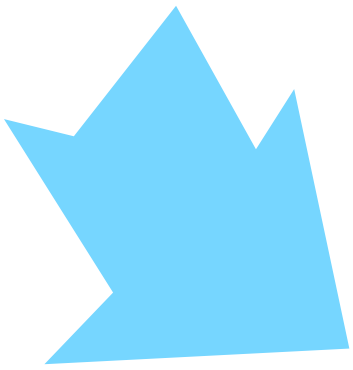
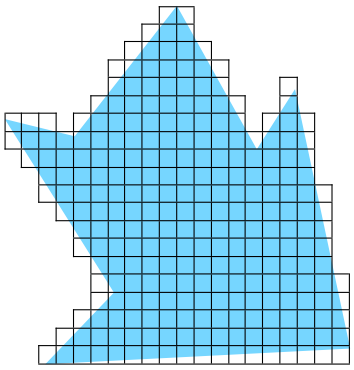
Polygons and Shading

- The last major step in the journey to the viewport is the actual painting of the polygons
- Painting the polygon includes a number of substeps:
  - How does the polygon translate into its final pixels?
  - Which pixels of the polygon are painted?
    - With *z*-buffer HSR, painting and HSR occur in the same loop
  - What color(s) should be used?
    - Color based on absolute color values vs. lighting model vs. textures vs. combination
    - Single color for the entire polygon vs. shaded
    - Blending with colors that have already been painted?

Scan-Line Conversion

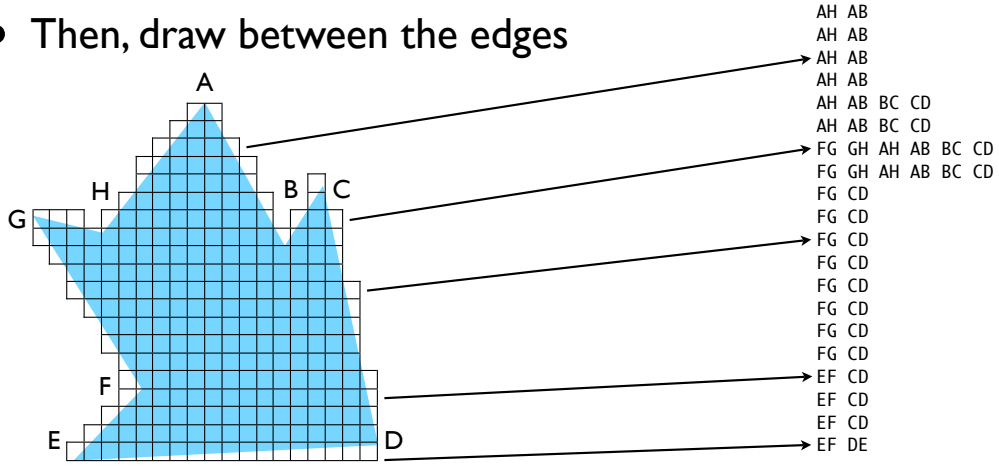
- Converts floating-point vertices into discrete pixels
- We do this repeatedly, so must be as fast as possible





Scan-Line Conversion Algorithm

- From top to bottom, build an *active edge table* showing what edges are "active" per scan line
- Then, draw between the edges



- Nested loop — for each scan line, then for each pixel along the scan line:
  - Compare the *z* value at that pixel to the current *z* in the depth buffer — if the current *z* is closer, then we skip this pixel: it is occluded
  - If we decide to draw this pixel, then we record its *z* value in the depth buffer, then decide on the color to use
- Options for determining the color:
  - *Absolute assignment per vertex* — in which case we need to determine what color to use *in between* vertices
  - *Assignment per vertex as a material* — which means that we have to factor in lighting calculations to determine the final color (note how this is still per vertex, so the question of how to light the pixels in between vertices remains)
  - *Mapping from a texture* — must determine where in the texture a pixel will "land," and use the color at that texel (based on the texture coordinates assigned to each vertex)
  - A combination of all of these, including whatever color might already be there, if blending or translucency is desired (e.g., fog effect)

