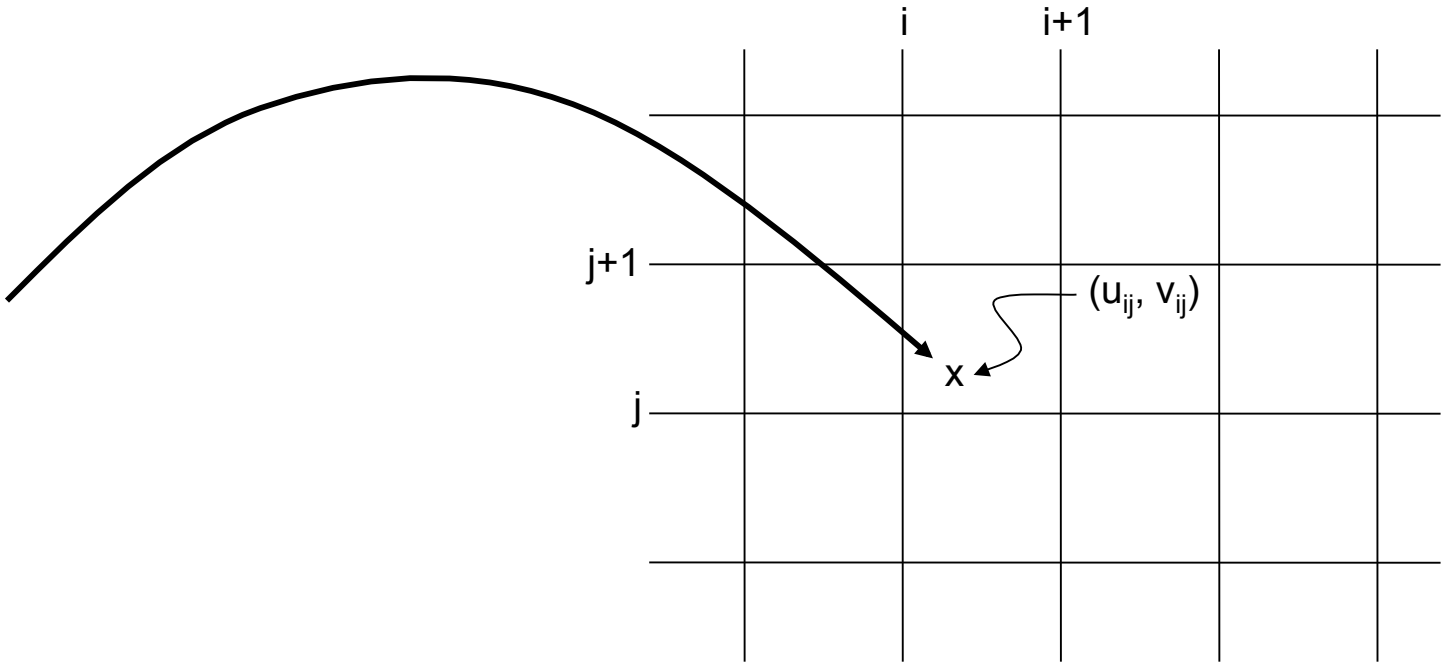


Texture map interpolation

Need to interpolate neighboring texels to determine the color of the sample point

i.e. Need to **resample** the texture

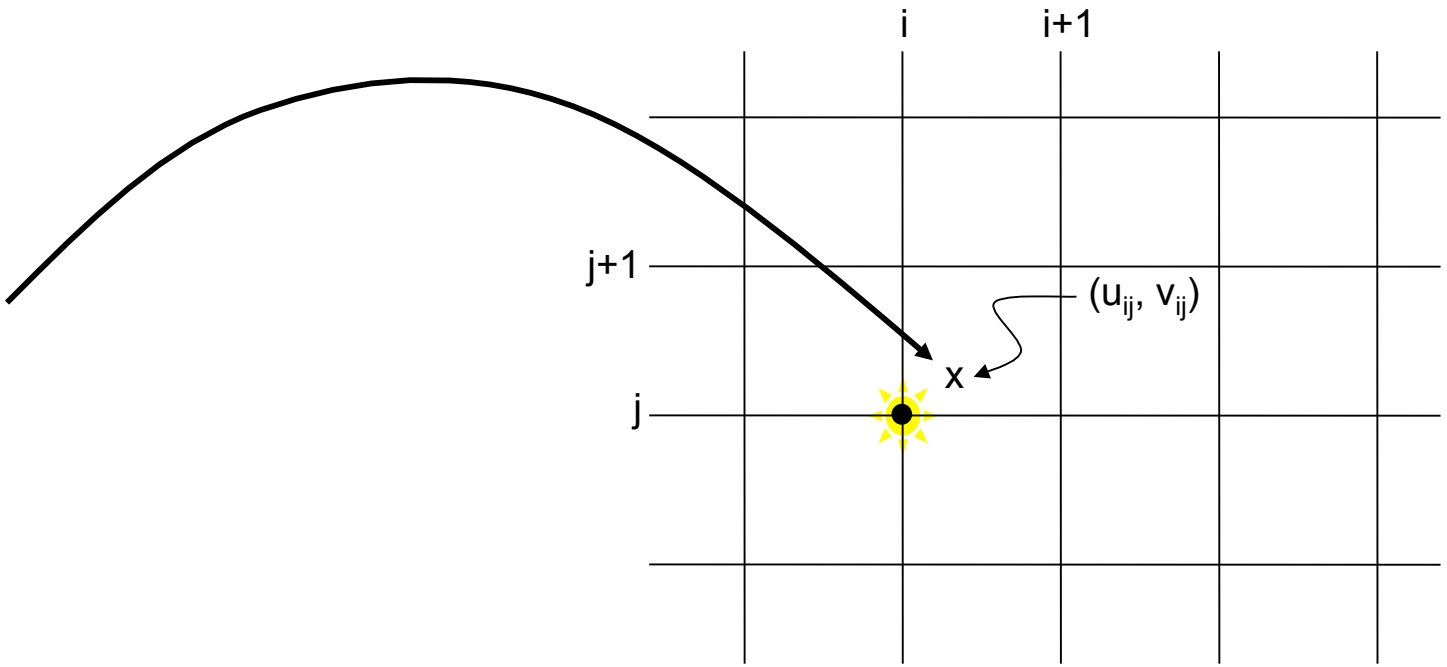
Strategies?



Texture map interpolation: Nearest neighbor

Choose the texel closest to the given texture coordinates

Multiply the texture coordinates by the texture map width and height and round to the nearest integer



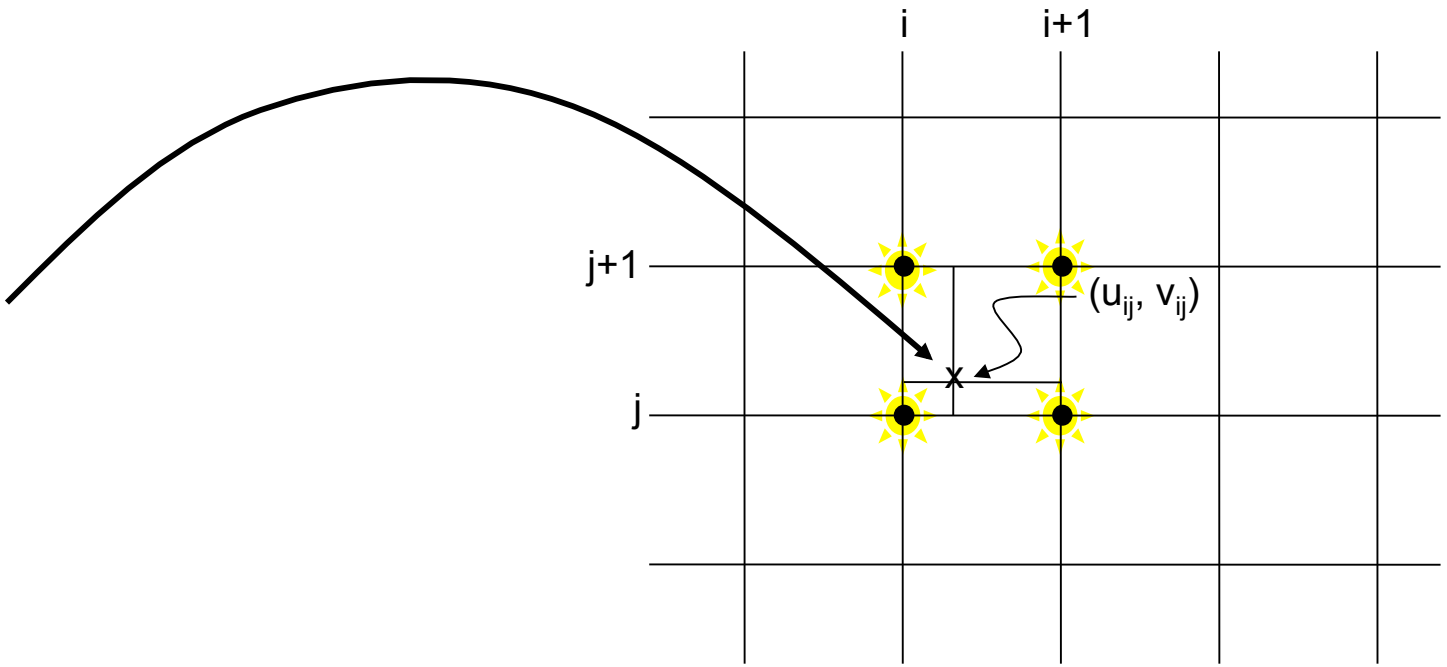
Texture map interpolation
Bilinear resampling

