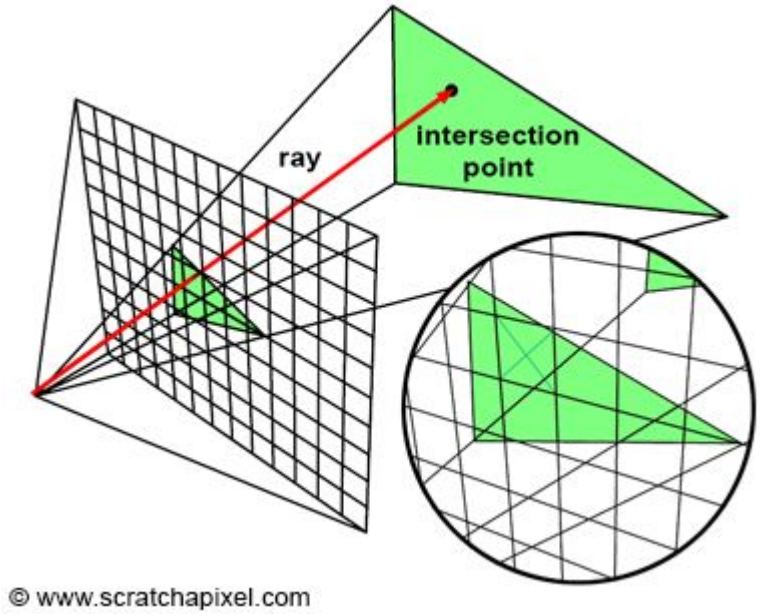
Shading and Visibility

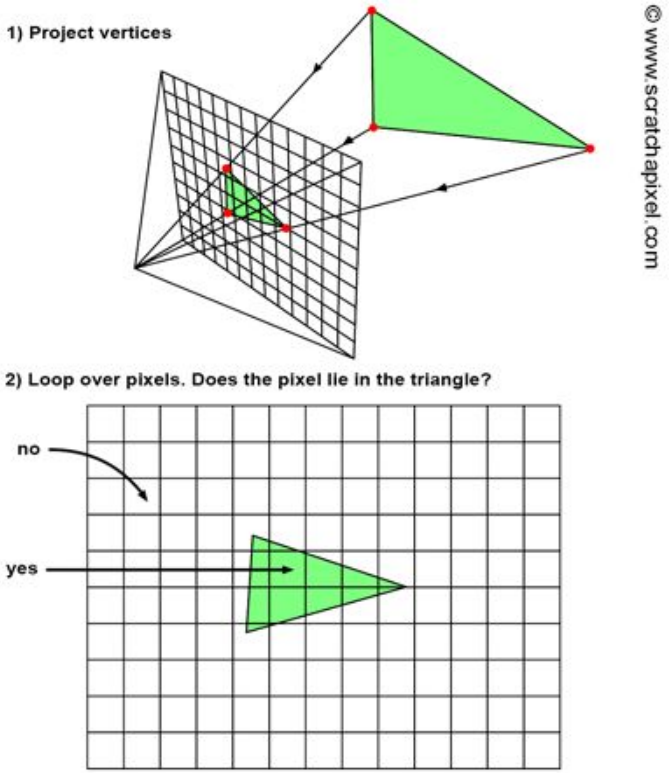
Raytracing is image-based rendering



Raytracing is image-based rendering

```
for each pixel do
  compute viewing ray
  for each object/triangle in scene do
    if (ray hits an object with t ∈ [0,∞)) then
      Compute n
      Evaluate shading model and set pixel to that color
    else
      set pixel color to background color
```

Image-based vs **object-based**



Rasterization - also called scanline rendering

```
for each object/triangle in scene do
  project vertices of triangles to screen
  for each pixel do
    if (pixel is contained in projected triangle) then
      Evaluate shading model and set pixel to that color
    else
      set pixel color to background color
```

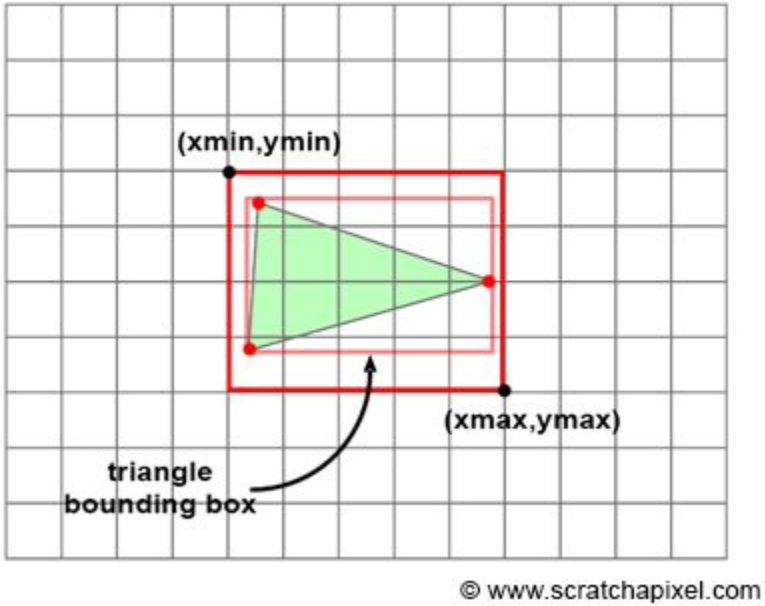
Side note: how can we use our BVH in rasterization?

```
for each object/triangle in scene do
  project vertices of triangles to screen
  for each pixel do
    if (pixel is contained in projected triangle) then
      Evaluate shading model and set pixel to that color
    else
      set pixel color to background color
```

Side note: how can we use our BVH in rasterization?

```
for each object/triangle in scene do
  project vertices of triangle and bounding box to screen
  for each pixel in bounding box do
    if (pixel is contained in projected triangle) then
      Evaluate shading model and set pixel to that color
    else
      set pixel color to background color
```

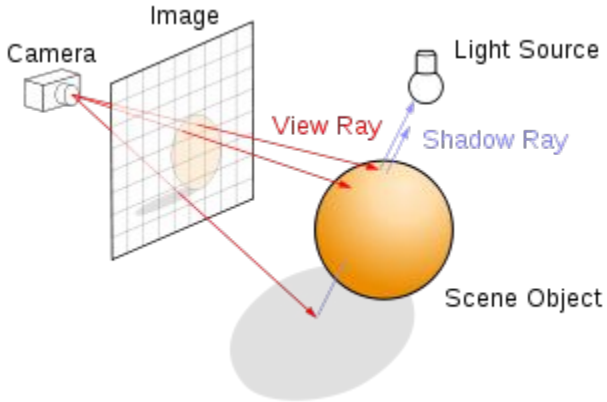
Side note: how can we use our BVH in rasterization?



