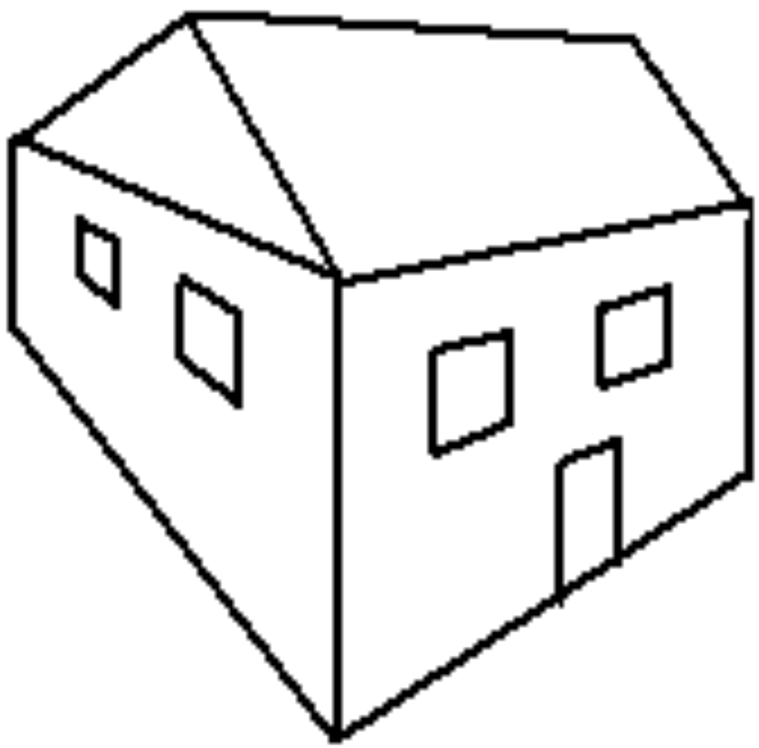
Computer Graphics

Lecturer: Simon McLoughlin

Lecture 7



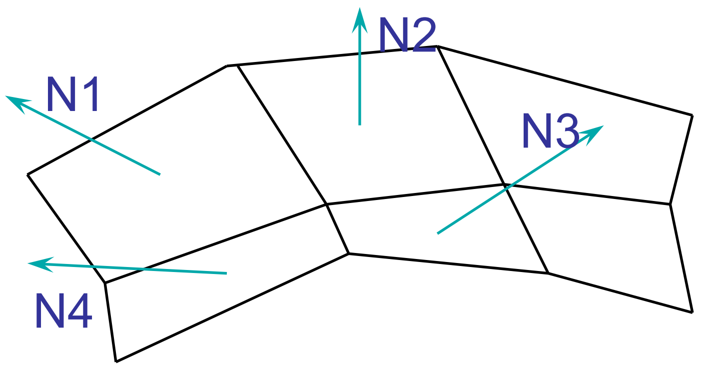
Introduction to Shading

- In the last lecture we saw how to do compute a lighting value (based on diffuse reflection) for a point on an polygon surface and we saw how to represent an object as a mesh of polygons.
- We can then render our model using a technique known as constant shading. This techniques proceeds as follows:

Pick a point on the surface of the polygon

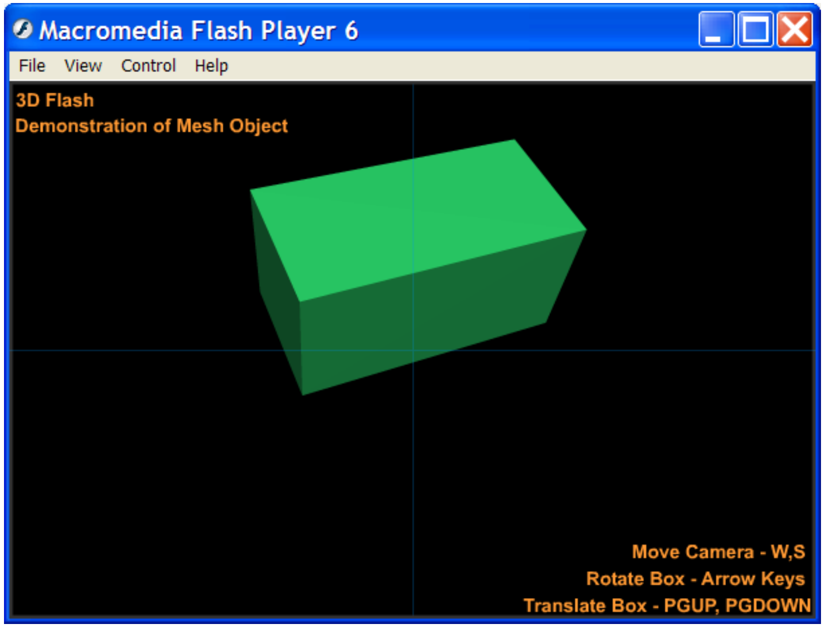
Apply the lighting model to get a colour value for this point (do this based on the normal vector for the polygon)

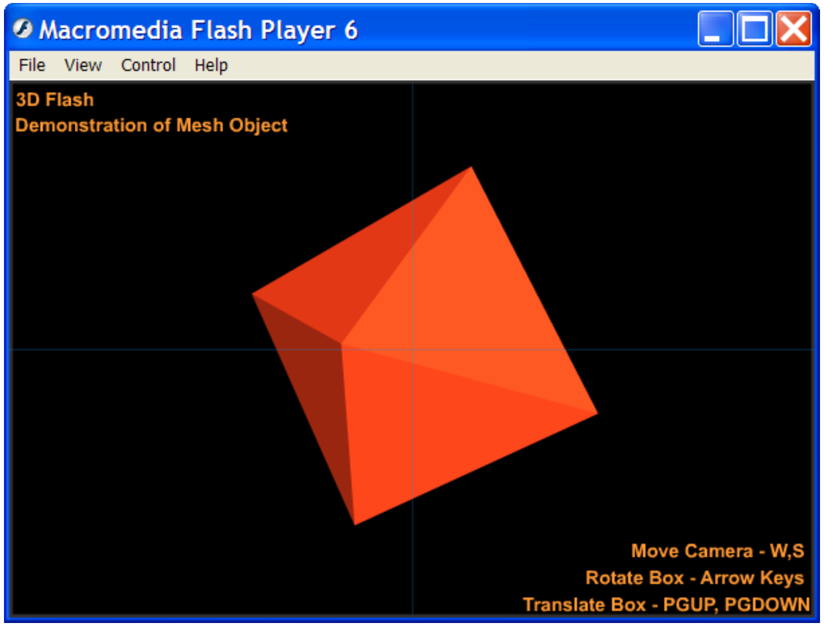
Project the polygon and then fill the projected polygon with this colour.



Constant Shading

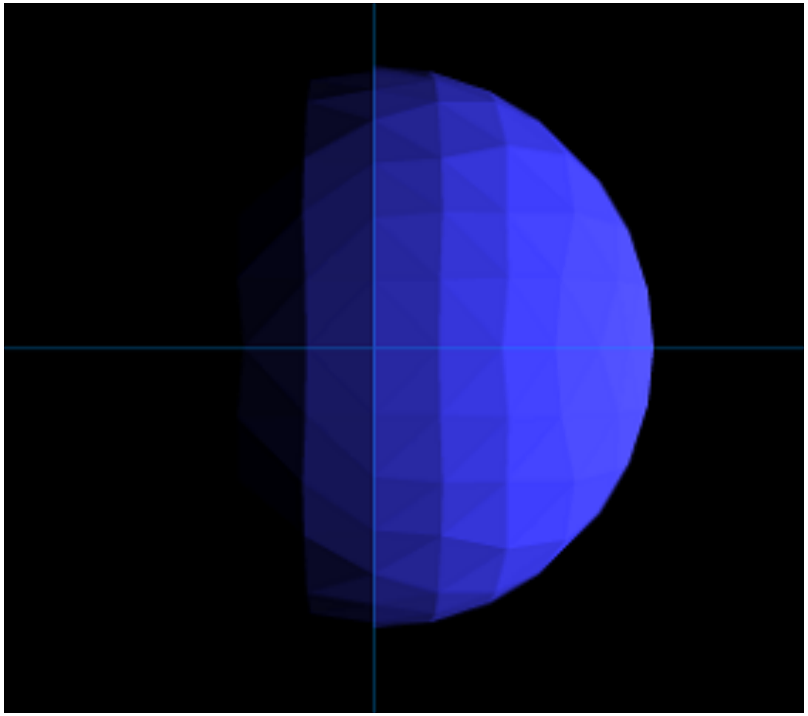
- The screen shots below show the results after implementing this and trying it out.





- In both cases it looks fine. Another example overleaf.

Constant Shading



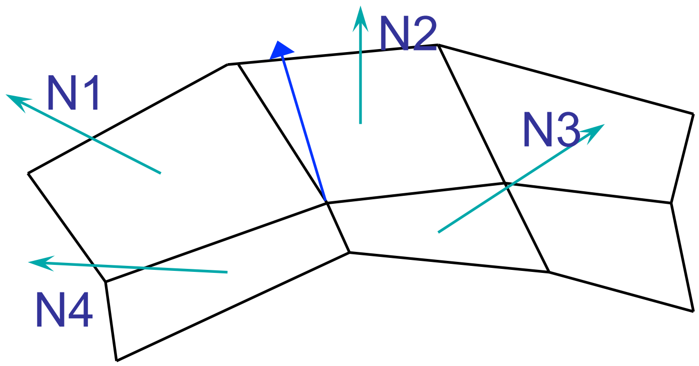
- This example doesn't look too good.
- The difference with this object is that the mesh is an approximation of a curved surface.

Constant Shading

- Constant shading is okay in the following situation:
- Object has flat surfaces and hence the surfaces can be modelled exactly with polygons (e.g. a box)
- The light source is sufficiently far away that there is no significant gradation in shading across the surface.
- So, for example, a box lit by a light a long distance away will look fine with constant shading.
- When we have a box made out of triangular polygons then we see the joins between the polygons if the lights are too close.
- Problems are made worse by the Mach band effect. This is when our visual system accentuates the boundaries between areas of slightly different colour.
- The Mach band effect results in polygon boundaries looking like raised edges. Not good!

Gouraud Shading

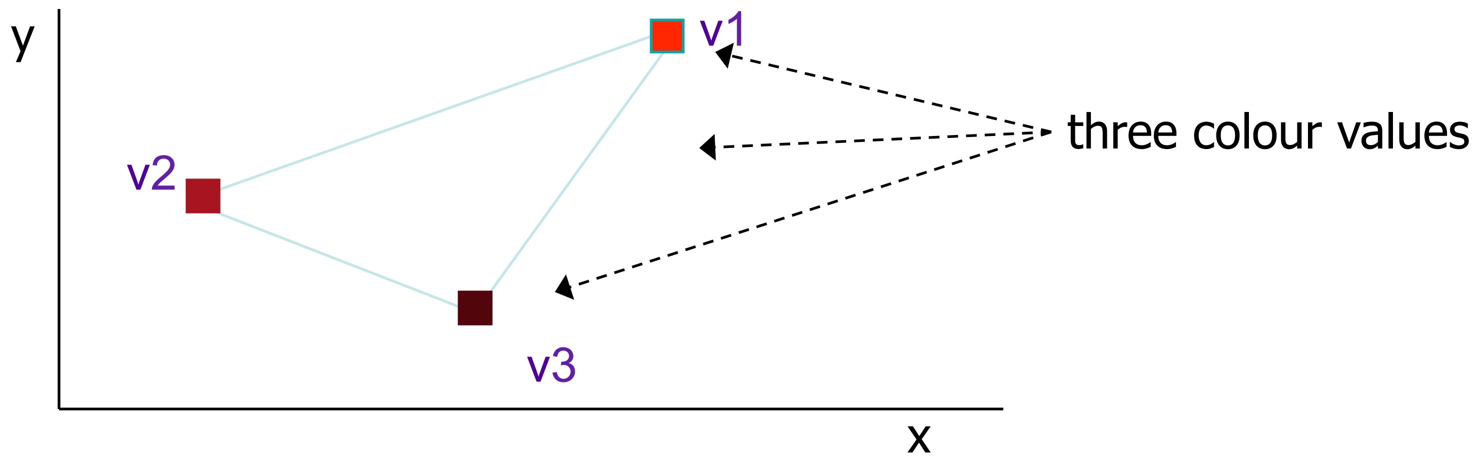
- In Gourand shading we calculate a colour value, using the Phong lighting model, at the vertices of each polygon, and then interpolate these values in order to get colour values for projected pixels.
- How do you run the lighting model at a vertex?
- What's the normal vector at a vertex?
- Get a normal vector for a vertex (a vertex normal) by averaging the normal vectors of the polygons that meet at that vertex.
- Do this by adding them together and then normalising.