Look at the four closest texels

$dx = (u - i/M)$

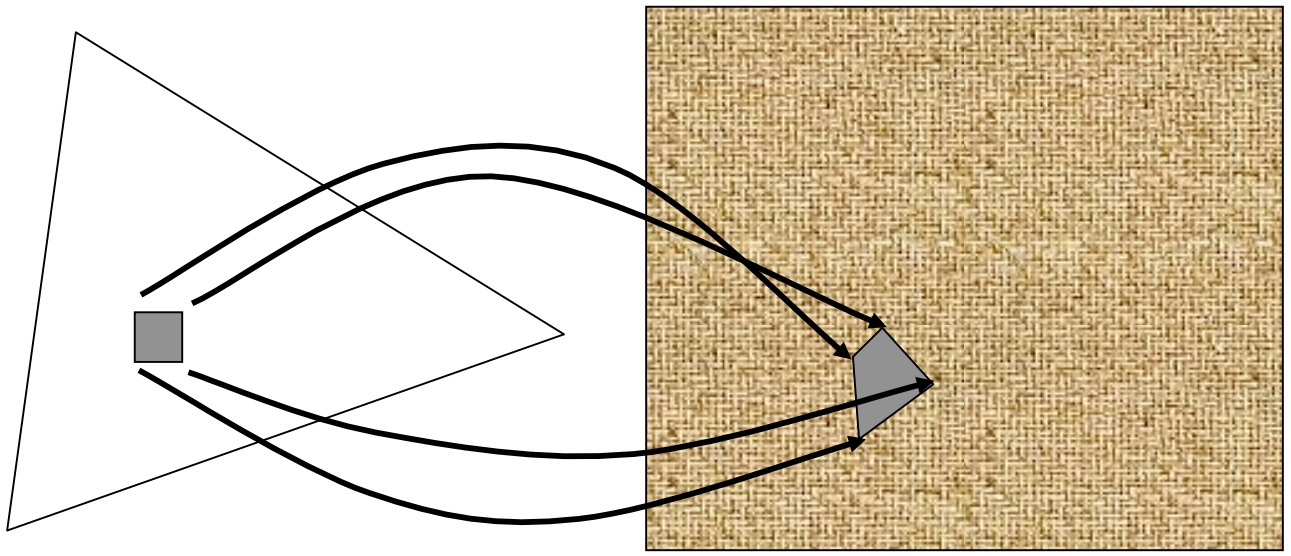
$dy = (v - j/N)$

$C_{uv} = (1-dx)(1-dy)C_{i,j} + (1-dx)dyC_{i,j+1} + dx(1-dy)C_{i+1,j} + dxdyC_{i+1,j+1}$



Texture map interpolation
Texture pre-filtering

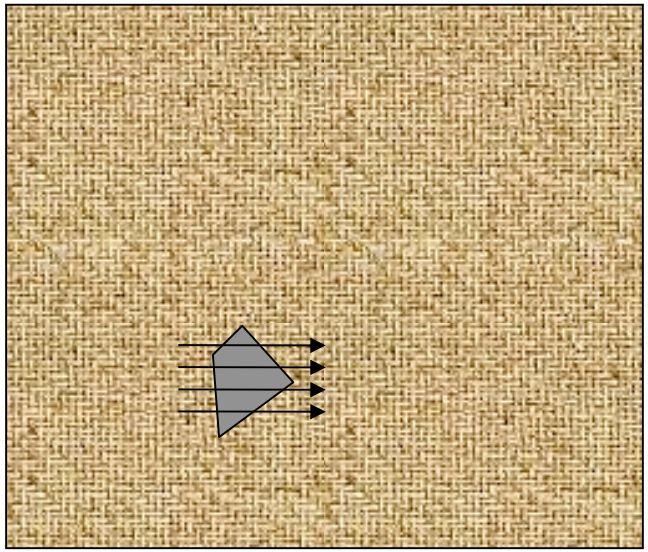
Map the boundaries of the image pixel into texture coordinates
Average the texture colors inside the mapped pixel boundaries
Expensive but reduces sampling artifacts



Texture map interpolation
Texture pre-filtering

To determine average texture color inside mapped pixel boundaries:

1. Rasterize the mapped boundary polygon
2. Average the interior colors



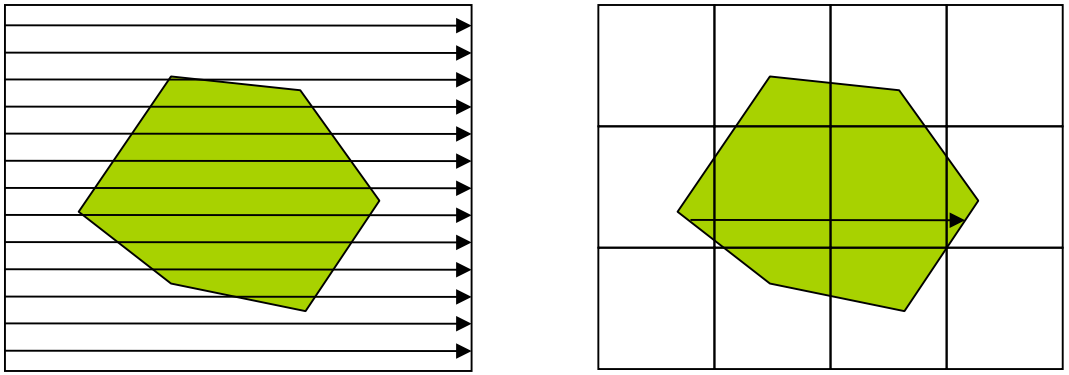
Raster-based filling

Last time: x-intersection array and edge lists

- Find limits of interior spans and then fill the pixels between the limits
- Scan-line serial

Graphics hardware uses tiled frame buffers

One interior span of a polygon may cross multiple tiles: inefficient memory access



Pineda's parallel rasterization algorithm

An inside/outside test for any pixel

A method to compute linear interpolation coefficients for any pixel

Doesn't need to

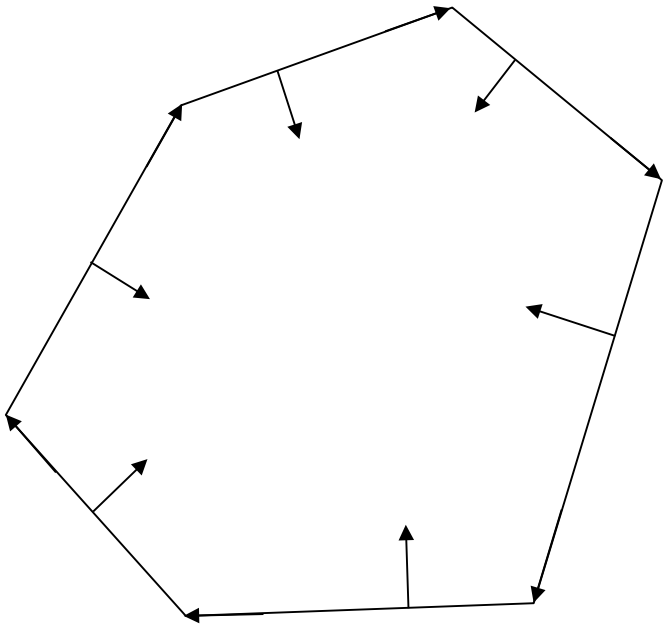
- Compute x-intersections of the span
- Compute colors or texture coordinates at the x-intersection points