Color from Lighting

- Light model approximates physics
- Light sources add up (i.e., red light combines with blue light to produce magenta light)
- Light intensity upon reaching polygon modified by:
  - Angle of incidence for diffuse light (the closer to the negative of the normal, the brighter)
  - Distance from polygon for attenuated light
  - Shininess of material for specular components

- Light and material are multiplied to determine the color at the polygon
  - For example, when magenta light [1.0, 0.0, 1.0] hits yellow material [1.0, 1.0, 0.0], you get [1.0 * 1.0, 0.0 * 1.0, 1.0 * 0.0] = [1.0, 0.0, 0.0] or red)
- Components (ambient, diffuse, specular) are added up to get the final color — per vertex
- Any final values > 1.0 are capped to 1.0
- Note how this model is solely between a polygon and its light sources — no interactions with other objects
  - Thus, no reflections and no shadows with this approach to lighting

Color from Texture

- Texture coordinate-to-vector mapping determines how a texture is "wrapped" onto a polygon
- Intervening pixels are interpolated
- Mapped texture color can be used as:
  - A material component (typically diffuse or all three) of the object at that pixel
  - An absolute color to be blended by some function
- If textures *and* lighting are active, the mapped texture color is combined with the current light color using some customizable function, such as a blend

Blending with Pre-existing Colors

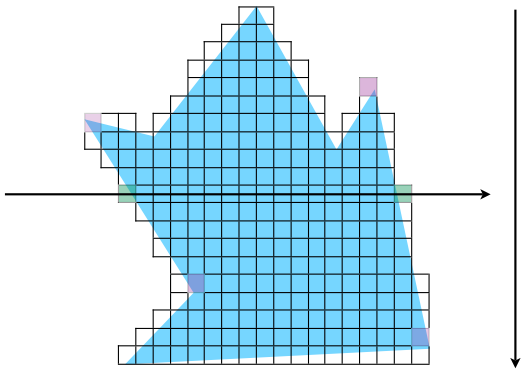
- When blending, the current pixel is sampled to see what color is already there
- This color is combined with the calculated color using some function — lots of ways to do this
- The final "blended" color is the result of this function
- Note how blending is sensitive to the order in which you draw your model — also it must not perform HSR, or else occluded pixels will never make it to the frame buffer!

Interpolation and Shading

- Note that so far, if we aren't doing any texture mapping, we only have the colors at the vertices
- How do we determine the color(s) in between?
- Two prominent algorithms: Gouraud and Phong
- Gouraud shading is the less computationally expensive algorithm; it is what OpenGL uses
- Shading has so many variations and possibilities that they are *programmable* in OpenGL 2.0 — algorithms are specified using a *shading language*

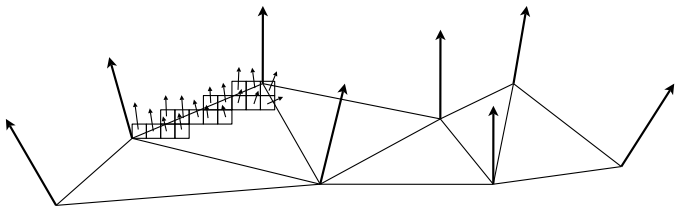
Gouraud Shading

- Take the colors at the vertices of each line segment, then interpolate vertically per scan line
- Within each scan line, interpolate horizontally



Phong Shading

- Instead of calculating the color for each vertex then interpolating the colors, Phong shading *interpolates the normal* at each pixel and performs the lighting calculation *per pixel*
- Shading quality is better but it's more work



Endless Possibilities

- Unlike geometric/vertex calculations, the area of shading is quite open-ended
- As mentioned, arbitrary/generalized reflections and shadows are still missing
- Other real-world-approximating features:
  - *Bump mapping* — textures "raise" or "lower" polygon's pixels (tree bark, golf ball dimples)
  - Filaments/fibers (fur, hair, cloth)
  - Liquids/refraction
- Computer graphics in entertainment (films, games) typically push the state of the art