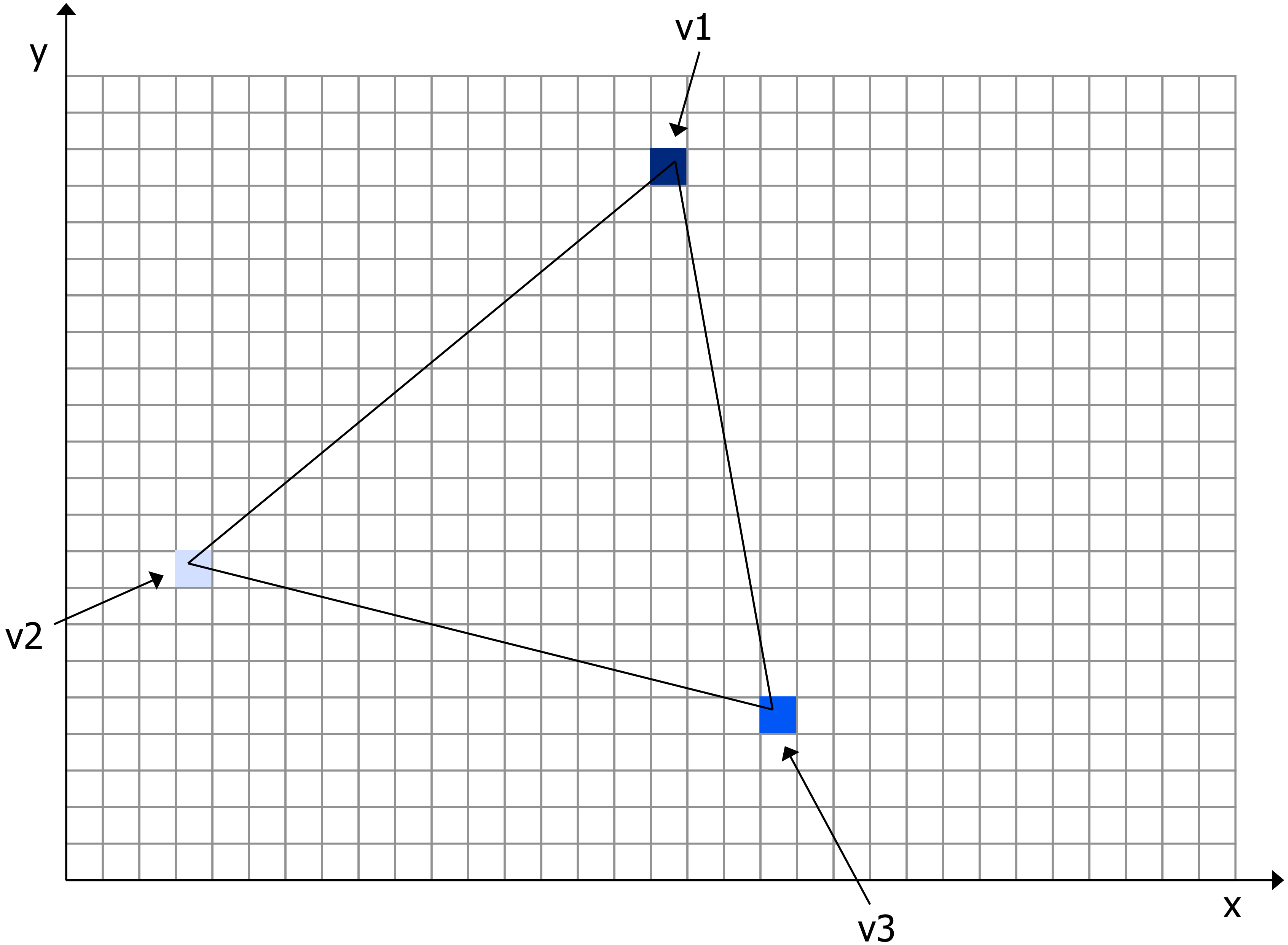
$$V1 = N1 + N2 + N3 + N4$$

Gouraud Shading

- Now we have a normal vector, we can run the Phong model to get a colour value.
- Suppose that the polygon is a triangle. It then has three vertices so we calculate three vertex normals and hence three colour values.
- Then we project the polygon and shade the interior on a pixel by pixel basis.



Gouraud Shading

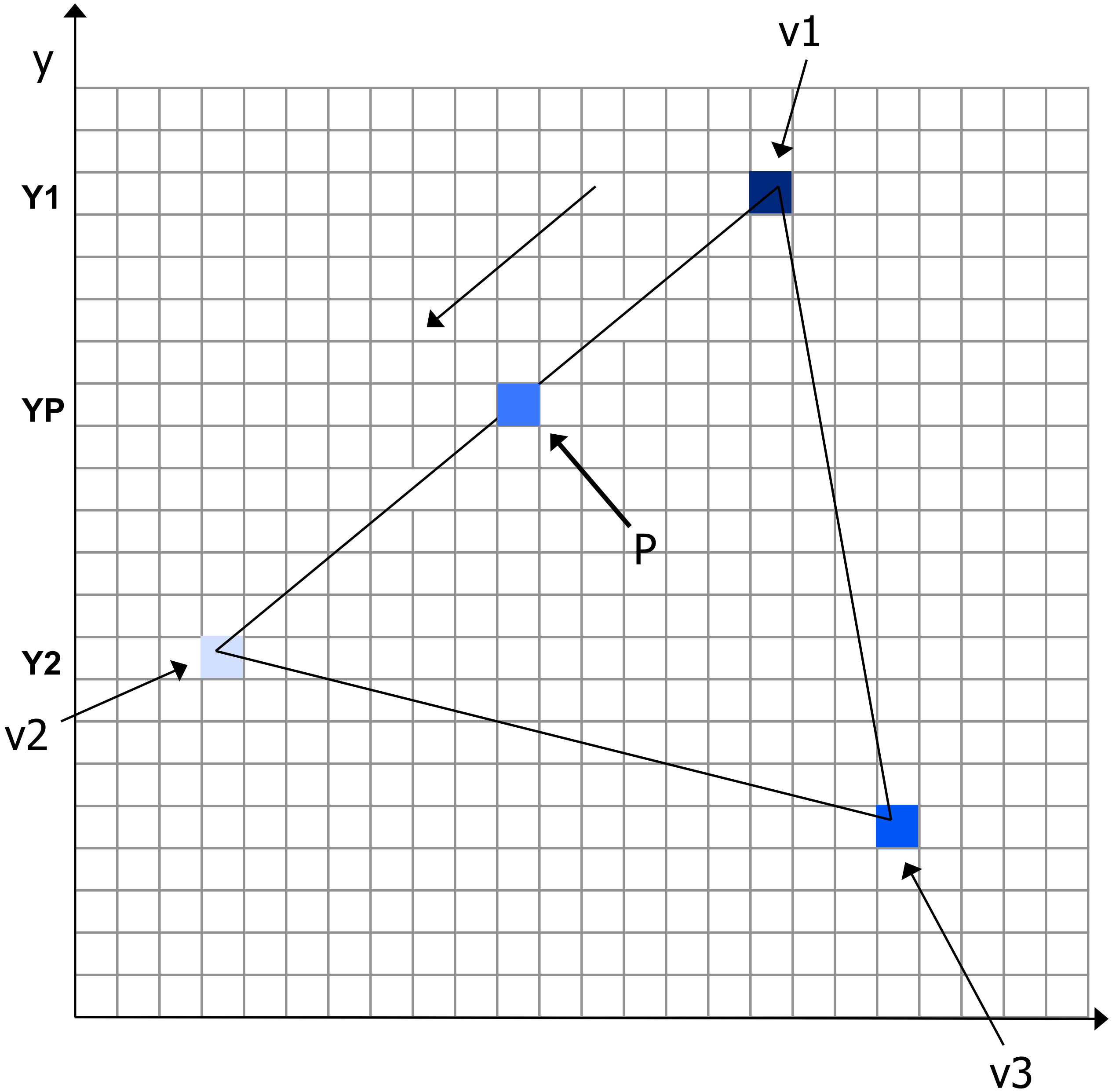


Gouraud Shading

- Previous slide shows the situation when the algorithm starts.
- We have three vertex colour values.
- We proceed by interpolating values along the edges of the polygon.
- This works on the principle that if we are trying to calculate a colour value for a pixel that is halfway along the edge between v1 and v2 then it receives a colour value that is halfway between the two of them.
- The following formula calculates a colour value for a pixel p which is along the edge between v1 and v2 (I1 is the colour value of pixel v1, I2 the colour value of pixel v2, y1 is the y coordinate of pixel v1, and y2 is the y-coordinate of pixel v2). yp is the y coordinate of pixel p.
- Assume colour is greyscale or else we do it three times …

$$Ip = \frac{yp - y2}{y1 - y2} I_1 + \frac{y1 - yp}{y1 - y2} I_2$$

Gouraud Shading



Suppose,

I1 = 50
I2 = 211

Y1 = 20
Y2 = 9

YP = 15

What's IP?

Ans:

On next slide!

Gouraud Shading

- Use the formula presented already …

$$Ip = \frac{yp - y2}{y1 - y2} I_1 + \frac{y1 - yp}{y1 - y2} I_2$$

$$Ip = \frac{15-9}{20-9} 50 + \frac{20-15}{20-9} 211$$

$$Ip = 27.27 + 95.90$$

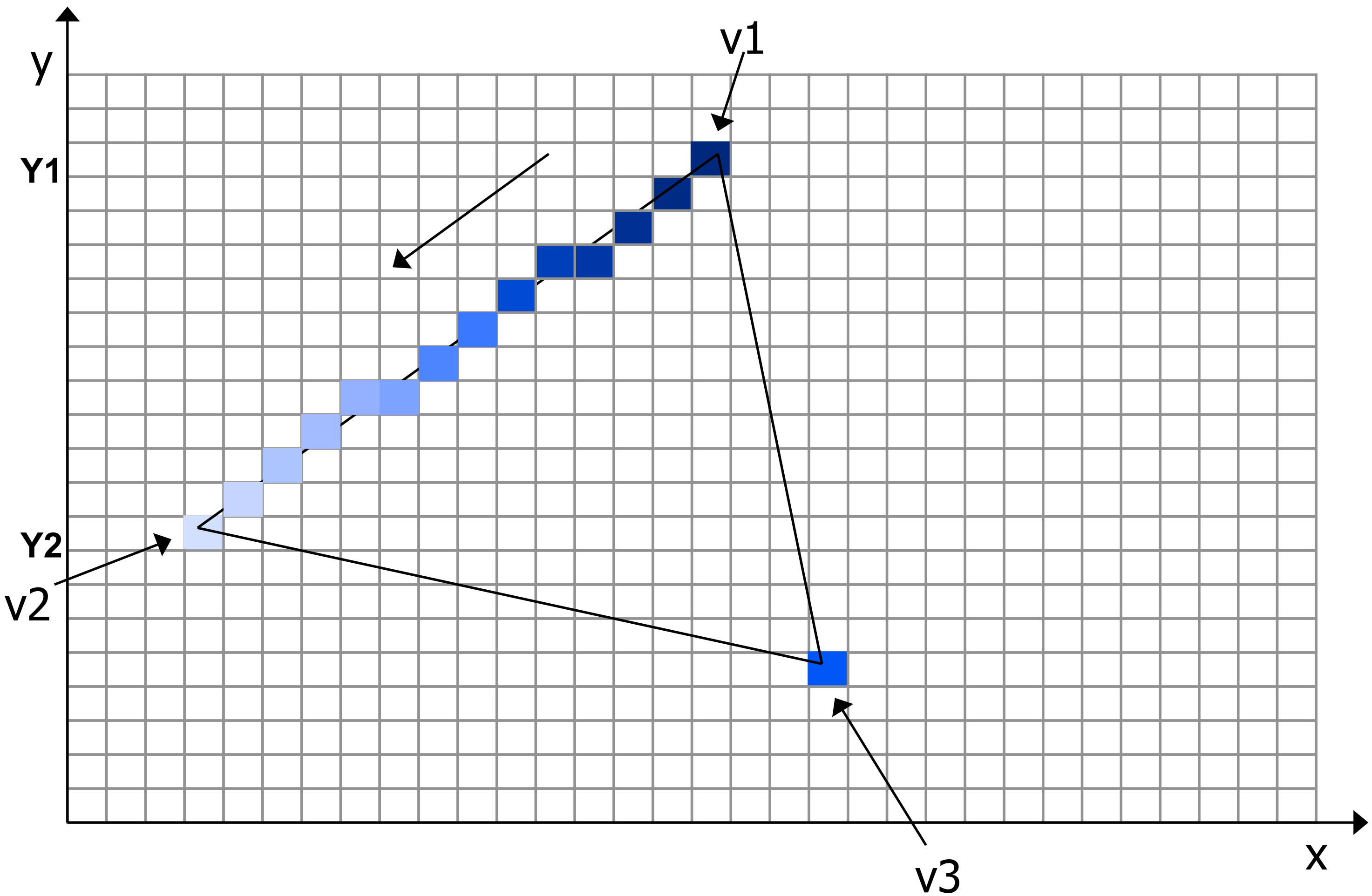
$$Ip = 123.17$$

$$Ip = 123$$

- Does 123 seem like the right colour value for a pixel roughly half-way between one of 50 and one of 211?