Appropriate for filling tiled frame buffers using parallel processors

- Suitable for modern graphics hardware

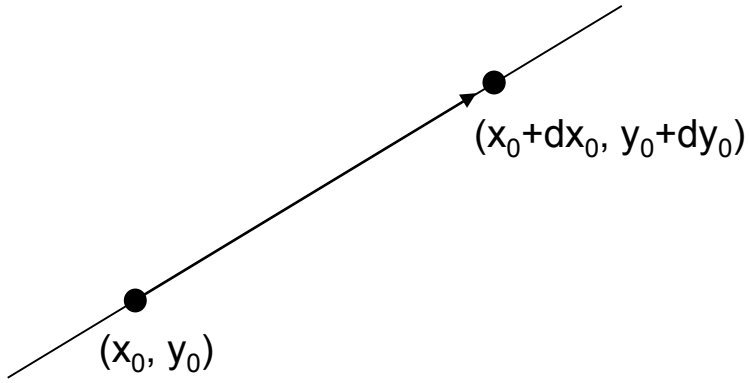
Pineda's inside/outside test

Assumes a polygon with directed edges
Interior points are on the same side of all of the edges



Pineda's inside/outside test

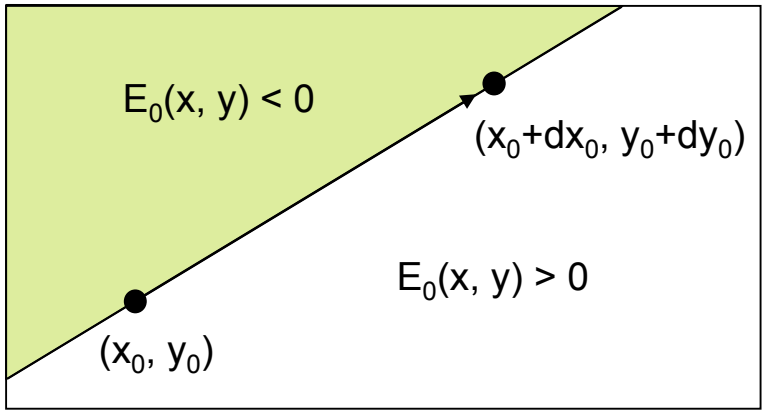
Consider a directed line from (x_0, y_0) to (x_0+dx_0, y_0+dy_0)



Pineda's inside/outside test

Define an edge function for an arbitrary point (x, y)

$$E_0(x,y) = (x - x_0)dy - (y - y_0)dx$$



Properties of the edge function:

$E_0(x,y) >$ for (x, y) to the right of the line

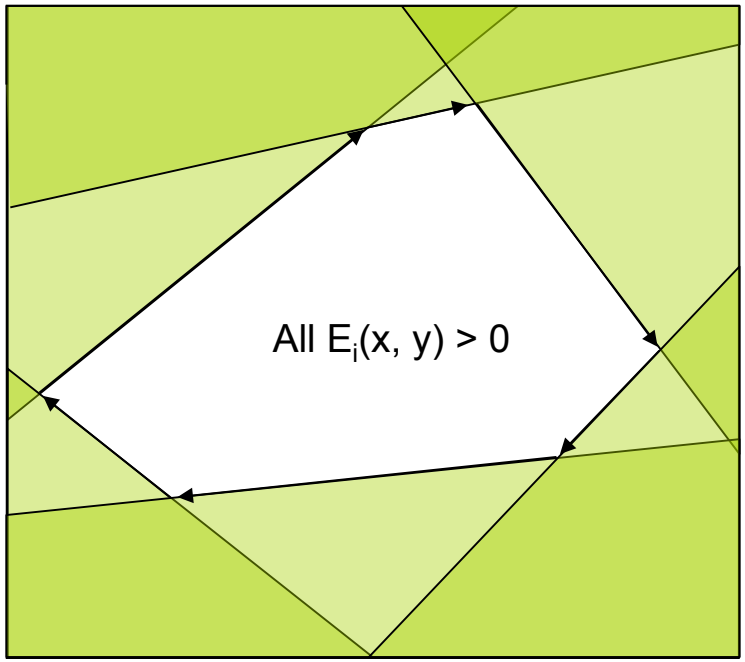
$E_0(x,y) =$ for (x, y) on the line

$E_0(x,y) <$ for (x, y) to the left of the line

Pineda's inside/outside test

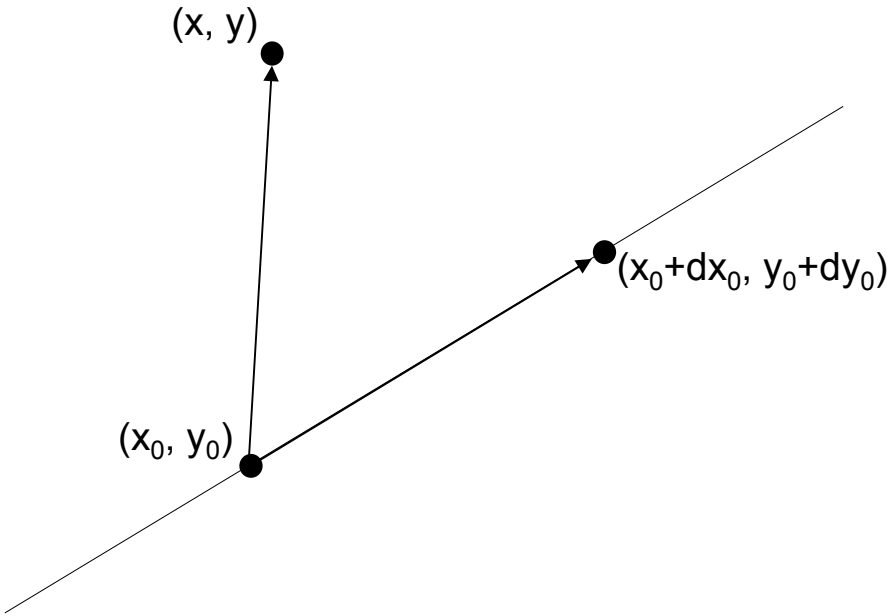
For each pixel, evaluate all of the edge functions $E_i(x, y)$

(x, y) is inside the polygon if every edge function is positive at (x, y)



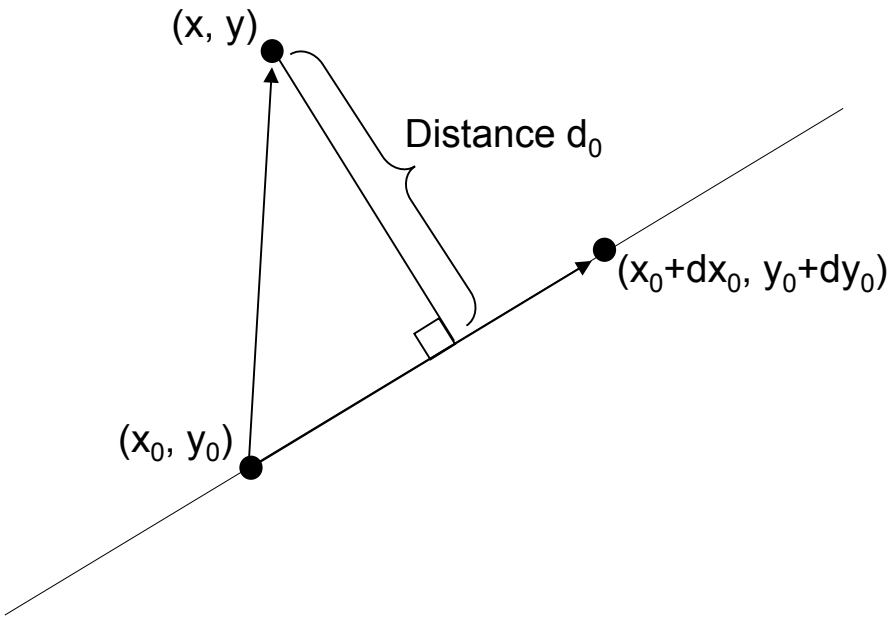
Geometric interpretation of E(x, y)