But how do we determine which objects are visible?

In raytracing, each pixel color is determined by the ray we shoot through it

So the ray only gathers information from objects which contribute to that pixel

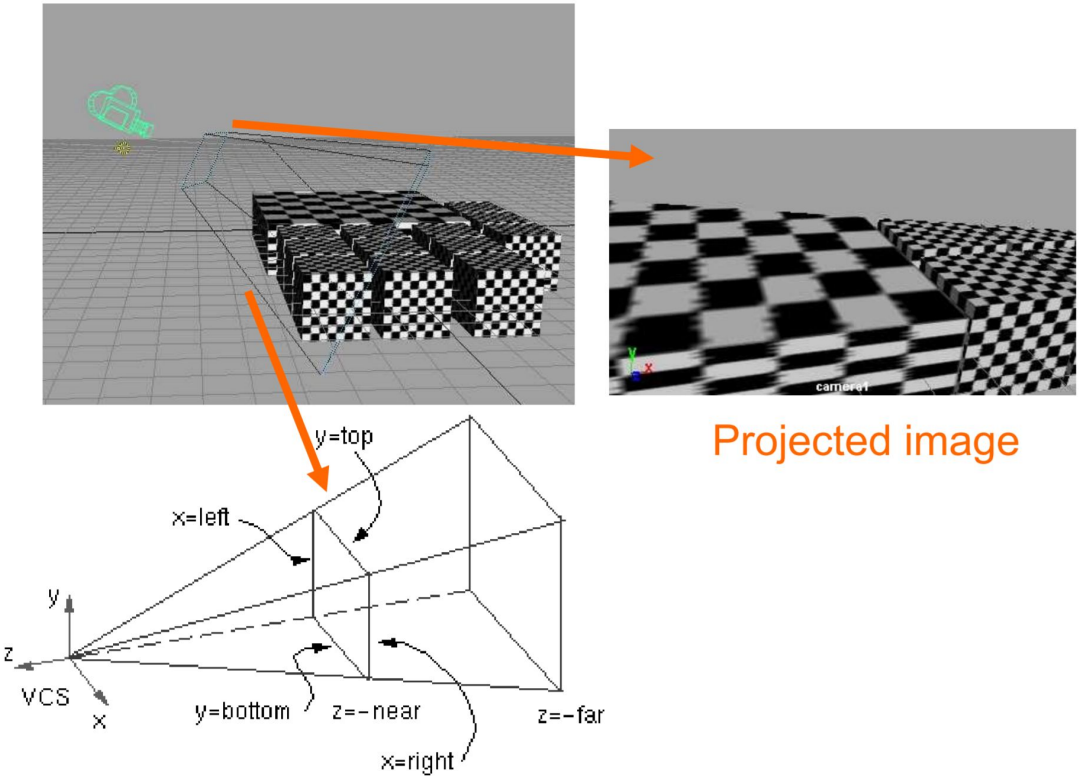


But how do we determine which objects are visible?

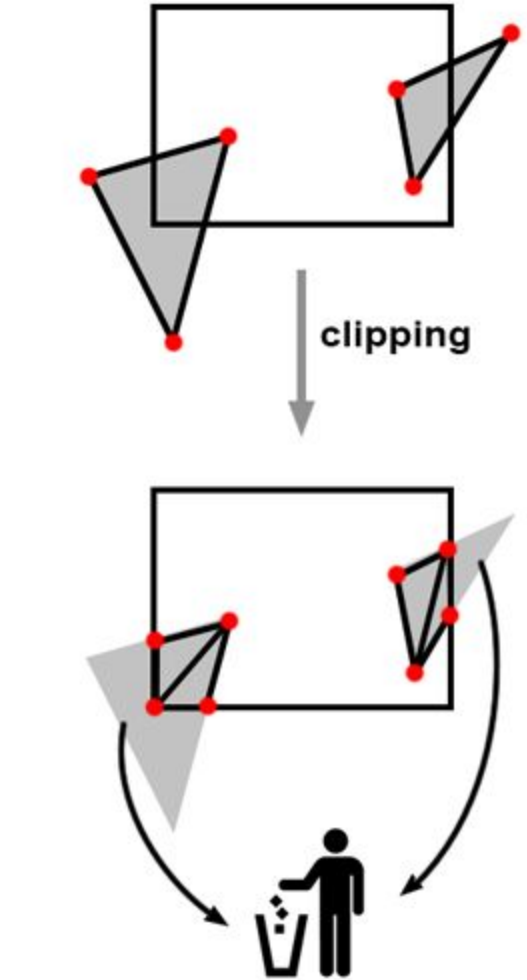
In rasterization, we project all vertices to the screen

- How can we remove objects which are outside the viewing volume?
- How can we render only the parts of the objects that are facing the screen?
- How can we get the correct ordering of objects?