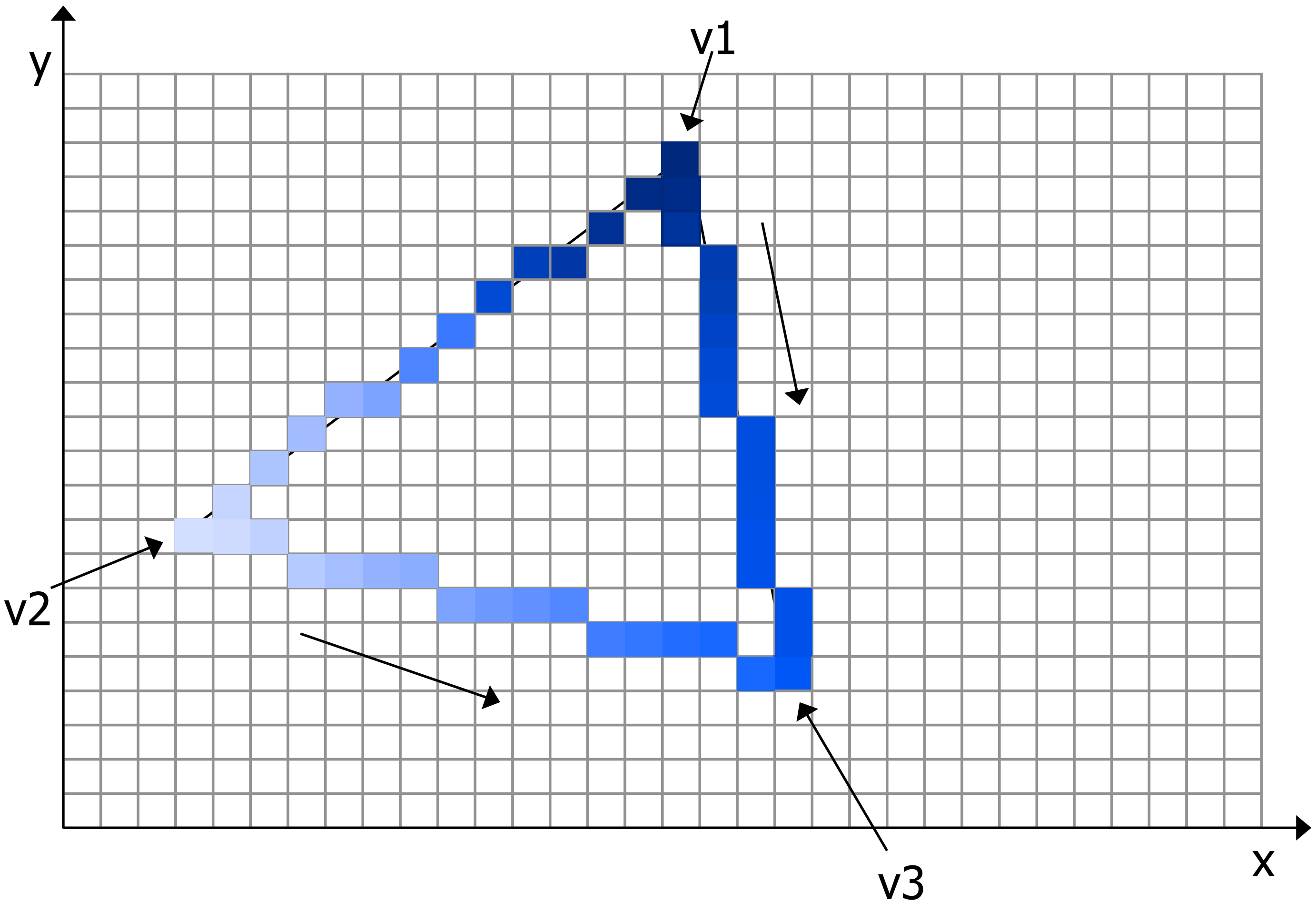
Gouraud Shading

- Now carry out this process for all pixels along the edge …



Gouraud Shading

- And then do the same for the other edges as well.



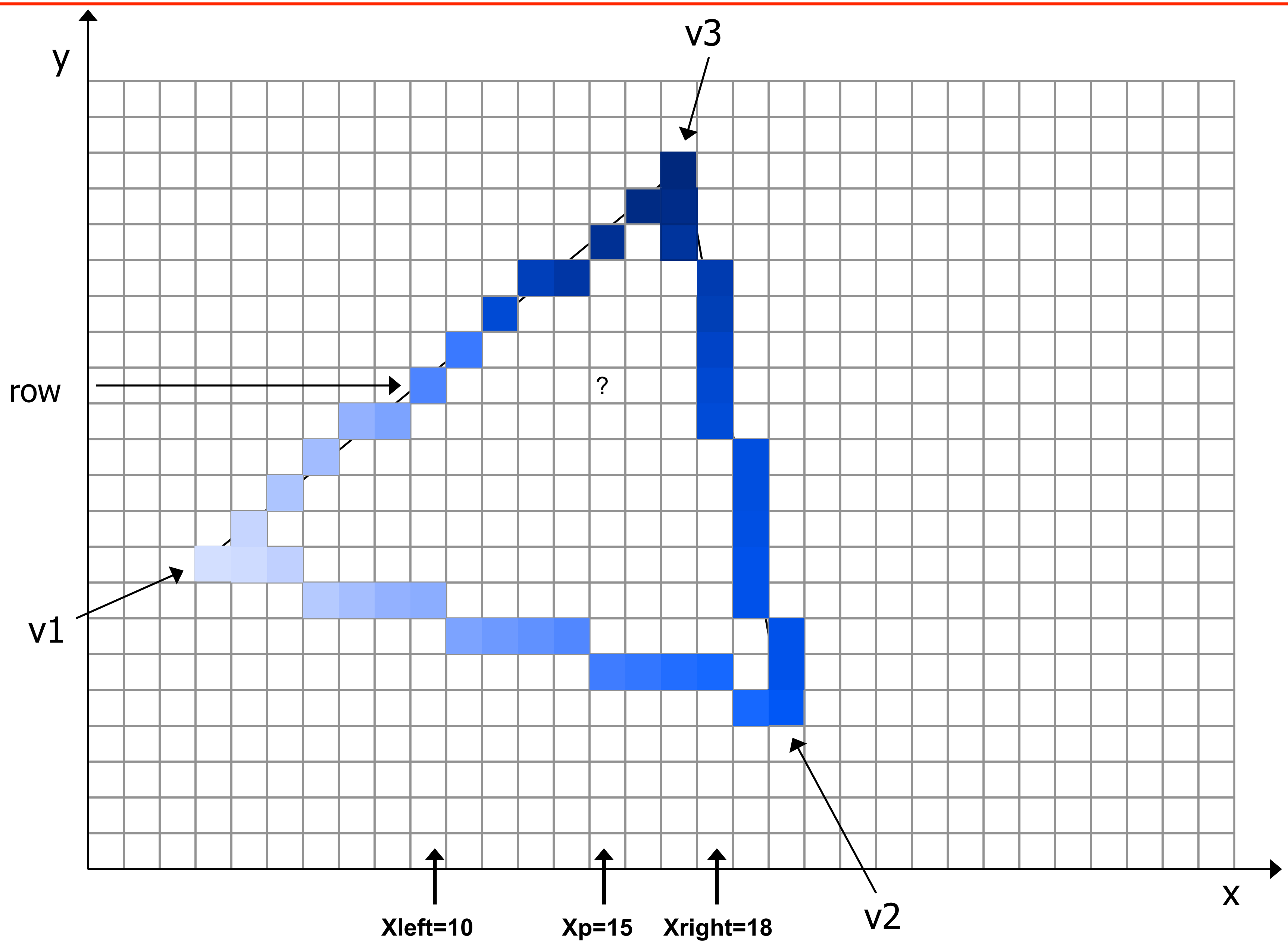
Gouraud Shading

- The triangle is then filled on a row by row basis starting at the top.
- We now have colour values for each end of the row so we can interpolate to get colour values along each row.
- The formula for this is:

$$I_p = \frac{xright - xp}{x_{right} - x_{left}} I_{left} + \frac{xp - x_{left}}{x_{right} - x_{left}} I_{right}$$

- Where xleft is the x coordinate of the point at the left end of the row and xright is the x coordinate of the point at the right end of the row.
- Xp is then x coordinate of the point we are calculating a colour value for.
- Ileft and Iright are the colour values at each end of the row

Gouraud Shading



Gouraud Shading

- Now suppose that Ileft (colour of left pixel) is 123 and Iright (colour of right pixel) is 200 then …

$$I_p = \frac{x_{right} - xp}{x_{right} - x_{left}} I_{left} + \frac{xp - x_{left}}{x_{right} - x_{left}} I_{right}$$

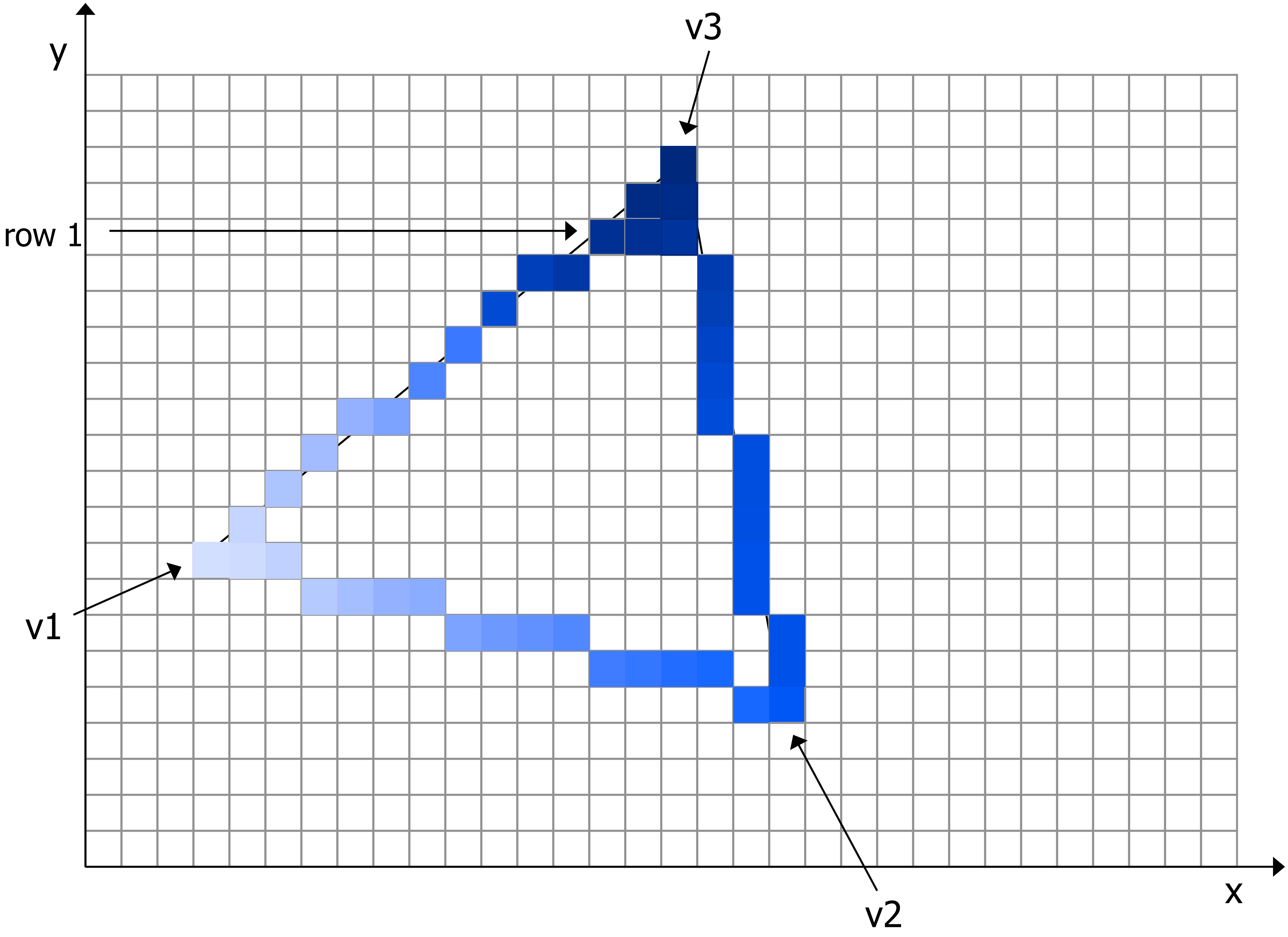
$$I_p = \frac{18-15}{18-10} 123 + \frac{15-10}{18-10} 200$$