E_0 (x, y) is the cross product of the vector from (x_0, y_0) to (x, y) and the vector from (x_0, y_0) to (x_0+dx, y_0+dy)



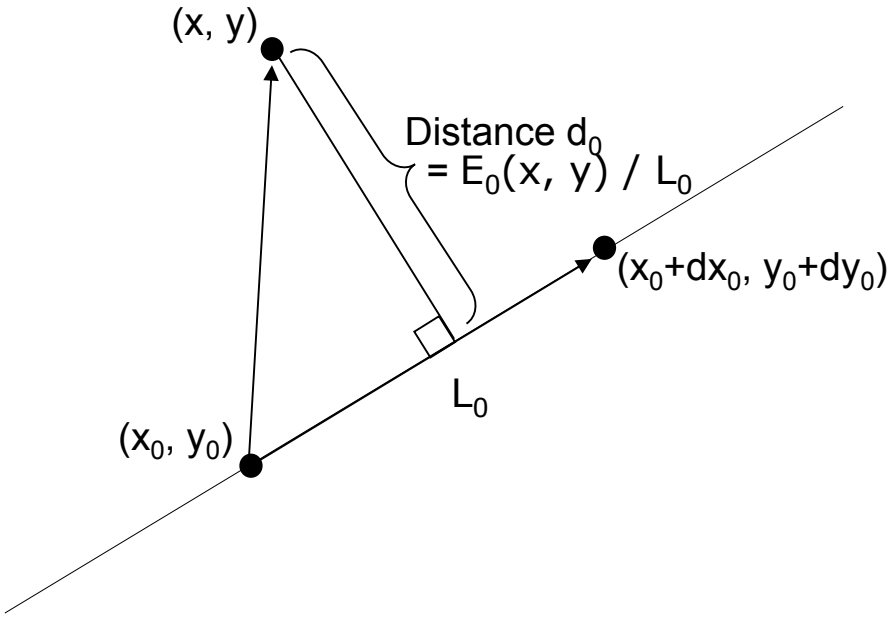
Geometric interpretation of E(x, y)

The cross product is proportional to the distance from (x, y) to the line defined by (x_0, y_0) and (x_0+dx, y_0+dy)



Geometric interpretation of E(x, y)

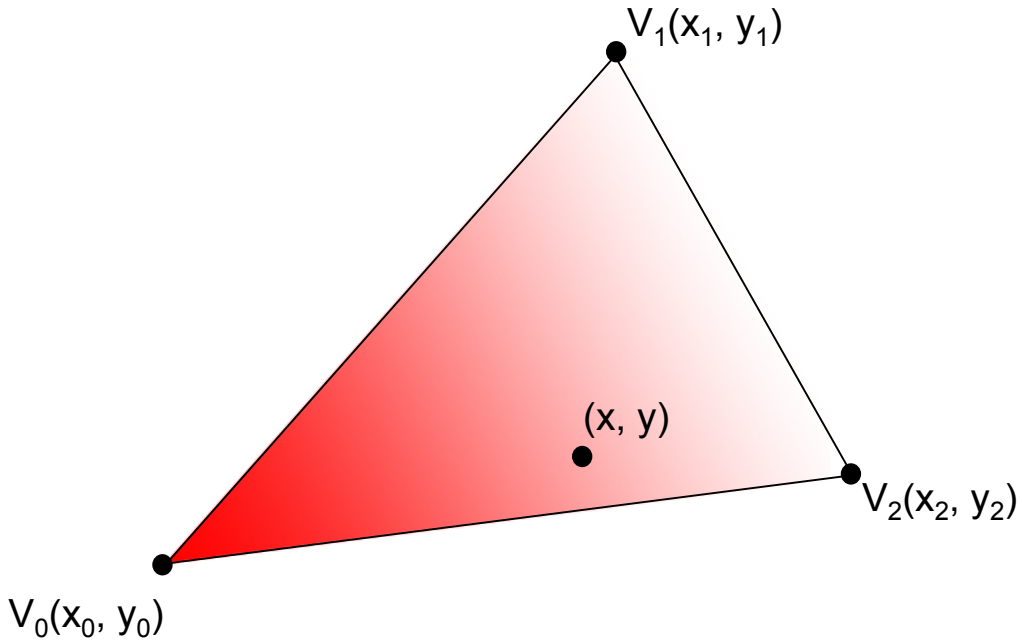
The proportionality constant is the length, L_0, of the line from (x_0, y_0) to (x_0+dx, y_0+dy)



Pineda's triangle filling

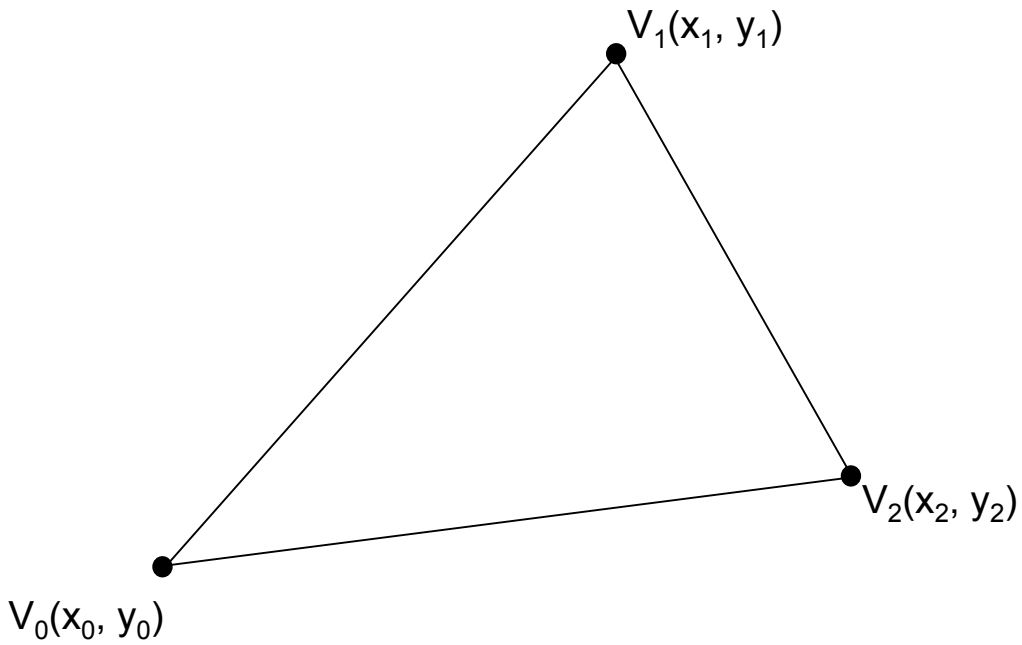
When all the edge functions are positive, (x, y) is inside the polygon

$$E_0(x, y) = (x - x_0)(y_1 - y_0) - (y - y_0)(x_1 - x_0)$$
$$E_1(x, y) = (x - x_1)(y_2 - y_1) - (y - y_1)(x_2 - x_1)$$
$$E_2(x, y) = (x - x_2)(y_0 - y_2) - (y - y_2)(x_0 - x_2)$$