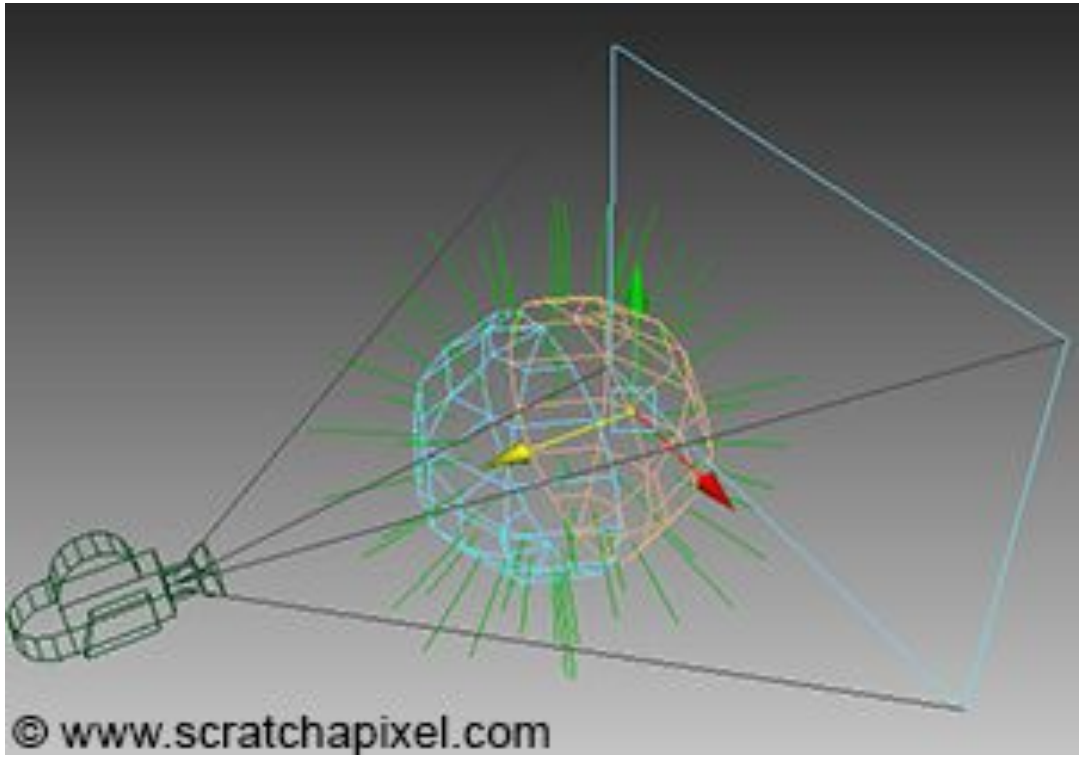
Viewing volumes



Clipping



Culling



Culling

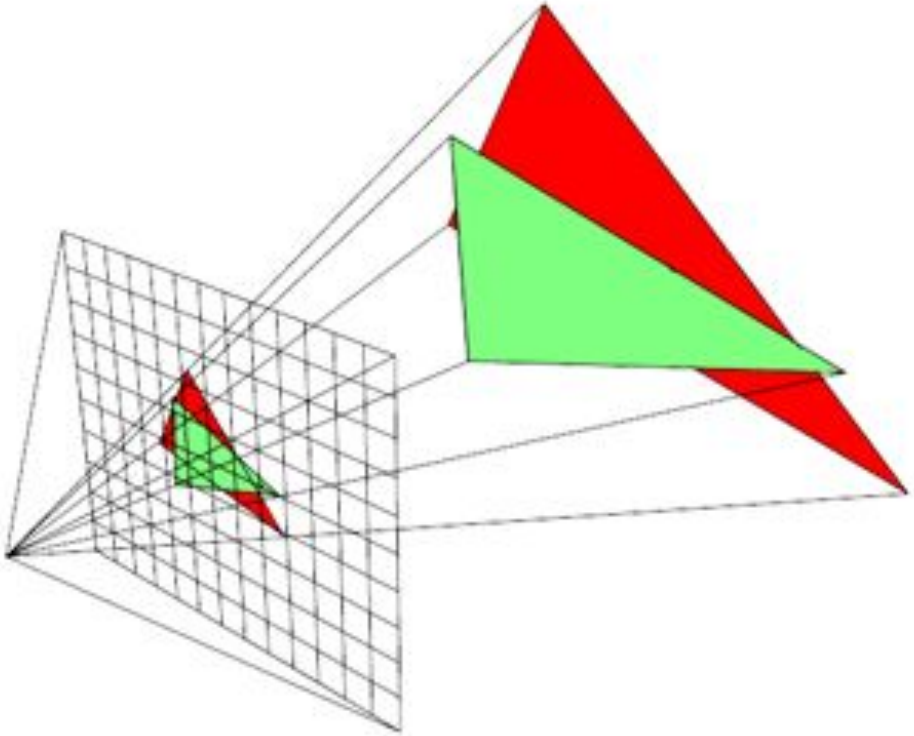
Only geometry whose normals are facing the camera will be visible in the final image

Discard all triangles where the dot product of their surface normal and the camera-to-triangle vector is greater than or equal to zero