$$I_p = \frac{3}{8} 123 + \frac{5}{8} 200$$

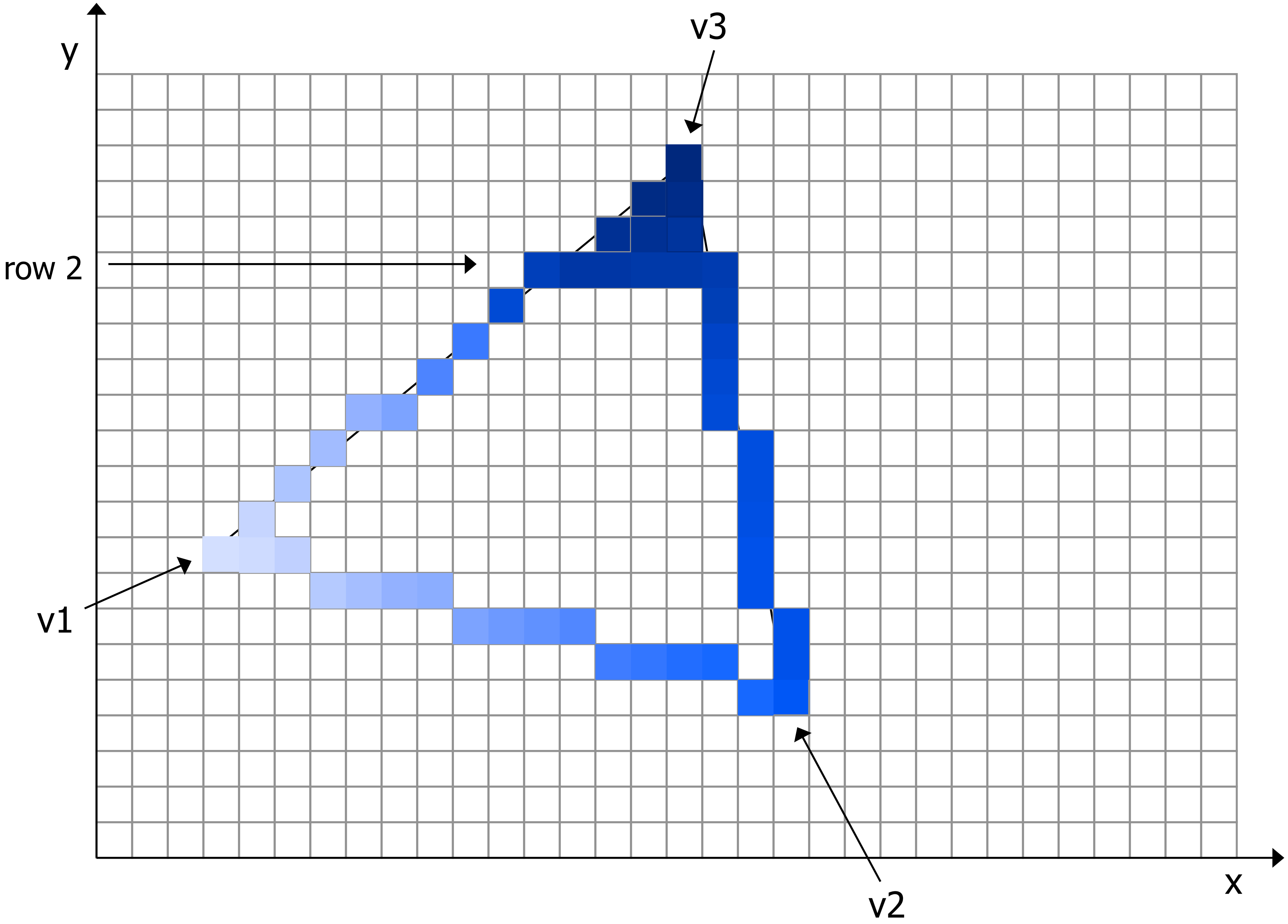
$$I_p = 46.125 + 125 = 171$$

- Do this row by row and pixel by pixel for each row.