Pineda's triangle filling

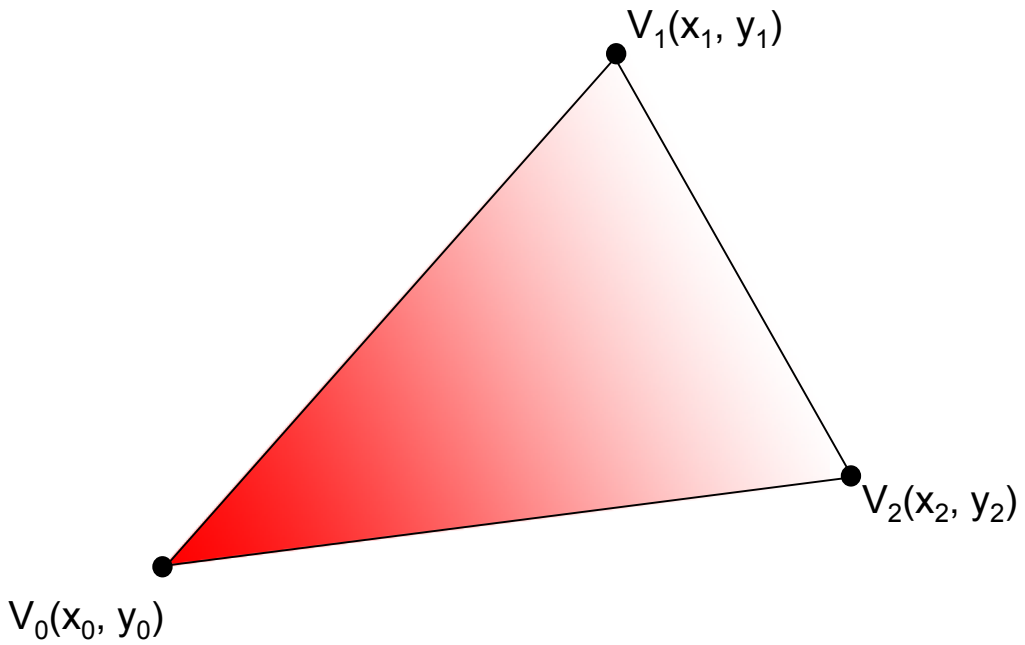
How do we interpolate (colors, texture coordinates, etc.) across the face of the triangle?



Pineda's triangle filling

The influence of the color from vertex V_0…

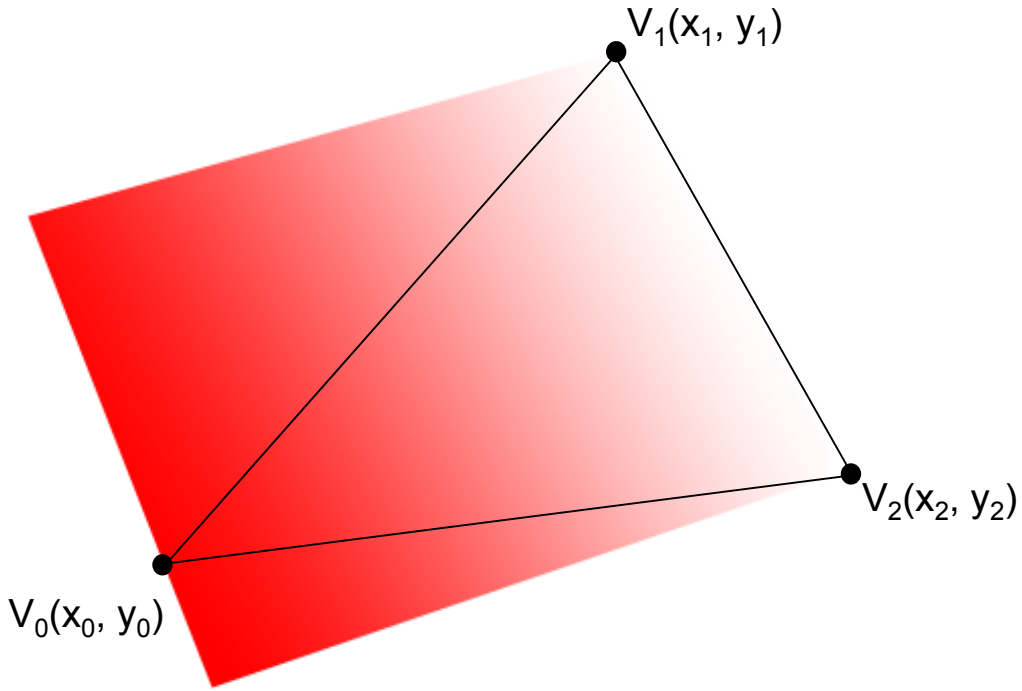
- is 1 at V_0
- is 0 at the edge opposite V_0
- varies linearly in between



Pineda's triangle filling

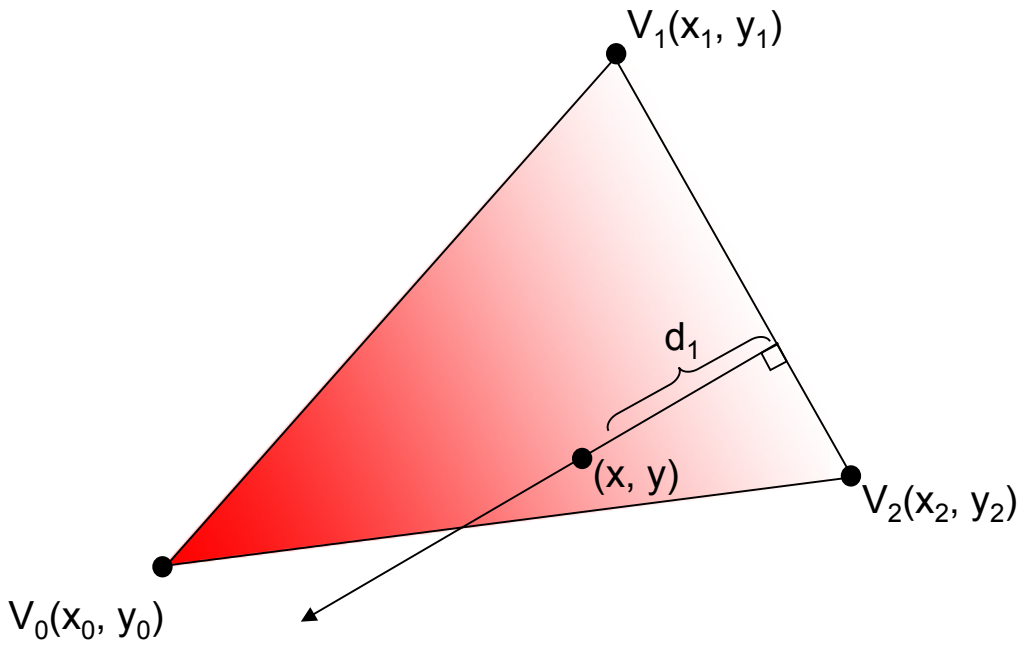
Think of the color influence varying linearly in the whole plane containing the polygon

The color gradient changes fastest in the direction perpendicular to the edge opposite V_0



Pineda's triangle filling

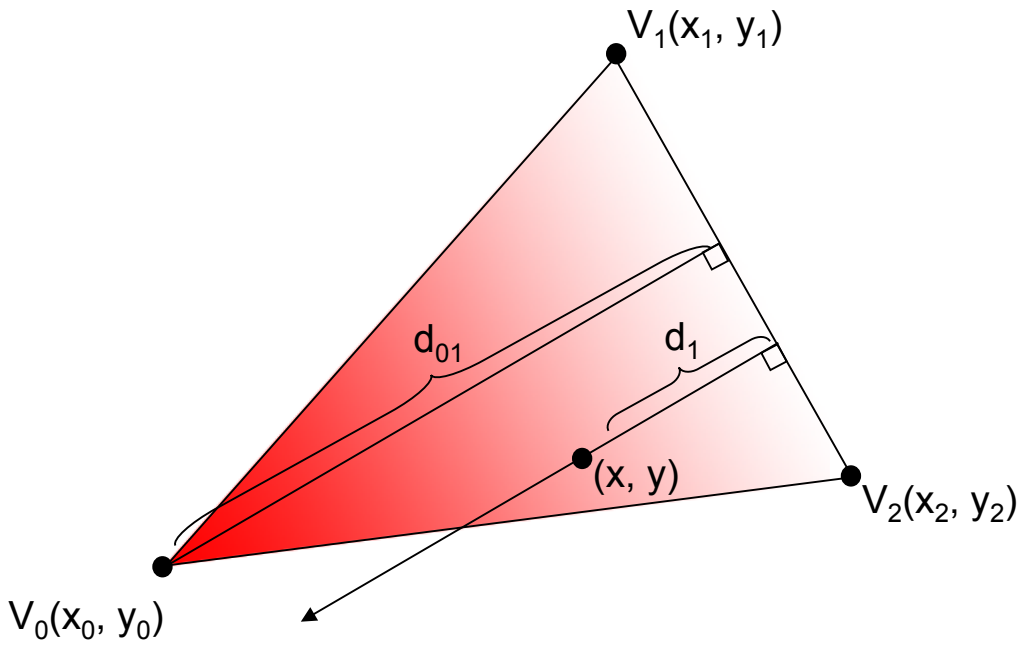
The distance d_1, from the edge opposite V_0 also increases fastest in the direction perpendicular to the edge opposite V_0