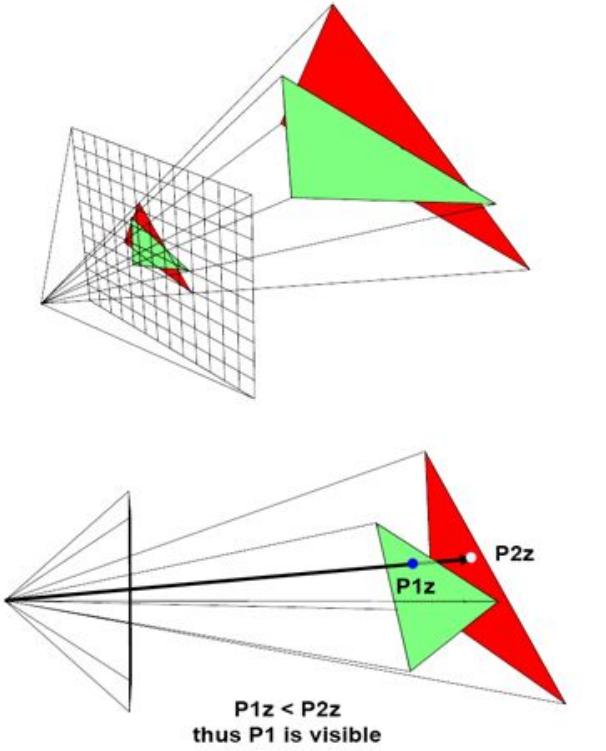
$$(V_0 - P) \cdot N \geq 0$$

P is viewpoint, V is vertex in triangle, N is normal of triangle

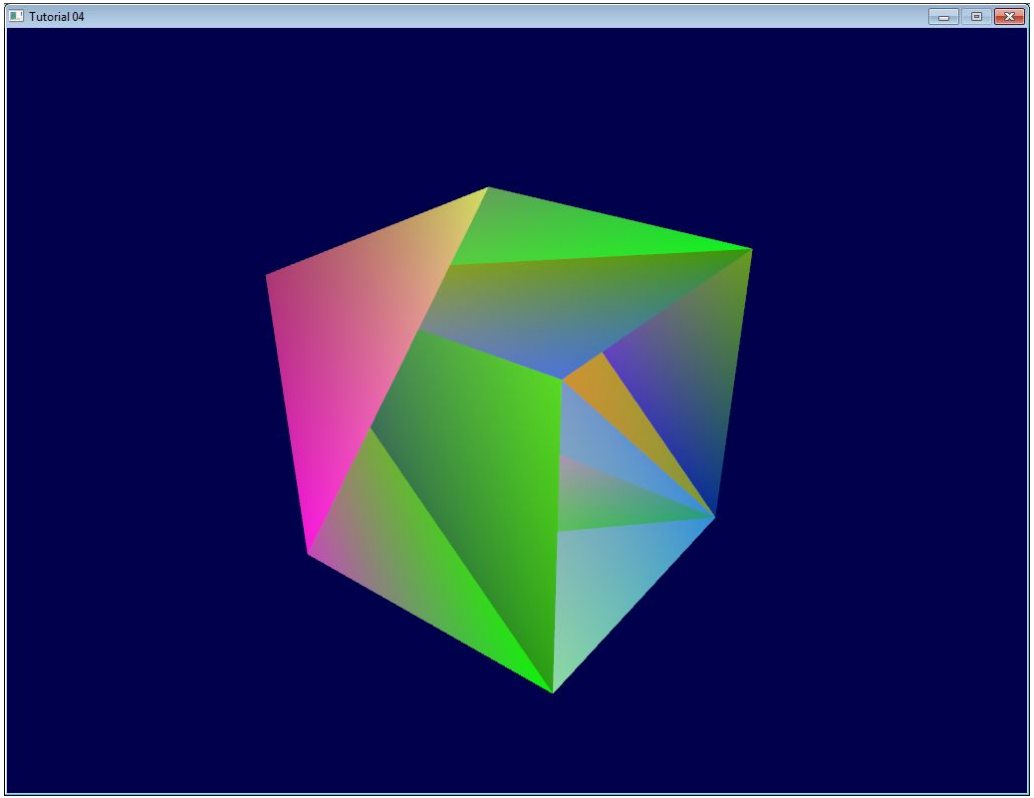
But how do we determine which objects are visible?



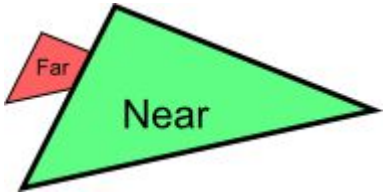
But how do we determine which objects are visible?

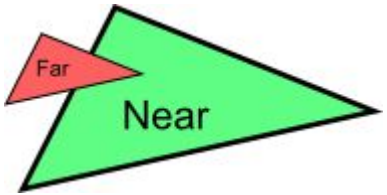


What can happen?

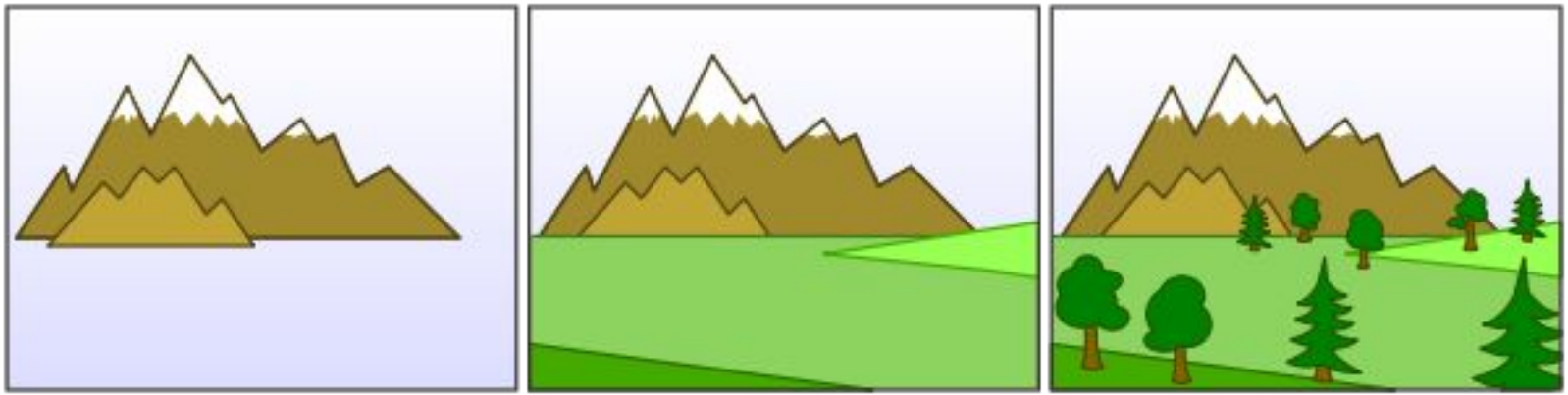


What can happen?





Painter's algorithm



Painter's algorithm

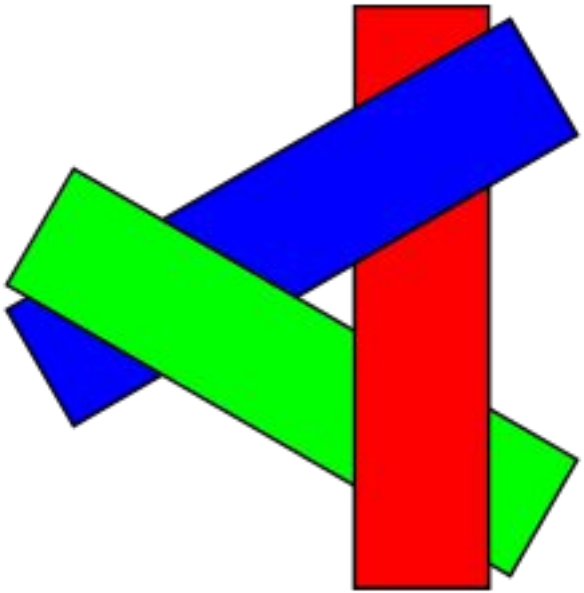
also known as priority fill

sorts all the polygons in a scene by their *furthest* depth and then paints them in this order, farthest to closest

paint over the parts that are normally not visible

What could possibly go wrong?