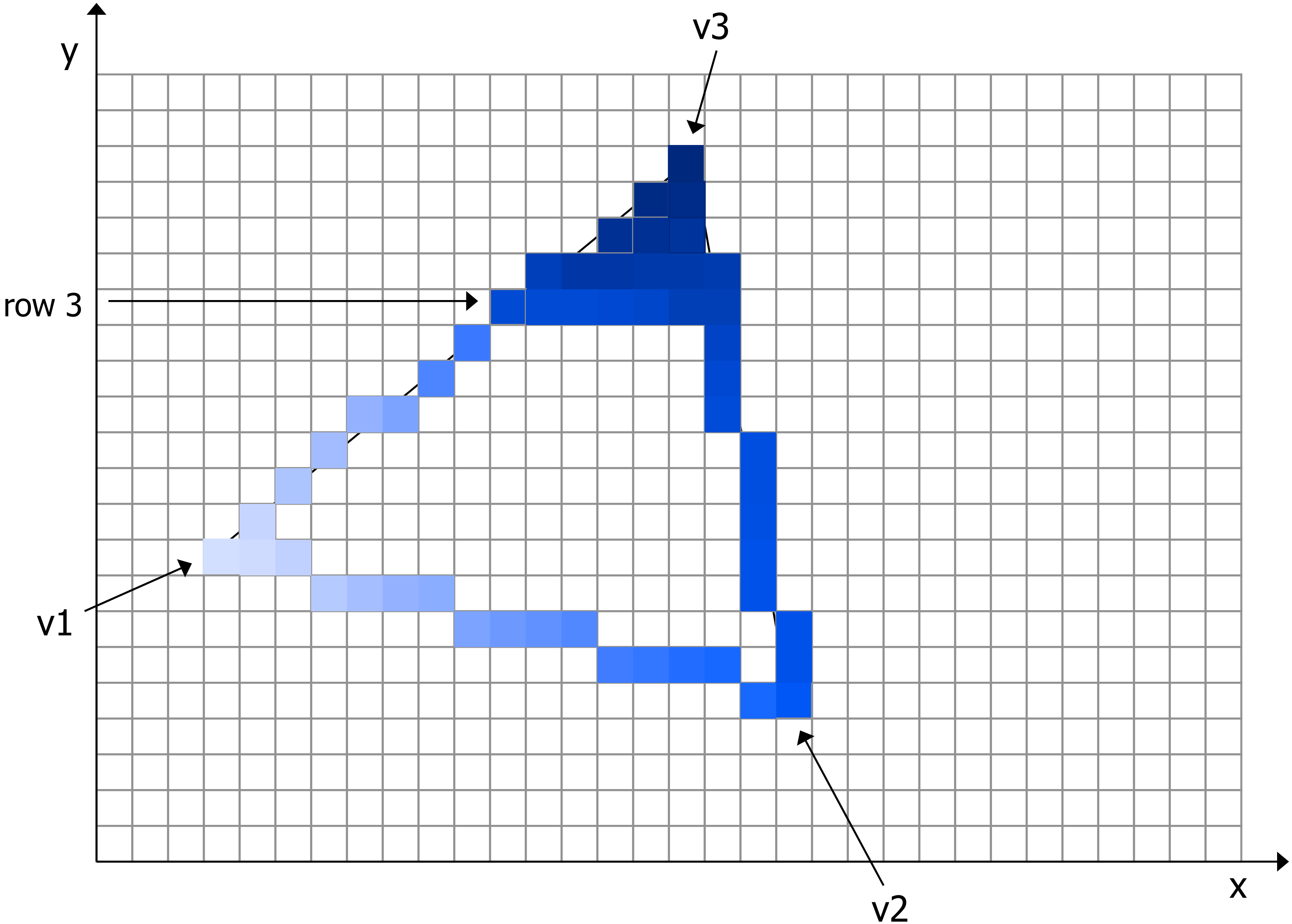
Gouraud Shading



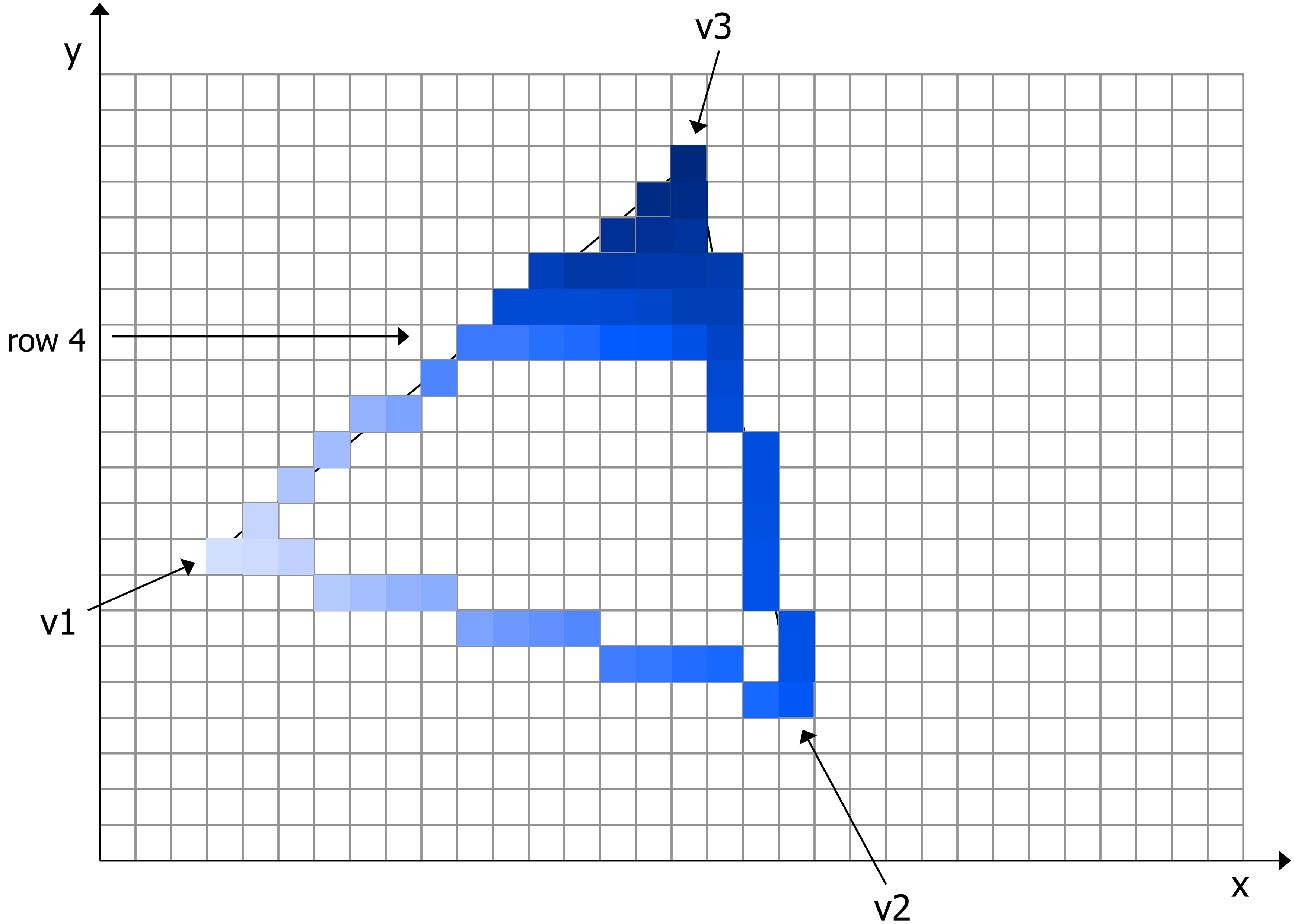
Gouraud Shading



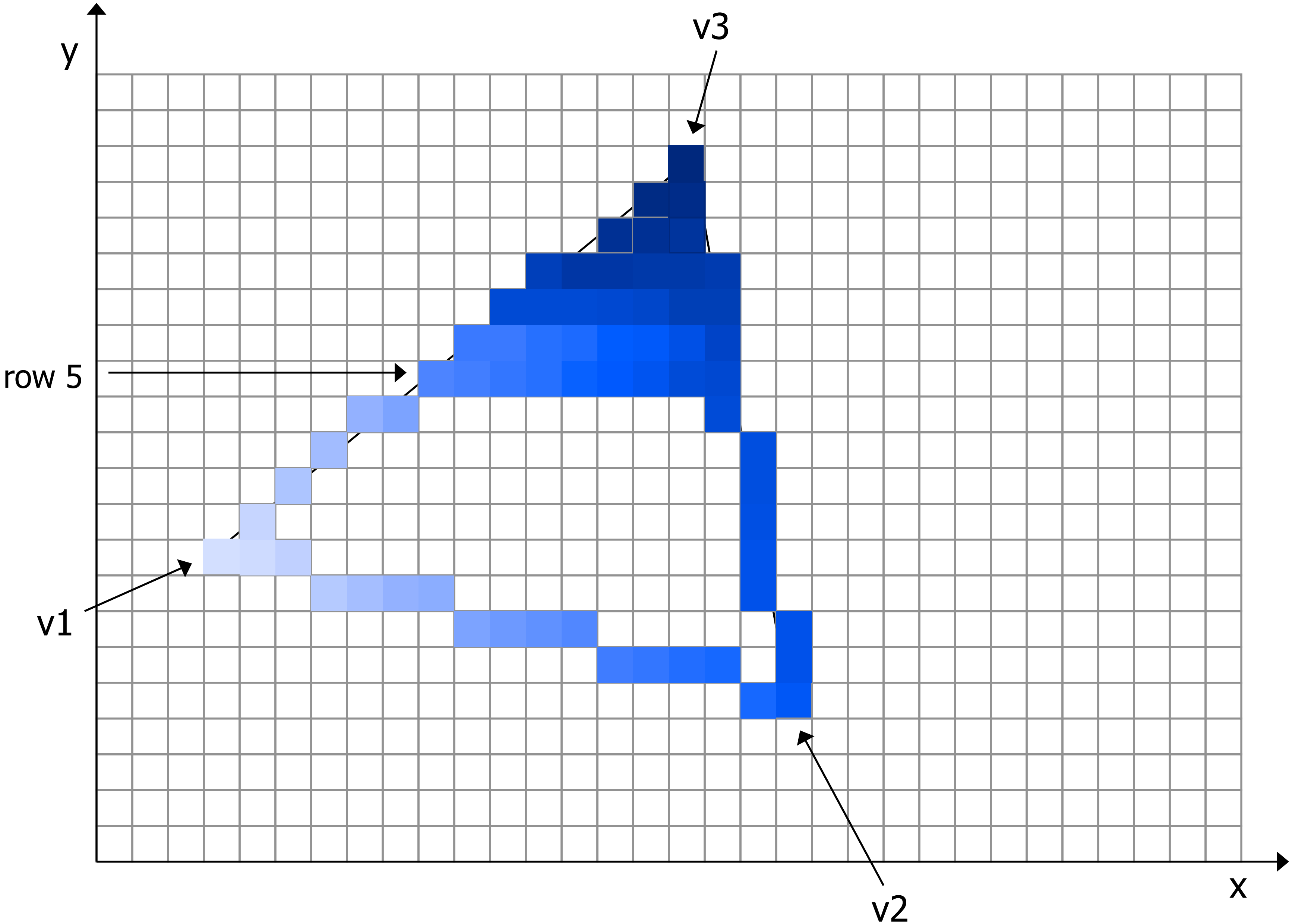
Gouraud Shading



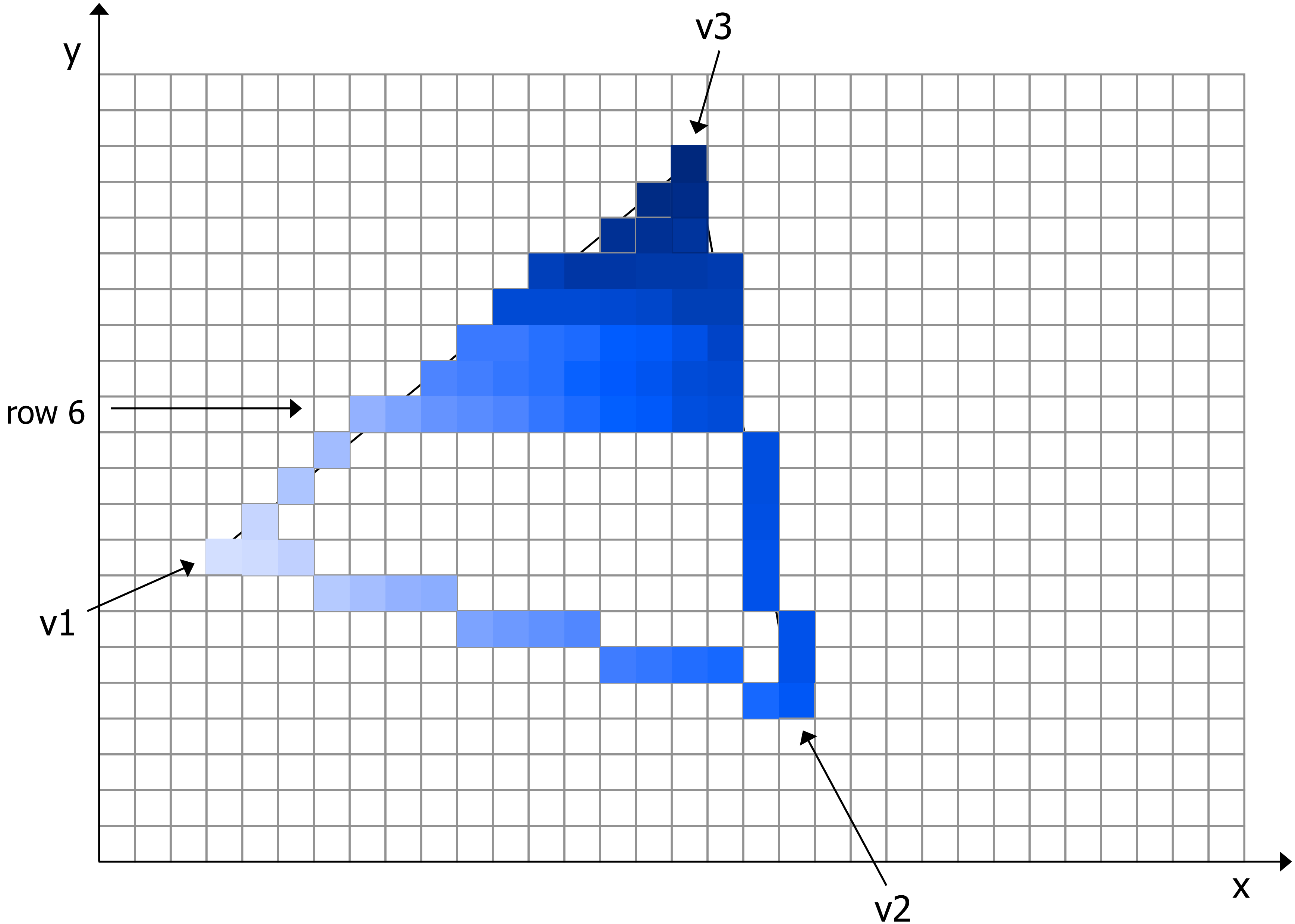
Gouraud Shading



Gouraud Shading

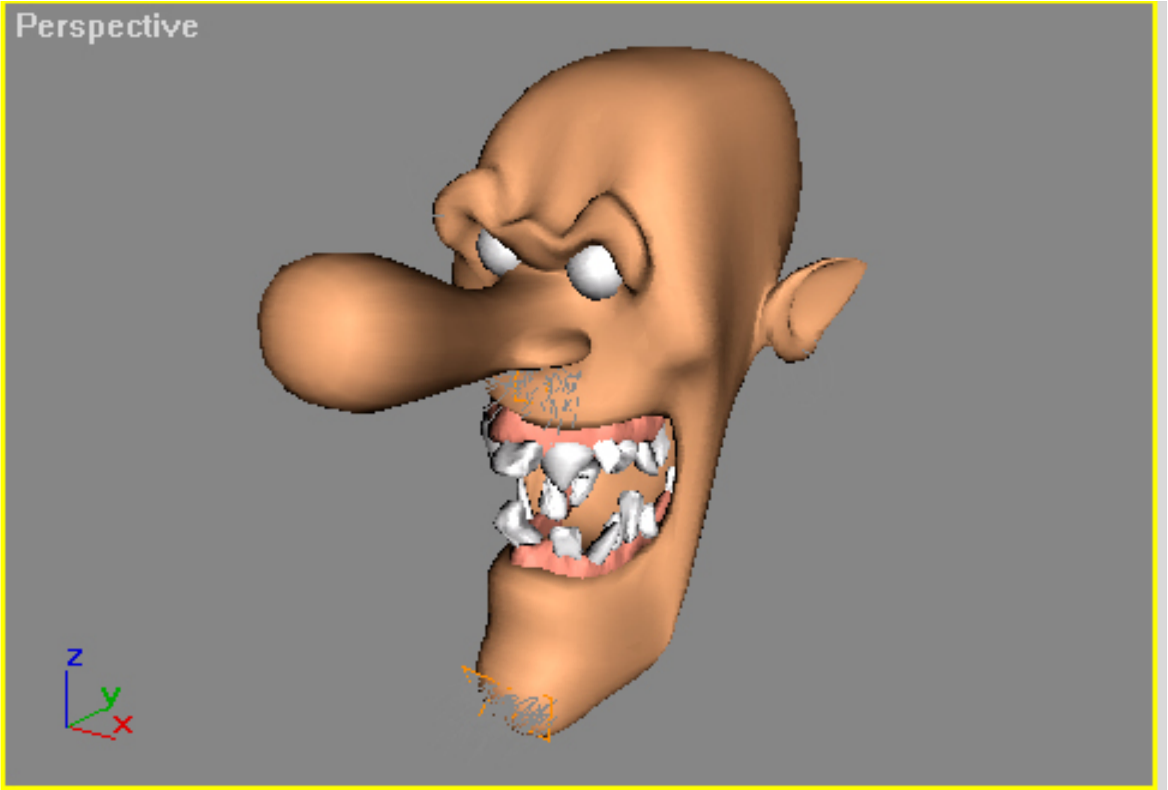


Gouraud Shading



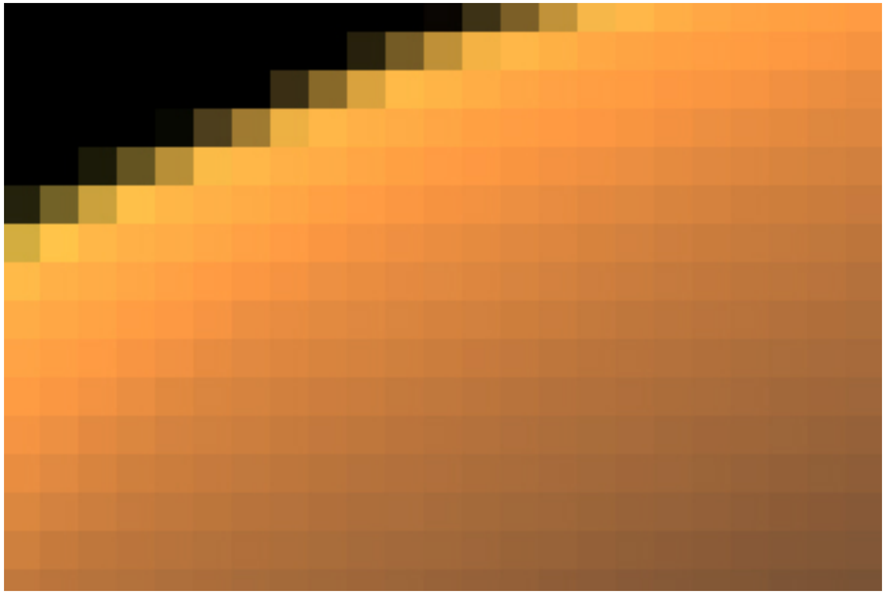
Gouraud Shading

- Process continues until the whole triangle is filled in and then …
- Do the next polygon until the whole object is done.
- Results below ..



Gouraud Shading

- Picture below is a close-up of part of the previous image. It shows clearly the Gourand shading on a pixel by pixel basis.



Introduction to Texturing

- Texture mapping greatly **enhances the realism** and the visual interest of Phong shaded scenes.
- It is relatively **cheap** in computational terms.
- Texture mapping leads to the use of **environment mapping** which allows us **render objects that reflect mirror like images** of their surrounding relatively easily.
- The general idea of texturing is that we use some sort of **map to vary and control the values of the diffuse colour components** across a surface on a pixel by pixel basis.
- This map could, for example, be a 2D bitmap of a wood texture (as you will see in the labs)
- As rendering proceeds, values are picked up for the Phong diffuse reflection co-efficients and the **diffuse component (the colour) of the shading changes as a function of the texture map**.
- See next slide.

Introduction to Texturing

2D Bitmap (created in Photoshop)

Texture Mapped onto teapot

- So effectively we **replace the constant diffuse colour value** across the surface with a **different diffuse colour value for each point**.
- These are picked directly from the bitmap.

Introduction to Texturing

- So in the previous example we are **modulating** the diffuse colour value **according to some value in a map**.
- However there are a number of properties of a surface we can modulate in this way.

1. Diffuse Colour

Most common object property that can be **controlled by a texture map**. This is what most people mean when they mention texture mapping.

2. Specular Colour

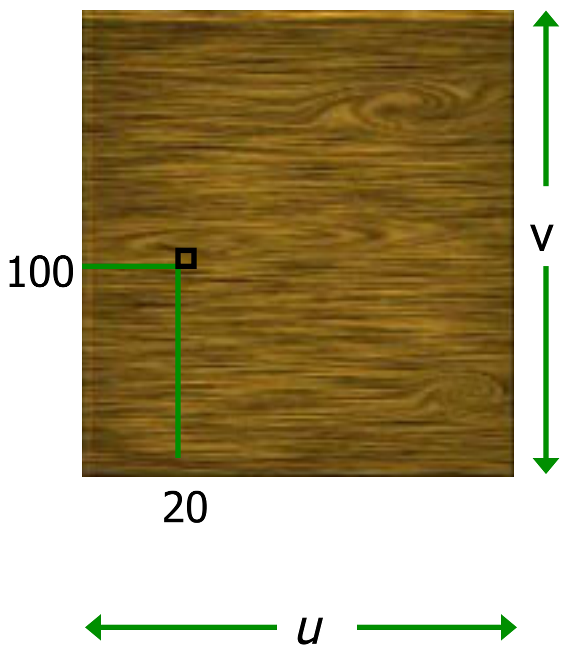
This technique is known as environment mapping or chrome mapping and allows us to use **texture map techniques** to create objects that **reflect their environments.**

3. Normal Vector

If we **modulate the normal vector** across a surface we arrive at a technique called bump mapping which allows us to **simulate uneven, 'bumpy' surfaces.**

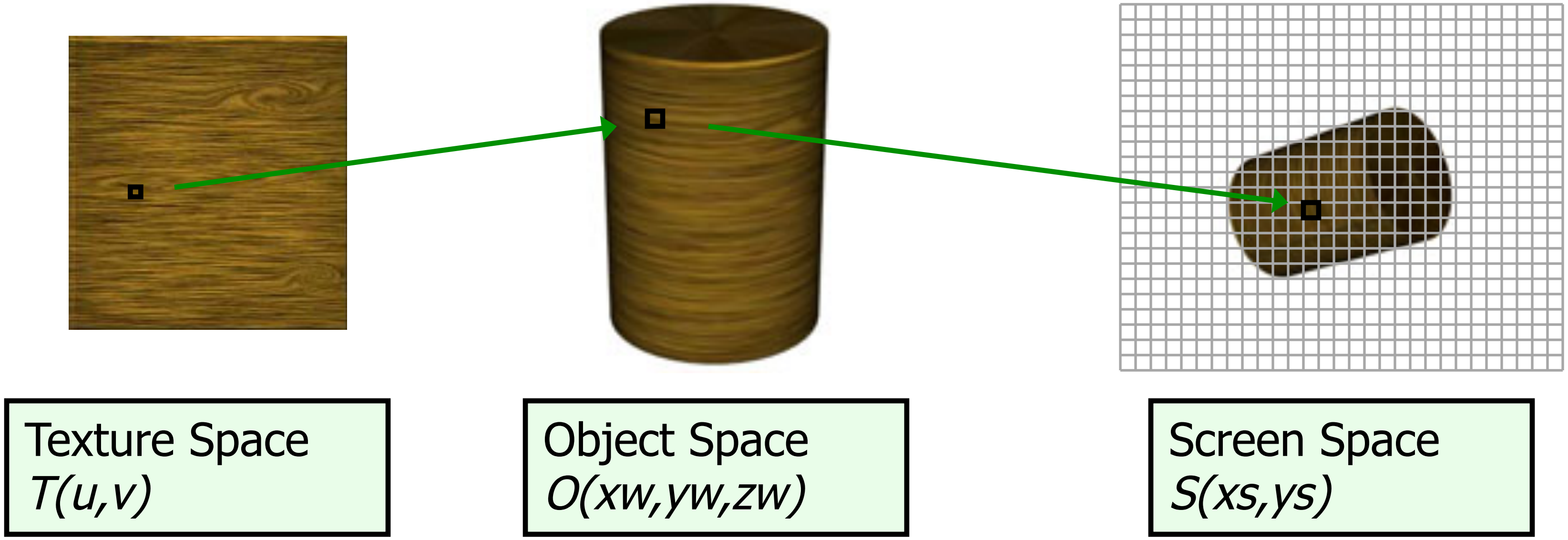
The 2D to 3D Problem

- The texture mapping process involves the following.
- We have some sort of 2D bitmap. This is known as the **texture space** and can be indexed with **two parameters** (call them u and v).