Pineda's triangle filling

The quotient d_1 / d_{01} varies linearly from 0 and 1

Use this as the color/texture interpolation coefficient



Pineda's triangle filling

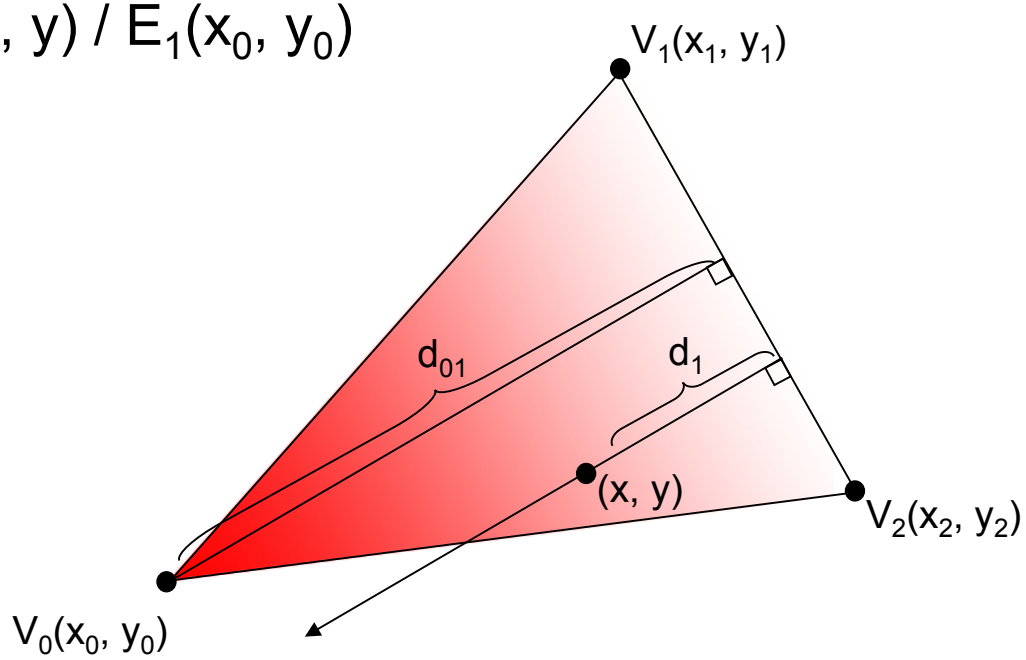
Recall

$d_1 = E_1(x, y) / L_1$

$d_{01} = E_1(x_0, y_0) / L_1$

Thus,

$d_1/d_{01} = E_1(x, y) / E_1(x_0, y_0)$



Pineda's Parallel Algorithm: Summary

For each pixel (x, y):

1. Compute the normalized edge functions

$$F_0(x, y) = E_0(x, y) / E_0(x_2, y_2)$$
$$F_1(x, y) = E_1(x, y) / E_1(x_0, y_0)$$
$$F_2(x, y) = E_2(x, y) / E_2(x_1, y_1)$$

2. Perform the inside/outside text: (x, y) is inside if

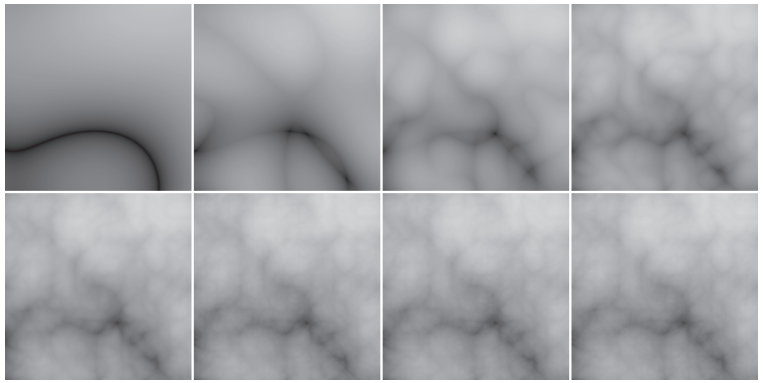
$$F_0(x, y) > 0 \quad \text{and} \quad F_1(x, y) > 0 \quad \text{and} \quad F_2(x, y) > 0$$

3. If (x, y) is inside, interpolate the pixel color (or texture coordinates) from the vertex colors

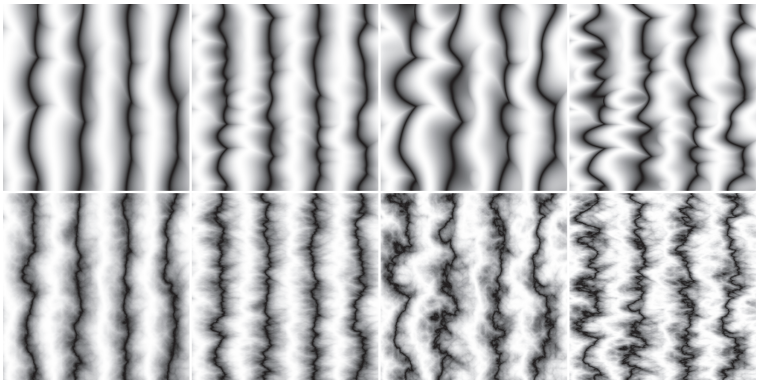
$$C_{xy} = F_0(x, y)*C_2 + F_1(x, y)*C_0 + F_2(x, y)*C_1$$

Procedural textures

Generate “structured noise” and turbulence for natural-looking textures



Sine pattern



Sine pattern with a turbulent noise function

What artifacts might you see from using a marble veneer instead of solid marble?

Solid textures

3D representation of non-homogeneous material (e.g. marble, wood)

Use **model space** coordinates to index into a **3D texture**

- Map each surface point to texture space

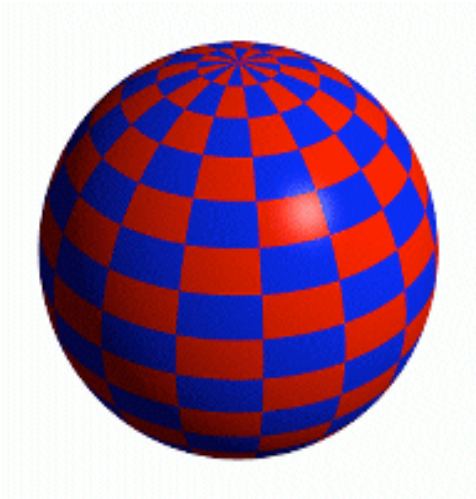
Difficulty is authoring the source texture

3D geometry rendered with procedural marble texture

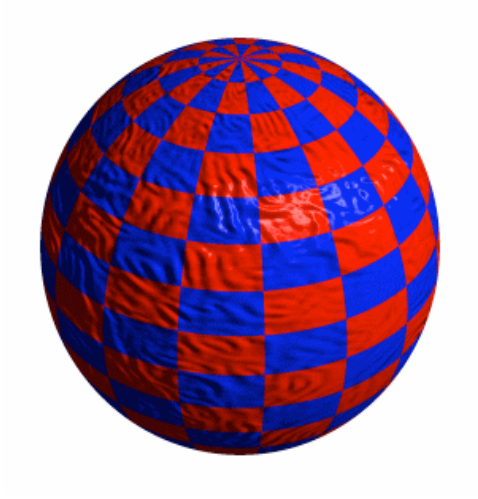
Texture maps

Can be used to modify

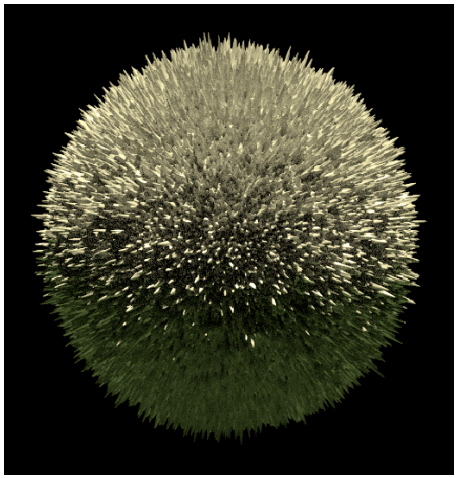
- Surface color
- Geometry (displacement mapping)
- Surface normals (bump mapping)