Problems with Painter's



Why does this happen?

topological sorting

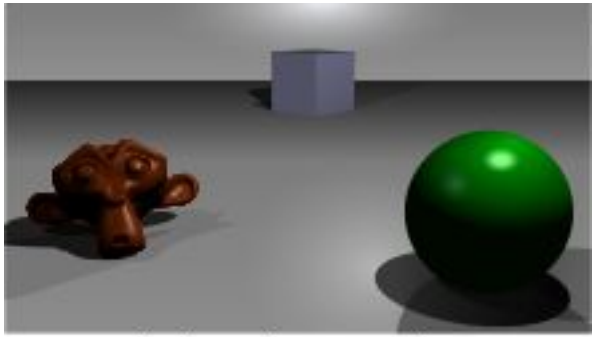
recall: like DFS

Problems with Painter's

Renders everything, even if not needed

Computationally expensive

Z-buffer



A simple three-dimensional scene

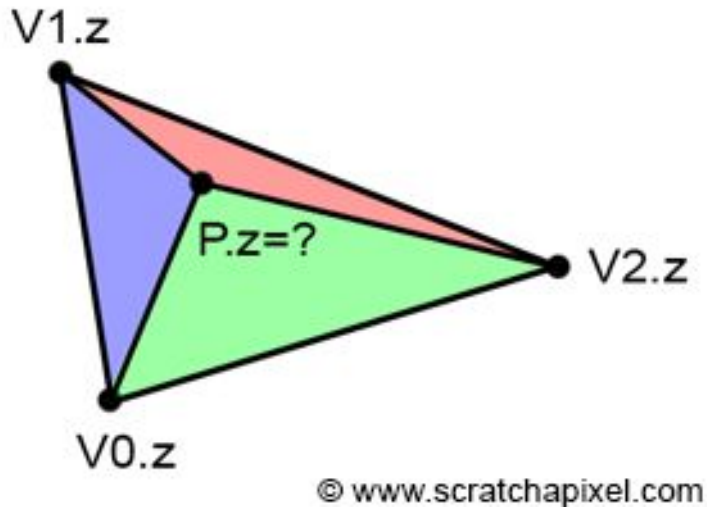
Z-buffer representation

Z-buffer

```
initialize z-buffer to size of # of pixels with max values
for each object/triangle in scene do
  project vertices of triangles to screen
  for each pixel do
    if (pixel is contained in projected triangle) then
      Compute z
      if (z is closer than object stored in z-buffer)
        update z-buffer with new z value
```

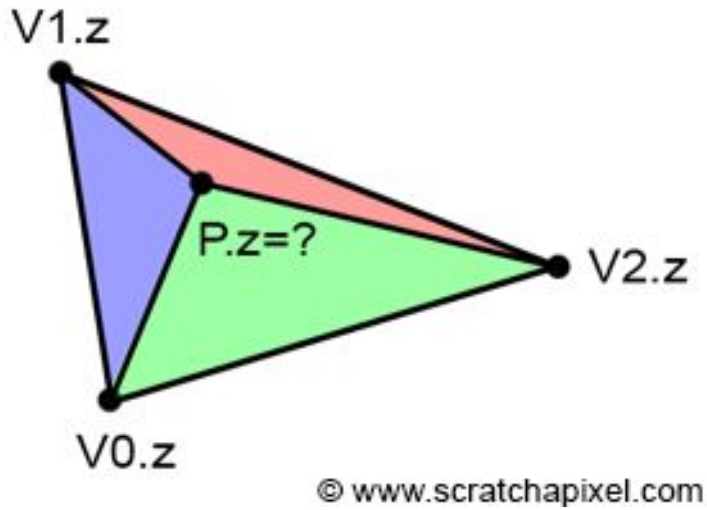
Z-buffer

How to compute z values?



Z-buffer

Can we use barycentric coordinates?

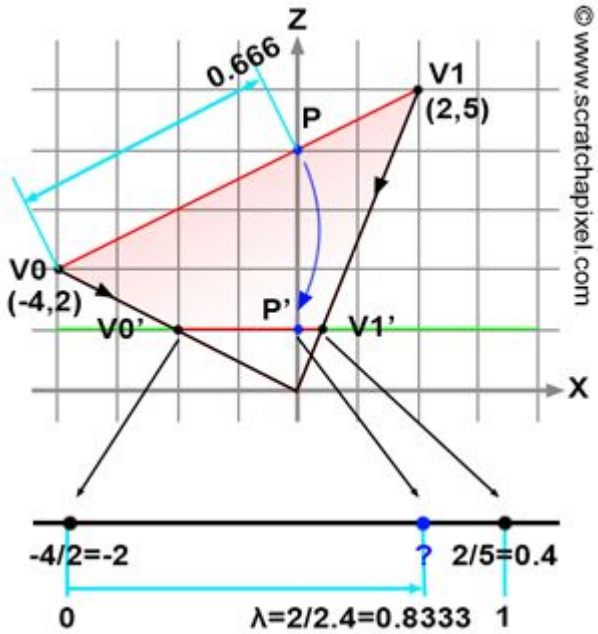


Z-buffer

No! 🤖

after we projected the vertices and performed the perspective projection divide, the z values don't vary linearly anymore!

perspective projection does not preserve distances



Z-buffer

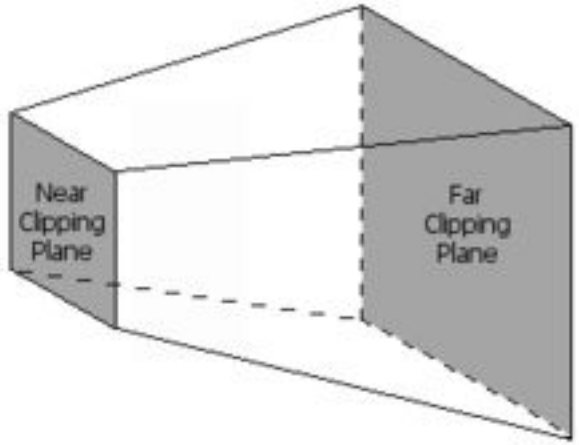
Computing z values (these are normalized between -1 and 1)

perspective projection:

$$z' = \frac{far + near}{far - near} + \frac{1}{z}\left(\frac{-2 \cdot far \cdot near}{far - near}\right)$$