- Texture Space *T(u,v)*
- Suppose this is a 200x300 bitmap.
- Then T(20,100) is a RGB value derived from the bitmap.
- This then get's **mapped** onto a point on the **surface of the object**.
- See next slide.

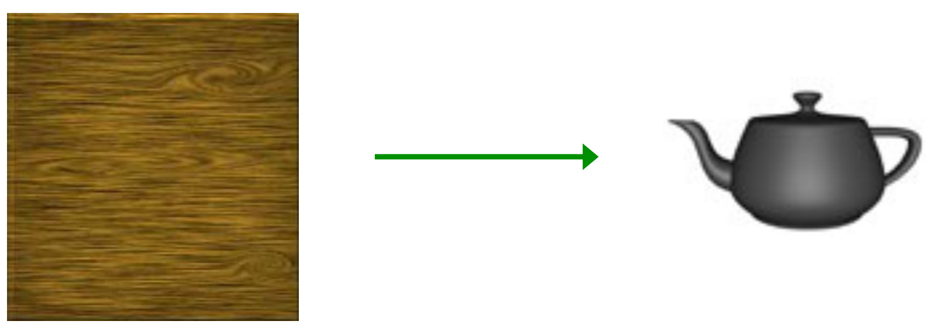
The 2D to 3D Problem



- So a colour value from *T(u,v)* gets **mapped onto a point on the surface of the 3D object** *O(xw,yw,zw)* and then after it is rendered, becomes the **basis for the colour value for a pixel on the screen *S(xs,ys)*.**
- Tricky bit is how do we carry out the mapping from *T(u,v)* to *O(xw,yw,zw)* ?
- How do we associate values in the 2D texture map with points on the 3D surface?
- How would you wallpaper a sofa?

Two Part Mapping

- **Two-part texture** mapping is a common technique that overcomes the problem of mapping 2D textures to 3D objects by using an **‘easy’ intermediate 3D surface** onto which the **texture is initially projected**.
- Consider trying to carry out the following mapping. Impossible!



Texture Space
T(u,v)

Object Space
O(xw,yw,zw)

Two Part Mapping

- A two-part mapping might use a cylinder as an intermediate surface as shown below.

Texture Space
T(u,v)

Intermediate Surface
T(u,v)->T'(xi,yi,zi)

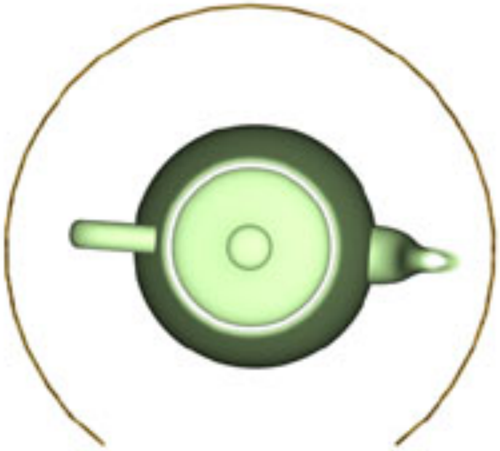
Mapping to Object
T'(xi,yi,zi)->O(xw,yw,zw)

Two Part Mapping

- The first mapping is from the **texture space to the intermediate surface**.
  - *T(u,v) -> T'(xi,yi,zi)*
- This is known as the **S mapping**.
- If we pick a **sensible** intermediate surface this should be a **simple mapping**.
- For example the inside of a cylinder can be **thought of as a flat plane** that has been **bent around**. Any **point on the cylinder** could be described in terms of a **2D coordinate** and hence we can **map directly from the texture space.**
- The four common intermediate surfaces that are used are:
  - Cylinder
  - Plane
  - Sphere
  - Box

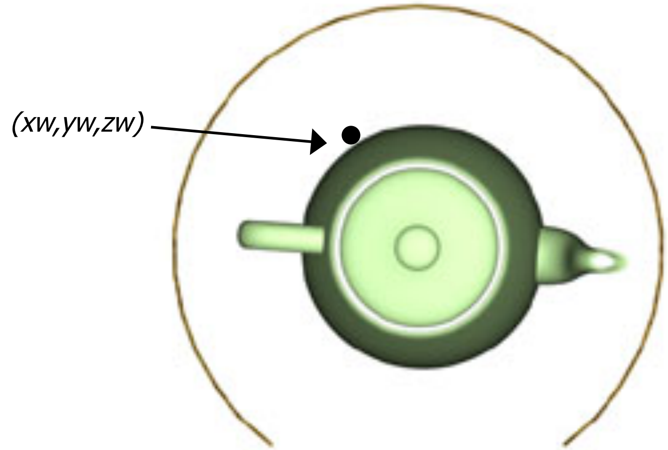
Two Part Mapping

- The second mapping is from the **intermediate surface to the object itself**.
  – *T'(xi,yi,zi) -> O(xw,yw,zw)*
- This is known as the **O mapping**. How is this accomplished?
- There are **four basic possibilities** here.
- We will look at each in turn and use the **example of a cylinder as the intermediate surface**.
- The diagram below **shows the situation** as if we were looking down on the cylinder with the object positioned inside it.



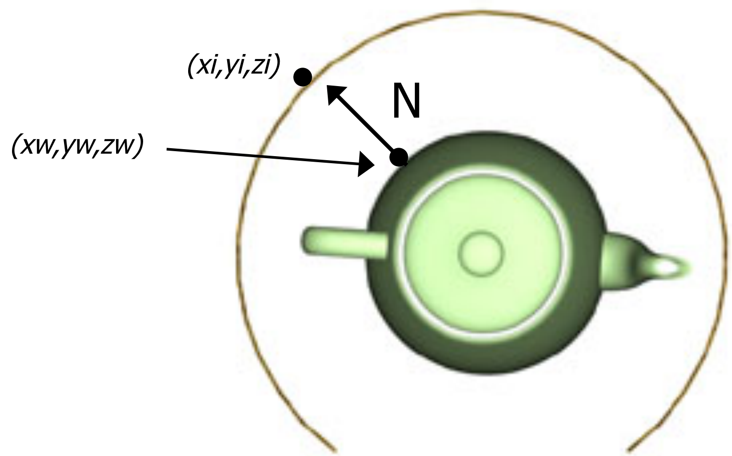
Two Part Mapping

- Basic problem is we have a point on the object **(xw,yw,zw) and we want to know what texture point on the intermediate surface (xi,yi,zi)** to map onto it.



O Mapping 1

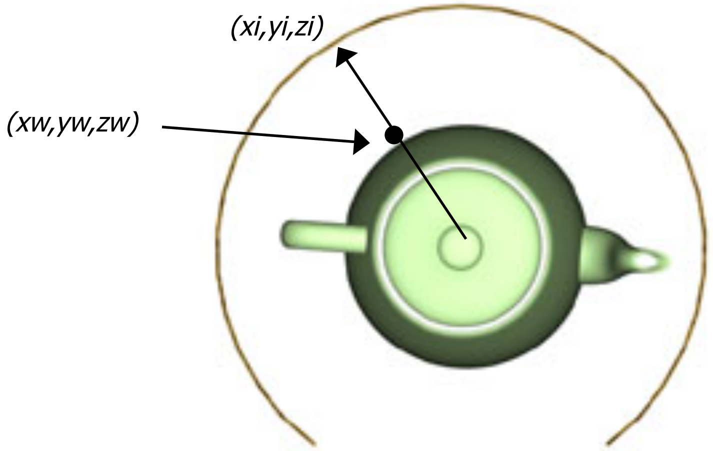
- Get the **surface normal at (xw,yw,zw)** and use the texure point where it **intersects with the inside of the cylinder**.



Two Part Mapping

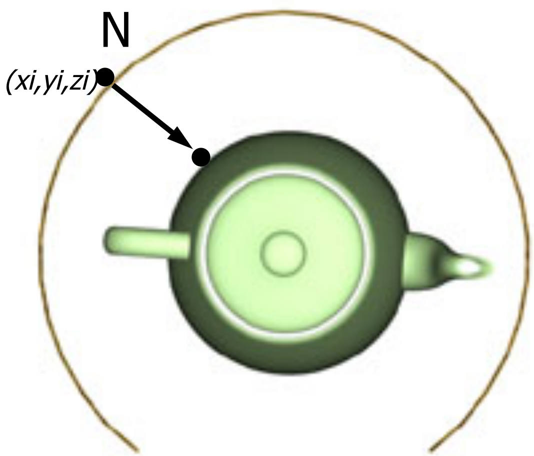
O Mapping 2

- Get the **line from the object centroid to (xw,yw,zw)** and use the texture point **where it intersects** with the inside of the cylinder.



O Mapping 3

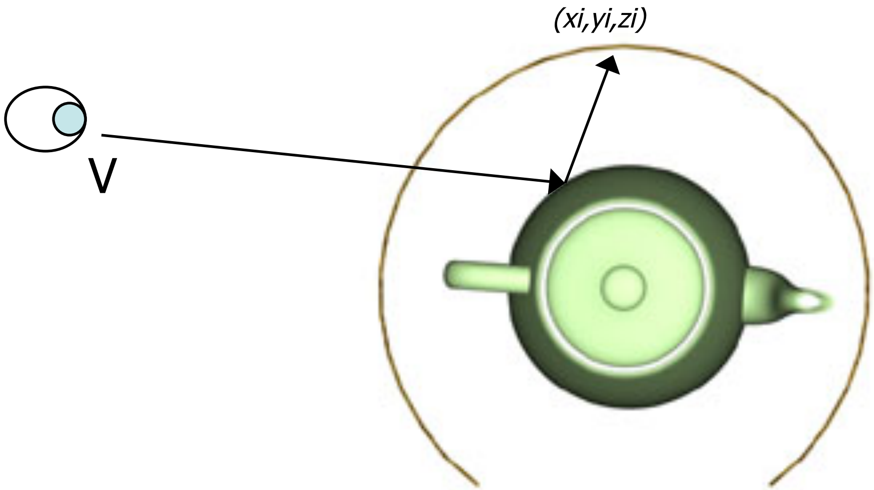
- Use the **surface normal from the inside of the cylinder** (i.e. the inverse of O Mapping 1).



Two Part Mapping

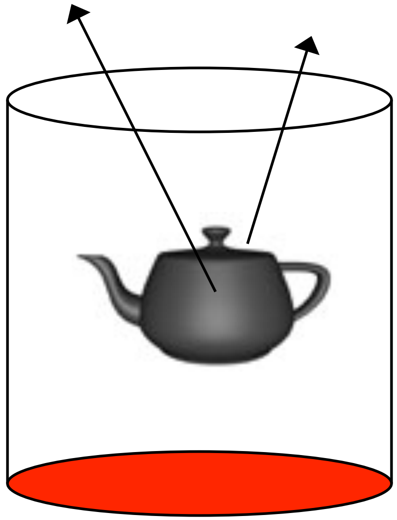
O Mapping 4

- This is a special case.
- Use the **reflection of the viewing direction** off the point and calculate where it **intersects with the intermediate surface**.
- This leads us to a technique called **environment mapping** that we will discuss later in the lecture.



Two Part Mapping

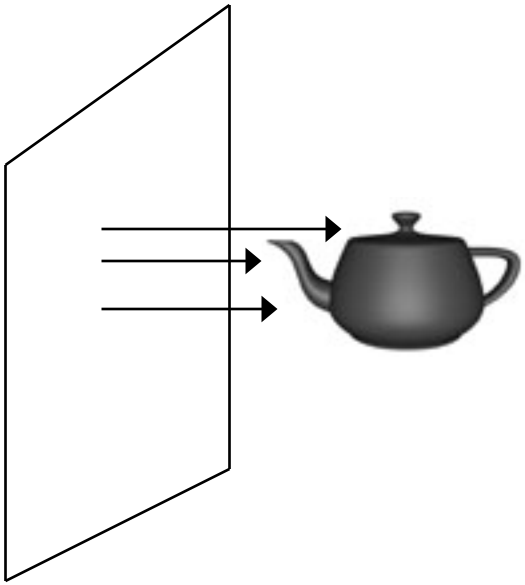
- Some of the four mappings we have described will not work very well with some intermediate surfaces.
- Suppose we are using the **cylinder that is open at the top**. Then all of the O mappings might **fail to find texture coordinates for points along the top** of the teapot.