Texture map



Texture map + bump map

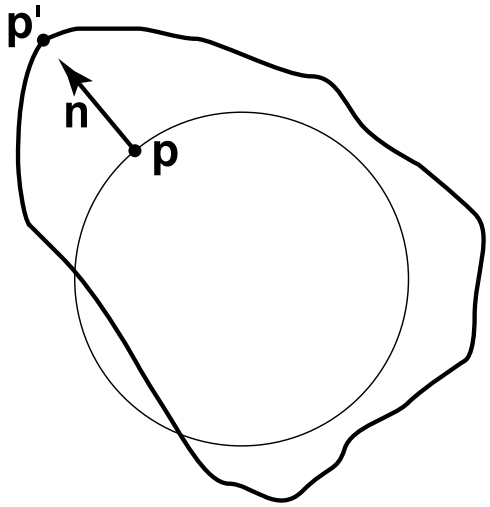


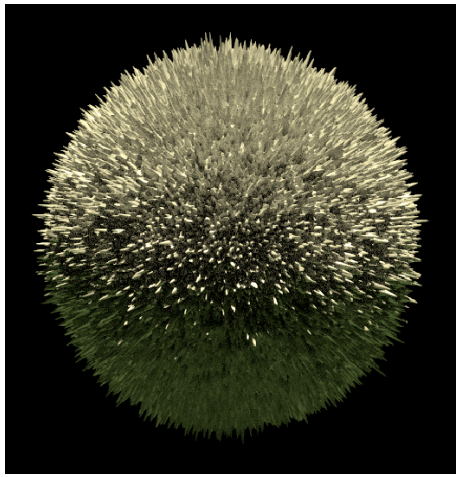
Displacement map

Displacement maps

Texture is used to perturb the surface geometry

When the surface moves or deforms, the displacements change

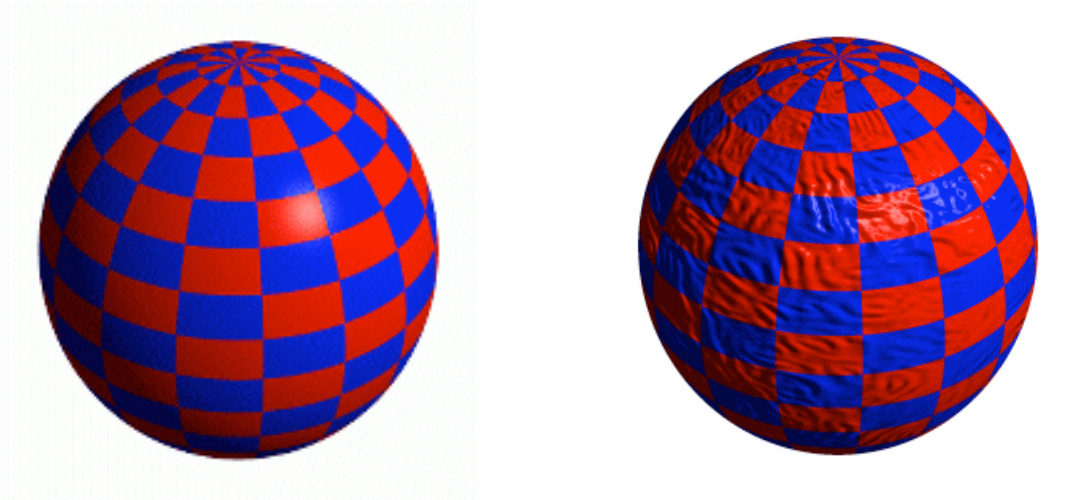




Bump maps

Instead of putting geometry there, can just specify surface normals

Bump maps are textures that perturb the normals
Use simpler geometry (the original model)
Use the normals from the displacement map for shading



What artifacts might reveal that bump mapping is fake geometry?

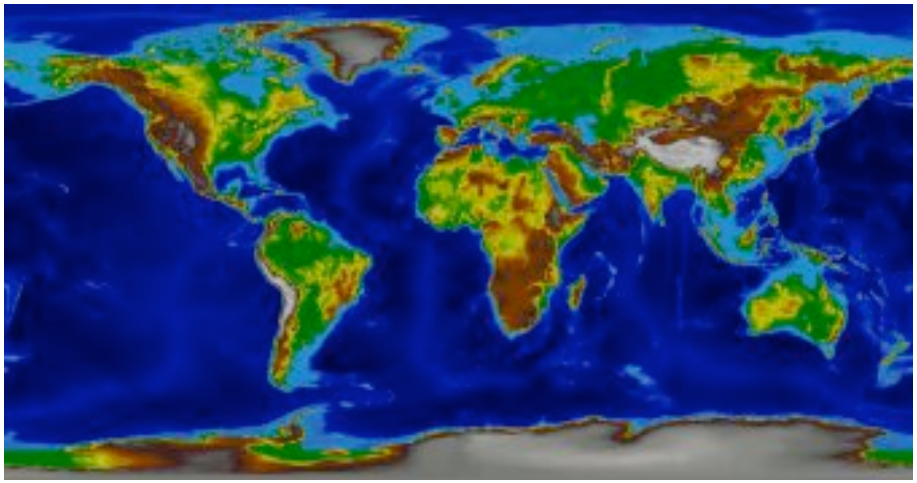
Bump maps

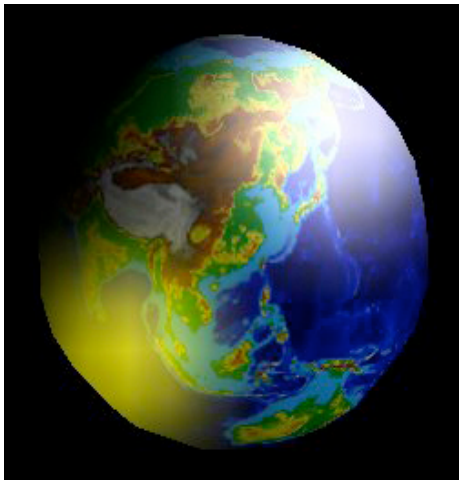
Texture map sets how each point is *colored*

Bump map sets how each point is *shaded*

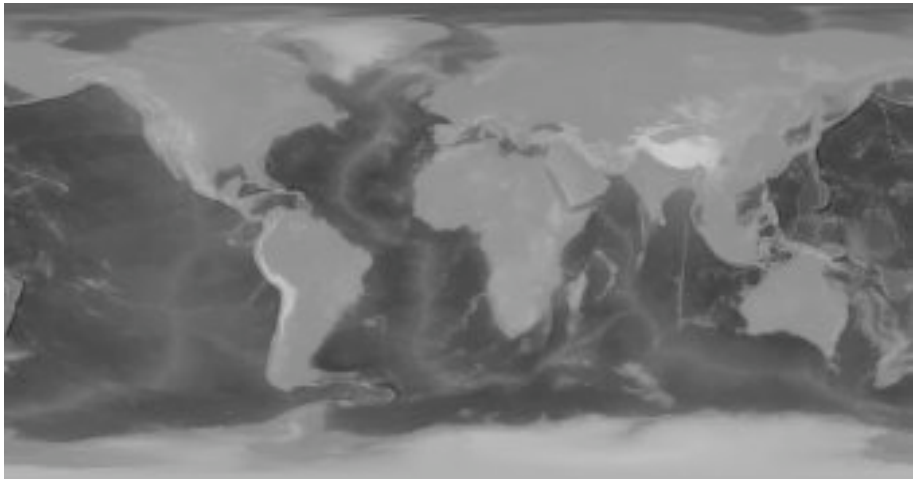
- Affects how light is reflected off of the object's surface
- A pseudo-height field that is applied to the surface
- An inexpensive way to add *geometric* detail to an object