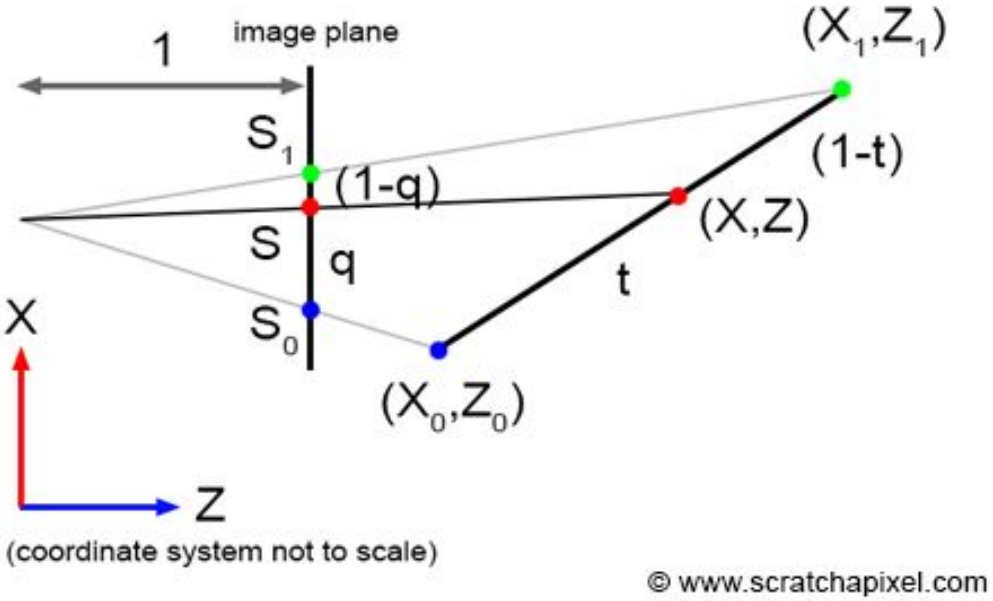
orthographic projection:

$$z' = 2 \cdot \frac{z - near}{far - near} - 1$$

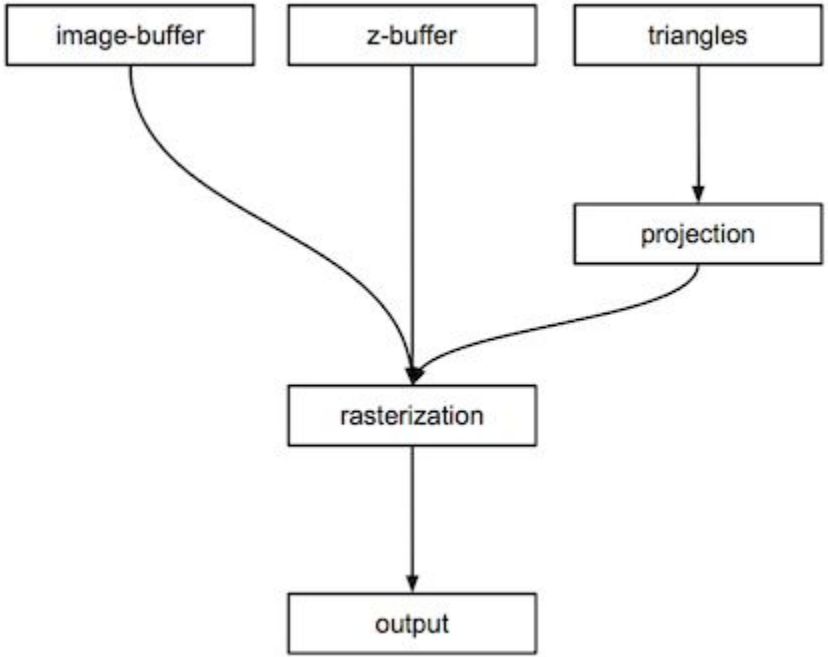


Z-buffer

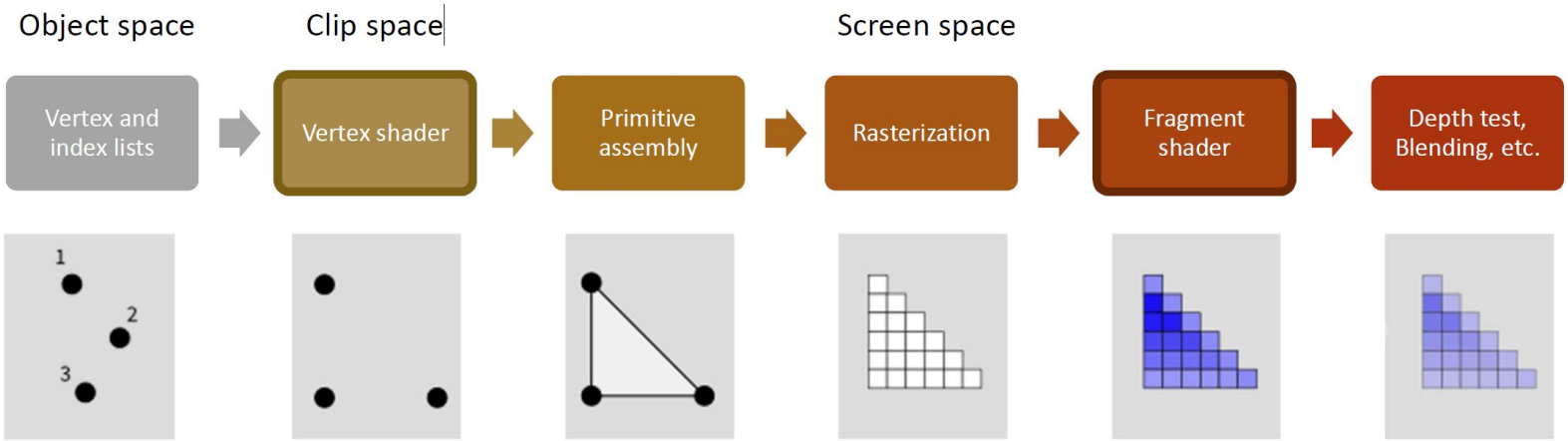
Why the inverse?



Pipeline (space)



Pipeline (time)



Pipeline

We do things to each vertex, and then to each pixel

This is what we call "embarrassingly parallelizable"

Let's do it to each unit (vertex and fragment) at the same time

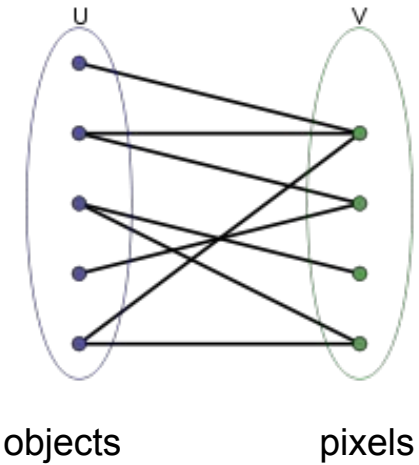
Data dependency

Why is it so parallelizable?