- We won’t have such problems with a **sphere**, or indeed a completely closed cylinder. A box is also completely enclosed but using a **box leads to discontinuity problems** because of its sharp corners.

Two Part Mapping

- What about a plane?
- The only mapping procedure which is really suitable for a plane intermediate surface is **number 3** (useing the plane surface normal).



- The effect of this would be a bit like using a slide projector to **project an image onto the surface of an object**.

Two Part Mapping

Cylindrical

Two Part Mapping

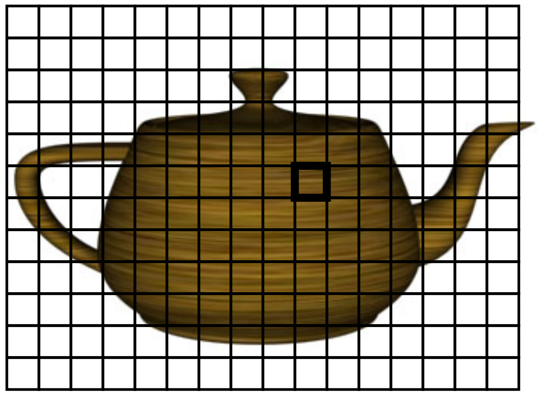
Spherical

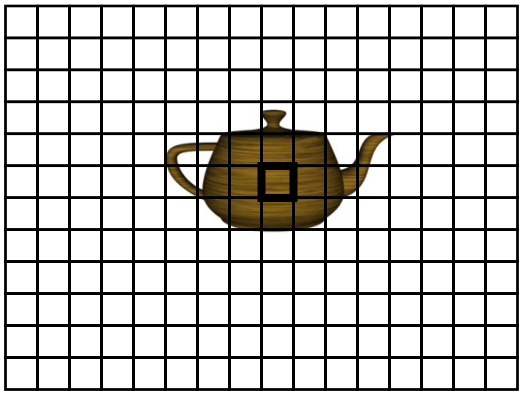
Two Part Mapping

Box

Texture Filtering

- There is a remaining problem with this however.
- A pixel on the screen does not necessarily directly **correspond to a texel** in the texture map.
- Why? Since the textured surface can be at an any distance or orientation with respect to the viewer, one pixel does not directly correspond to one texel.



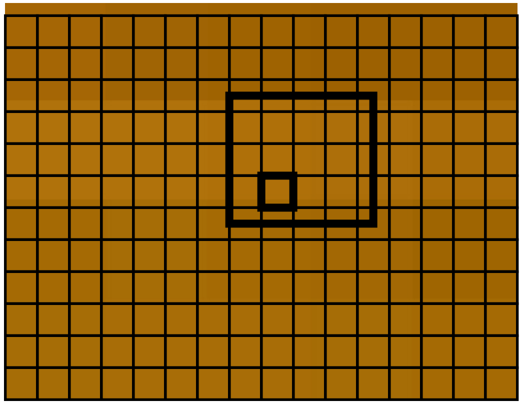


Texture Filtering

- So for example consider the pixel highlighted in the two pictures on the previous slide.
- On the right hand version the viewer has zoomed away from the teapot.
- Hence the area of the texture map covered by the pixel is much larger than on the left.
- In both of these cases it is obvious that one pixel corresponds to many texels in the texture map.
- **Only one colour can be used for the pixel**, so the area of **texels has to averaged or filtered** in some way to get the most representative colour for the pixel.

Texture Filtering

- This is known as **texture minification.**
- **Texture magnification happens** in the opposite case.
- Suppose we are zoomed in close to an object and hence one texel value has to be applied to more than one pixel.
- This gives the close-up blocky effect …



Big square = texel
Small square = pixel

Texture Filtering

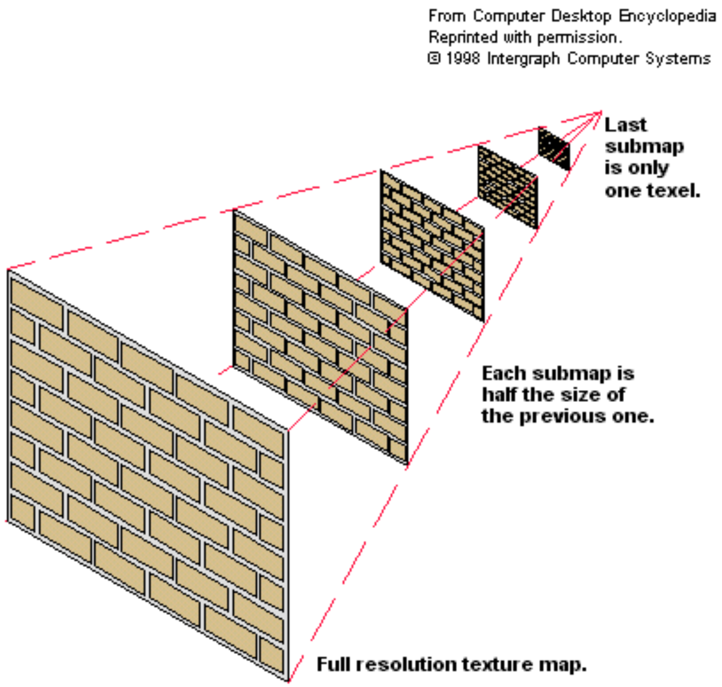
- One problem with **texture minification** is that a lot of processing can be involved.
- Suppose for example the object to which the texture is mapped, is far away, and covers only a few pixels of screen space.
- Even worse it is possible that the **entire texture could take up just one pixel of screen space**.
- In this case the system is going to have to read all of the texels and combine their values with some filtering operation.

Texture Filtering

- This is hugely inefficient and can be avoided by using a process called **mipmapping.**
- This involves **pre-filtering the texture** and storing successively smaller and smaller versions of it, all the way down to one pixel in size.
- As the textured surface moves further away the system switches to a smaller version of it.
- Filtering still has to be employed but since it is always working with a version of the texture which **roughly corresponds to the screen size** required it is nowhere near as computationally expensive.

Texture Filtering

- Different sizes of the **mipmap are known as levels**.
- If the texture has a basic size of 256 by 256 pixels then the associated mipmap set may contain a series of 8 images, each one-fourth the size of the previous one.
- 128×128, 64×64, 32×32, 16×16, 8×8, 4×4, 2×2, 1×1 (a single pixel)

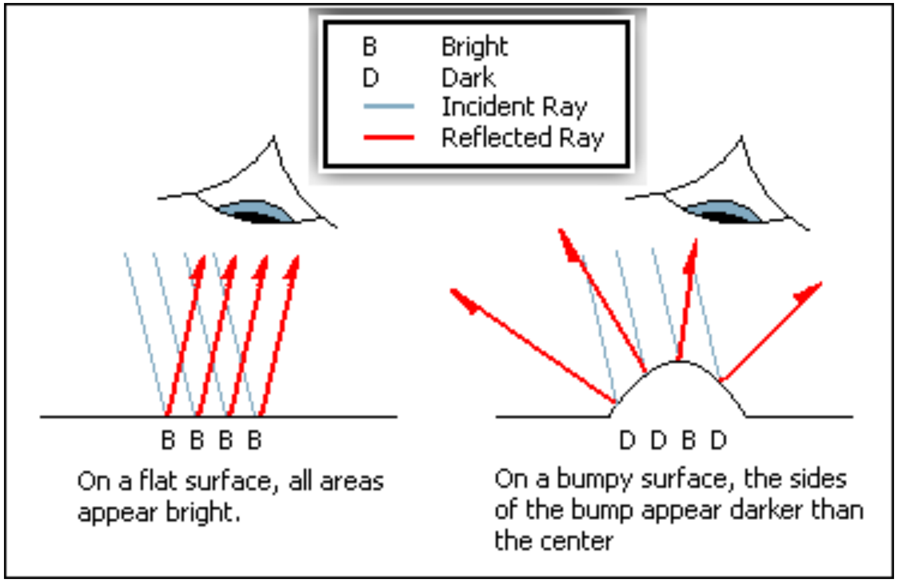


Bump Mapping

- At the start of the lecture we mentioned that we can use a **texture map** to **modulate more than just the diffuse colour** value of the surface.
- One of the alternatives is to use it to **modulate the normal vector**.
- This is known as **bump mapping** and enables a surface to appear as if it was **wrinkled or dimpled** without the need to **geometrically define these depressions in the surface**.
- Instead, the **surface normal at each point is angularly perturbed according to information in a two-dimensional texture map**, and this 'tricks' the **illumination model** into producing what looks like **geometric variations** across what was modeled as, a smooth surface.
- Why do we use it?
- Well **to model tiny bumps and dips in a surface** with polygons would mean a **massive increase in the amount of polygons**. Too much processing would then be involved.

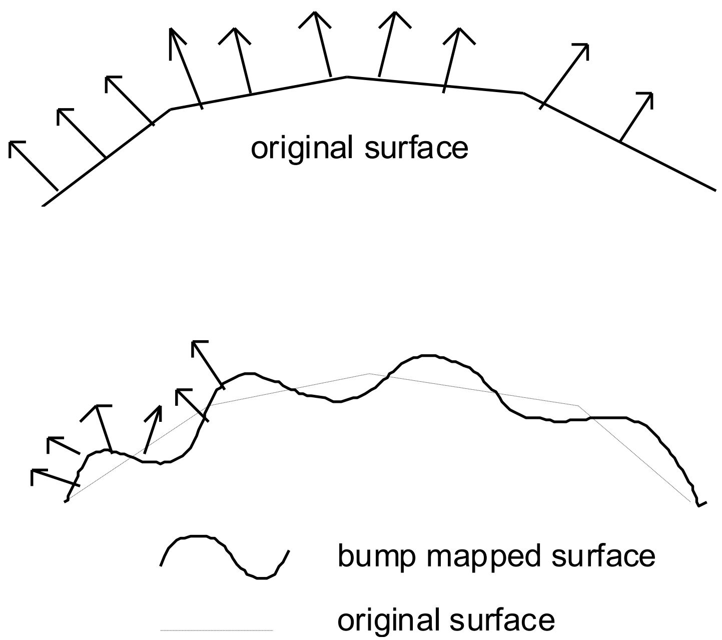
Bump Mapping

- It works because the **illumination model** (e.g. Phong) computes a value **based on the normal vector**.
- If it is given a sequence of normal vectors across a surface **that are all the same**, it will **give back the same colour value** for the points across the surface.
- If however, it is given a **sequence of normal vectors** which have been **angularly perturbed**, it will return **variations in the colour** value that look like **bumps**.



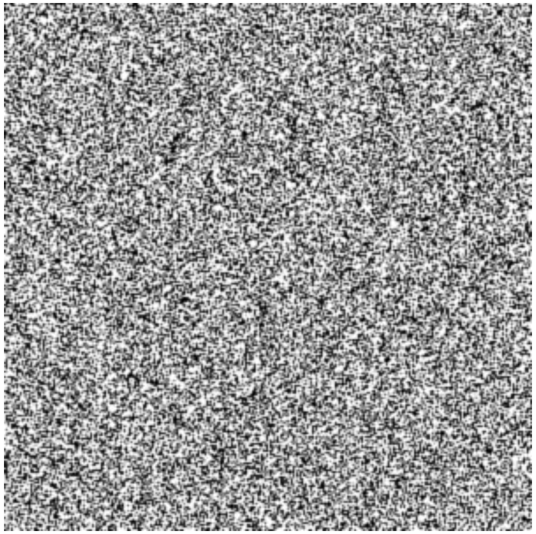
Bump Mapping

- The diagrams on the right show a side view of a polygonal surface with normal vectors pointing in the directions we would expect.
- Under this we have a graph **representation of a bump map**. The **value on the y axis** would **indicate the amount** that the **normal vector to the corresponding point on the object would be perturbed**.
- In effect this leaves us with a **'bumpy' surface** as shown at the bottom.



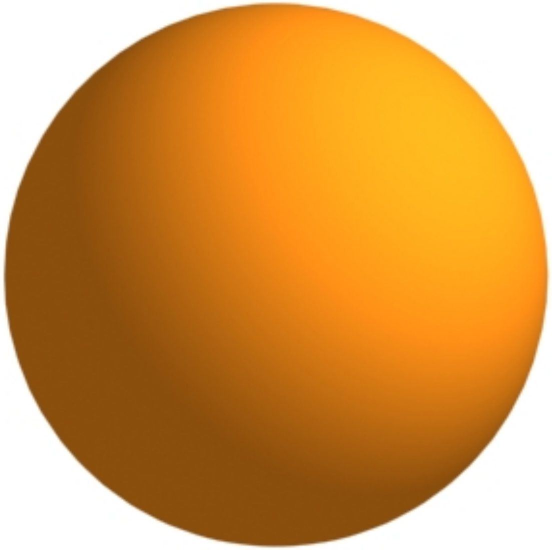
Bump Mapping

- Let’s look at some examples.
- The following 300x300 bitmap was created in Photoshop by creating a white bitmap and then using the Add Noise filter on it.