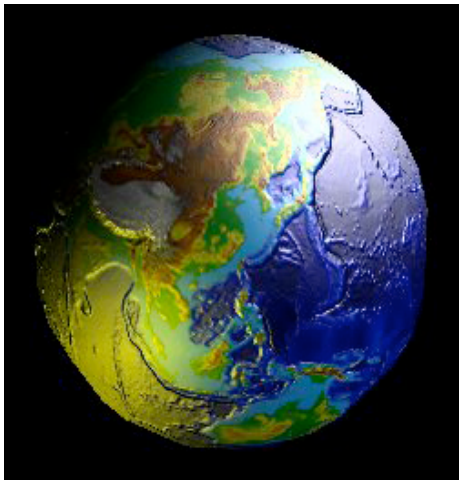
Texture maps and bump maps





Sphere with texture map





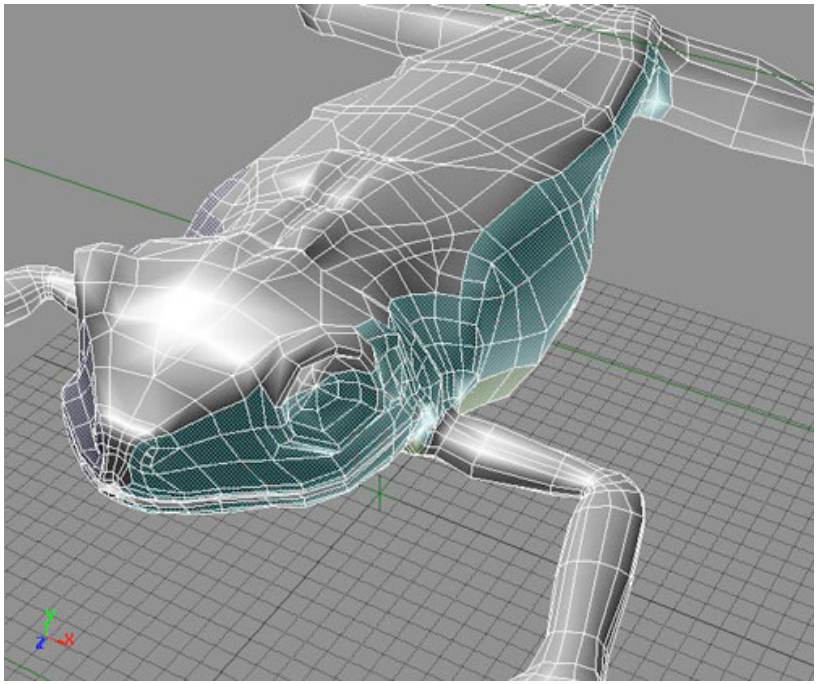
Sphere with texture map and and bump map

Texture maps and bump maps

Texture map (color detail)

Bump map (geometric detail)

Texture maps and bump maps



3D geometry

3D geometry rendered with texture & bump maps

Environment maps

a.k.a. **Reflection maps**

Texture is used to model the environment around a 3D model

Rays are bounced off objects into the environment
Color of the environment determines color of the illuminated object
A simple form of **ray tracing**

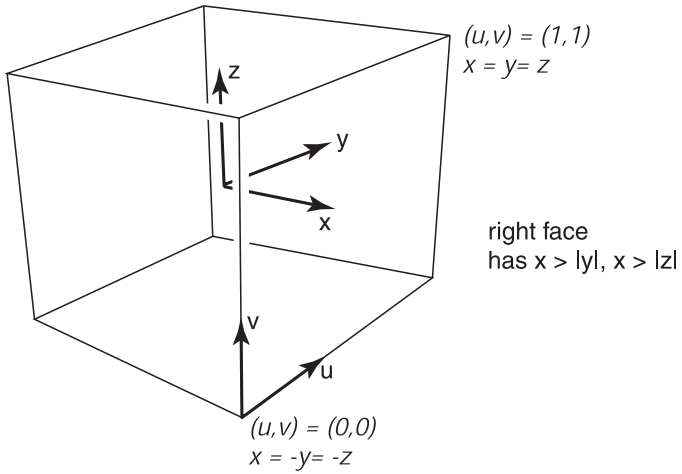
Store as a sphere or **cube map**

Cube map

Infinitely large cube with textured faces

Environment textures determine the illumination of objects inside the cube

(With simplifying assumptions) can be done real-time in hardware



Source: Unreal Engine

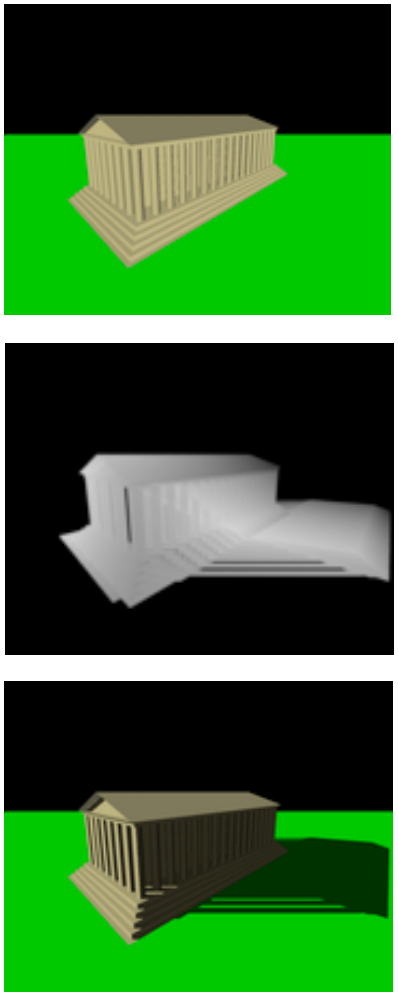
Shadow maps

Add shadows to 3D scenes

From the light's point of view, determine the depth of objects in the scene

Store the depth as a texture

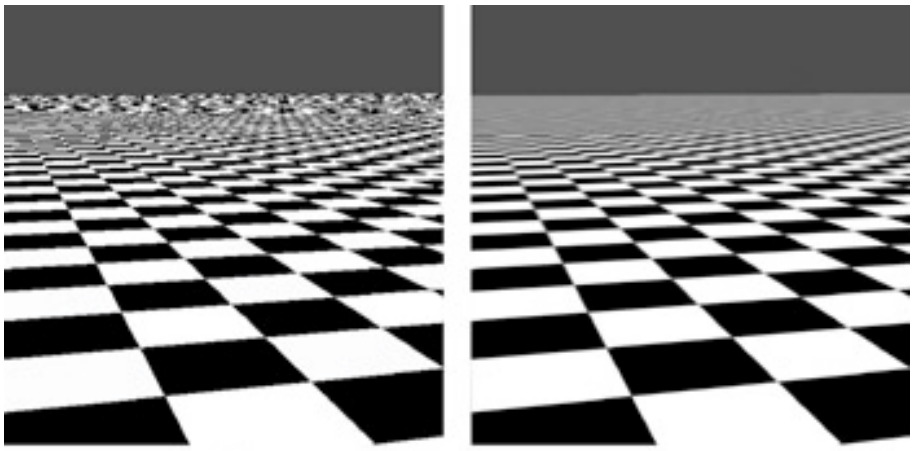
Determine whether points are in shadow or light by testing visibility against the **depth map**



Source: Wikipedia

Aliasing

Point-sampling the texture map can result in aliasing artifacts



Source: NVIDIA

Proper **antialiasing** requires area averaging in the texture (details next week)

Summary

Solid and gradient fill
- Interpolating colors

Pattern fill

Texture maps
- Interpolating texture coordinates
- Pineda's parallel rasterization algorithm
- Bump maps
- Displacement maps
- Environment maps

Lab 2: 2D Drawing

Midpoint line algorithm (any variant)

Polygon drawing

Fractal drawing

Due 10/1