In raytracing, we use a ray for each pixel.

Parallelizable? yes, but for one pixel, we need multiple objects' information.

Rays bounce and data is accumulated.

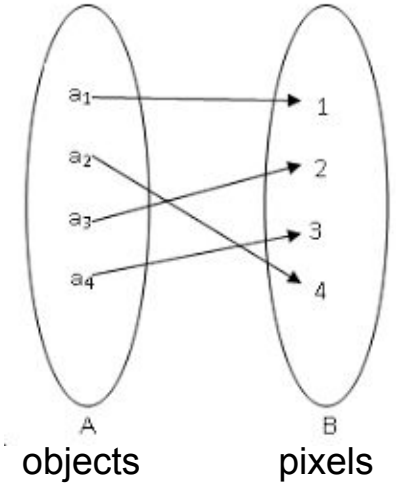


Data dependency

Why is it so parallelizable?

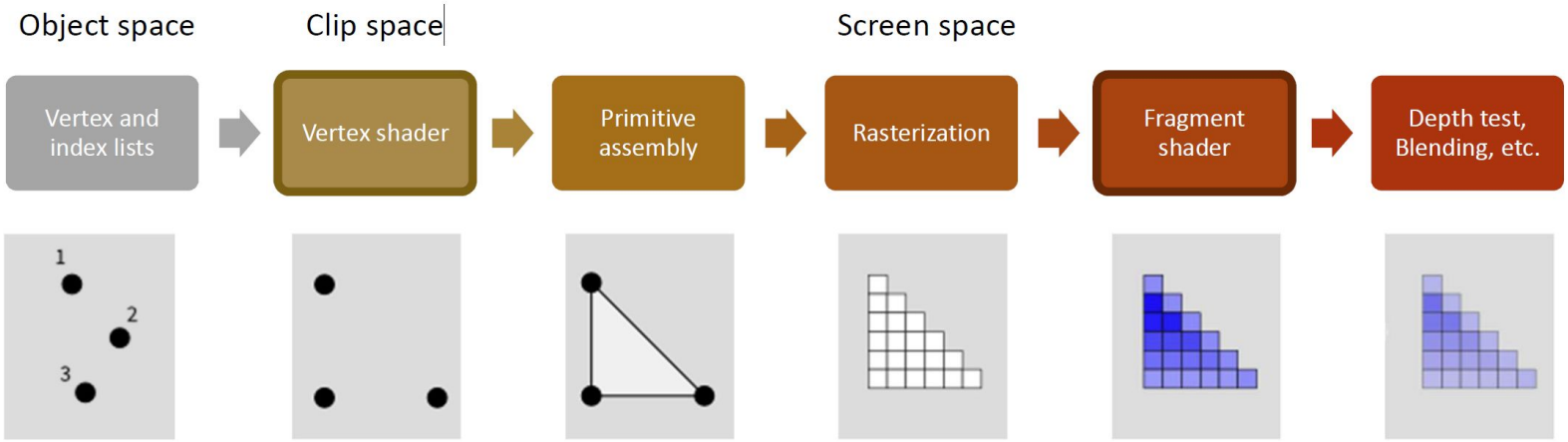
In rasterization, for each pixel, we only need data for one object at a time.

We can do everything to each vertex *and* each fragment at the same time.



It's so parallelizable, our hardware does it for us

a thread for every pixel!



Programmable shaders

http://www.cs.toronto.edu/~jacobson/phong-demo/

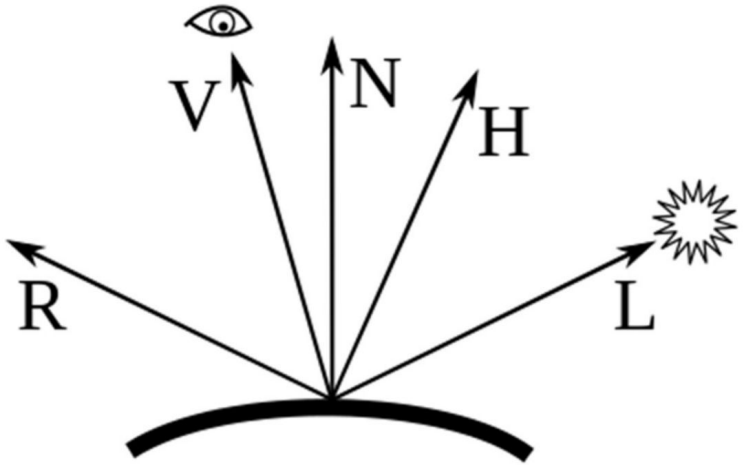
Phong vs Gouraud Shading

```
for each object/triangle in scene do
  project vertices of triangles to screen
  for each pixel do
    if (pixel is contained in projected triangle) then
      Evaluate shading model and set pixel to that color
    else
      set pixel color to background color
```

Phong vs Gouraud Shading

```
for each object/triangle in scene do
  project vertices of triangles to screen
  for each pixel do
    if (pixel is contained in projected triangle) then
      Evaluate shading model and set pixel to that color
    else
      set pixel color to background color
```

Recall the Phong model



shininess

Ambient term

$$I_{\mathrm{p}} = k_{\mathrm{a}} i_{\mathrm{a}} + \sum_{m \in \text{ lights}} \left(k_{\mathrm{d}} (\hat{L}_m \cdot \hat{N}) i_{m,\mathrm{d}} + k_{\mathrm{s}} (\hat{R}_m \cdot \hat{V})^{\alpha} i_{m,\mathrm{s}} \right)$$

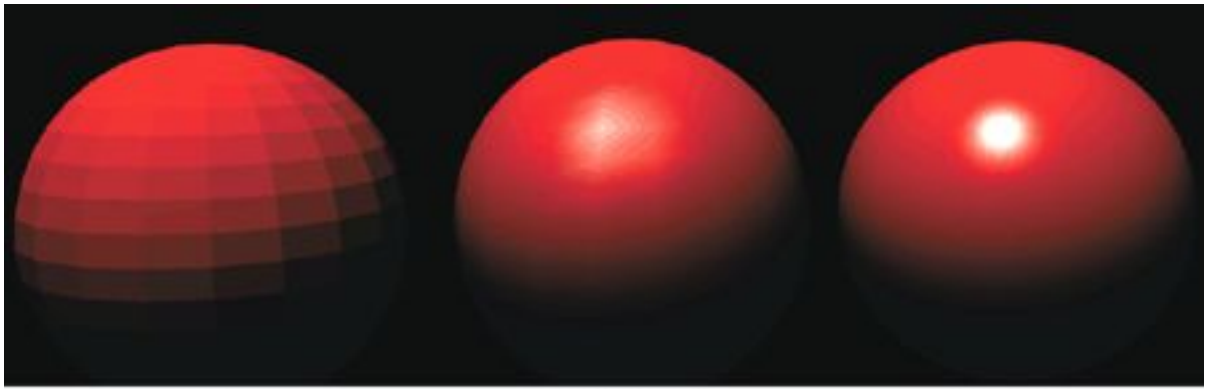
Diffuse term

Specular term

Phong vs Gouraud shading

Ordering is different

Interpolation at different stages

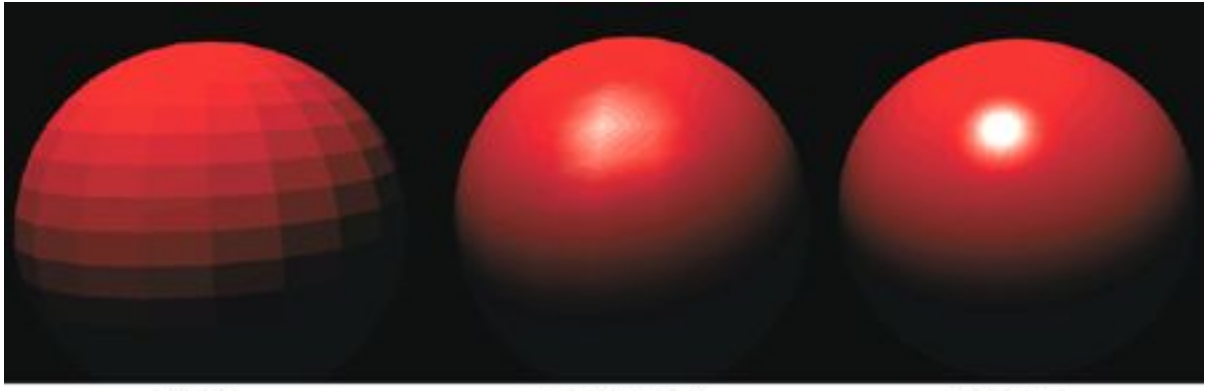


Flat shading

Find triangle normals

Apply Phong model to each triangle

Each triangle (or other polygon) has a constant color across it

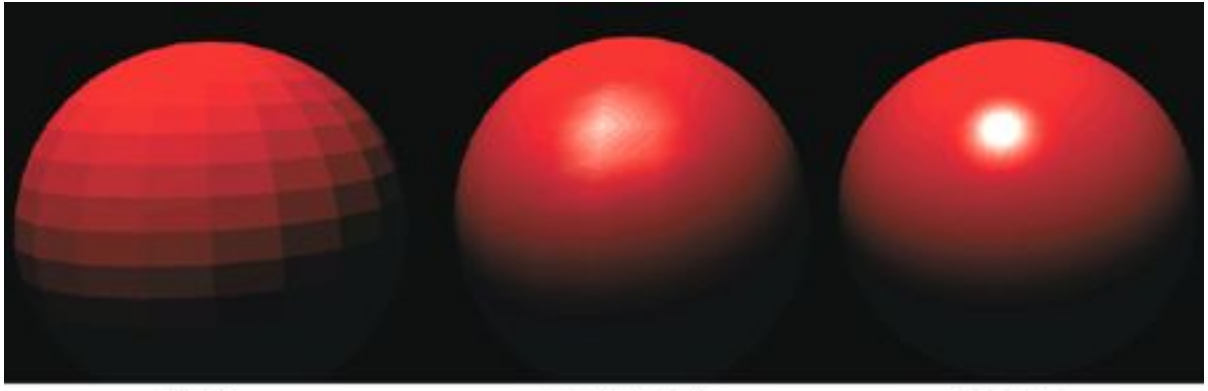


Gouraud shading

Find averaged vertex normals (like in the mesh assignment)

Apply Phong model to each vertex

Interpolate vertex colors across each triangle



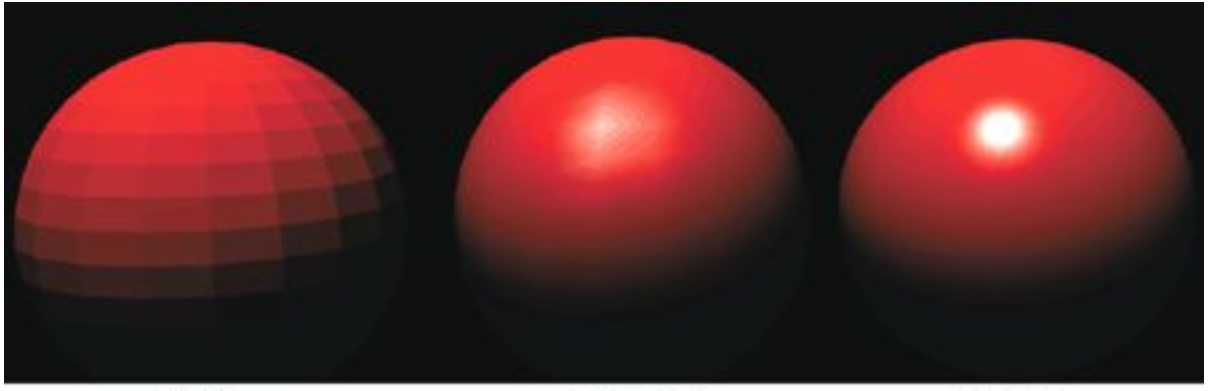
Phong shading

Find averaged vertex normals

Interpolate normals across triangle edges

Interpolate edge normals across triangle (or other polygon)

Apply Phong model to each fragment (which corresponds to each pixel)



Shaders can do cool things!

https://www.shadertoy.com/