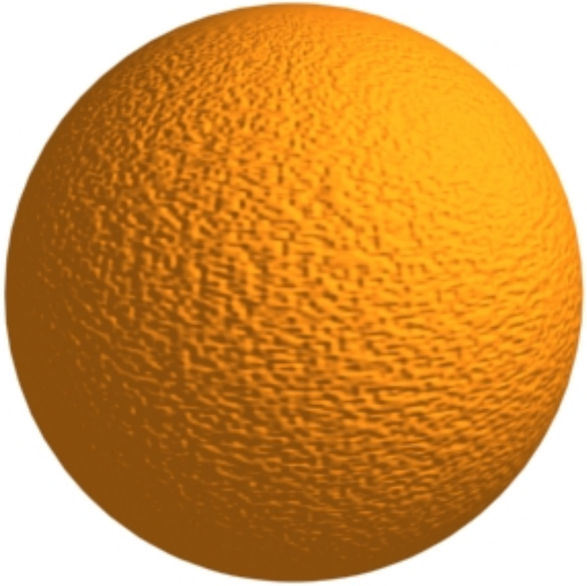
Bump Mapping

- An smooth orange sphere, when rendered, may look like this.



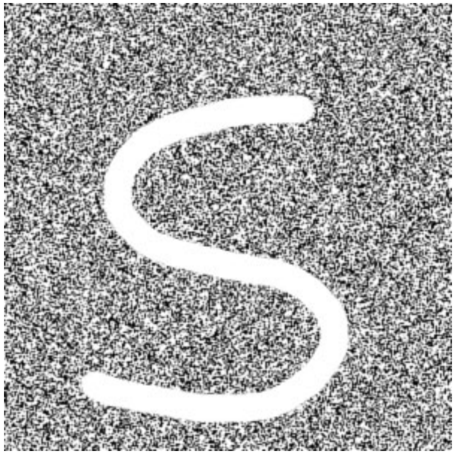
Bump Mapping

- After applying the bump map from two slides ago we get:

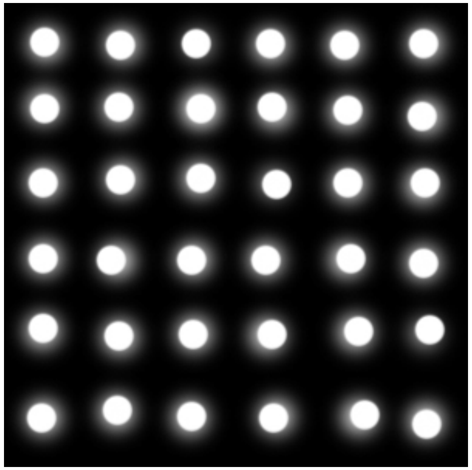


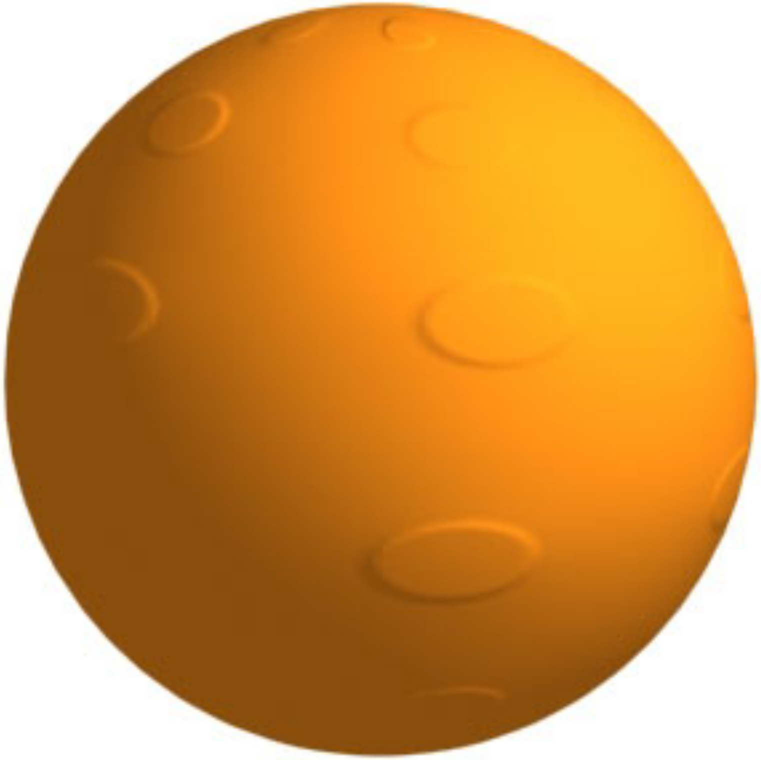
- The following slides have some more example along with the bump maps used to create them:

Bump Mapping

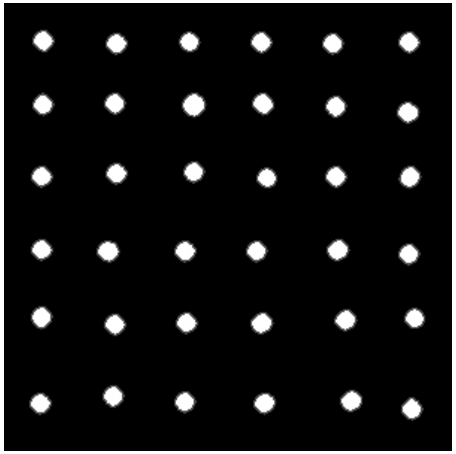


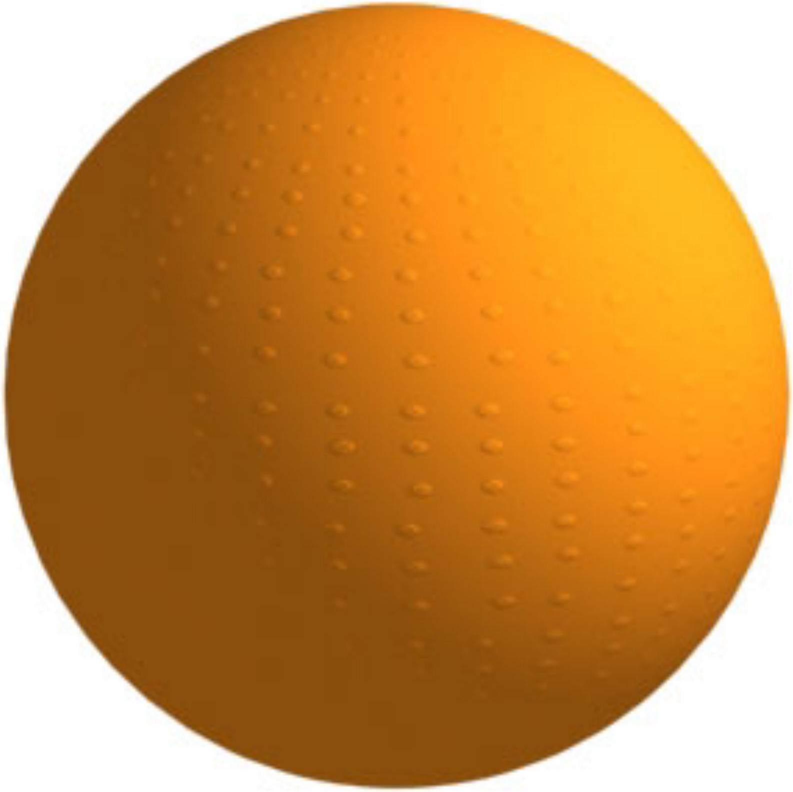
Bump Mapping





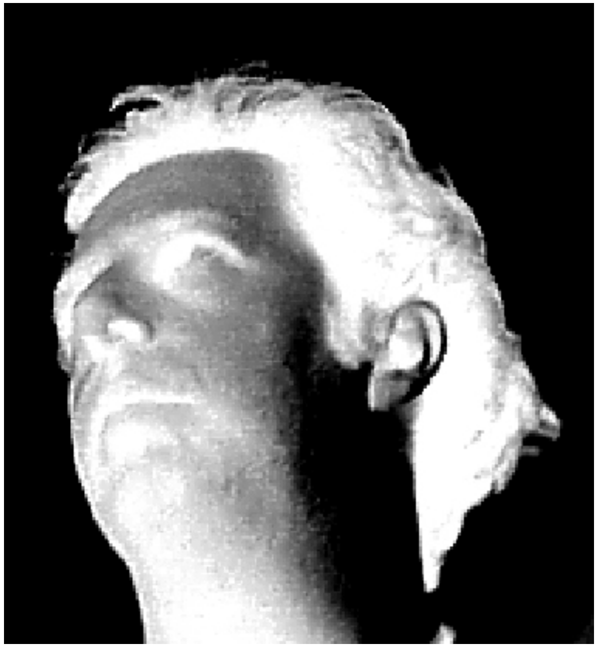
Bump Mapping

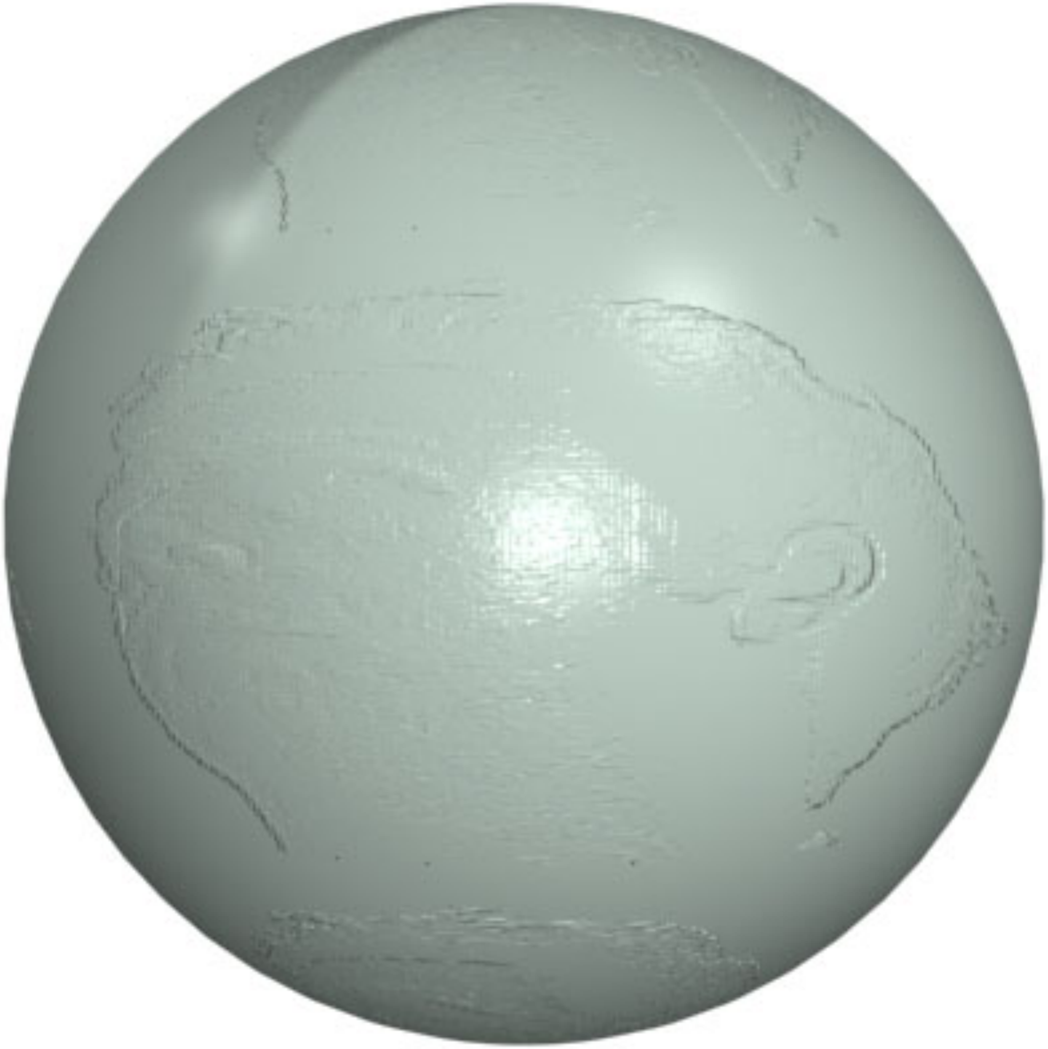




Bump Mapping

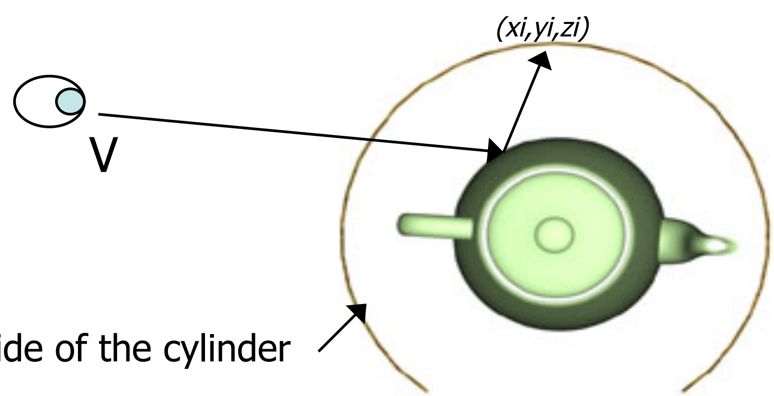
Bump Mapping





Environment Mapping

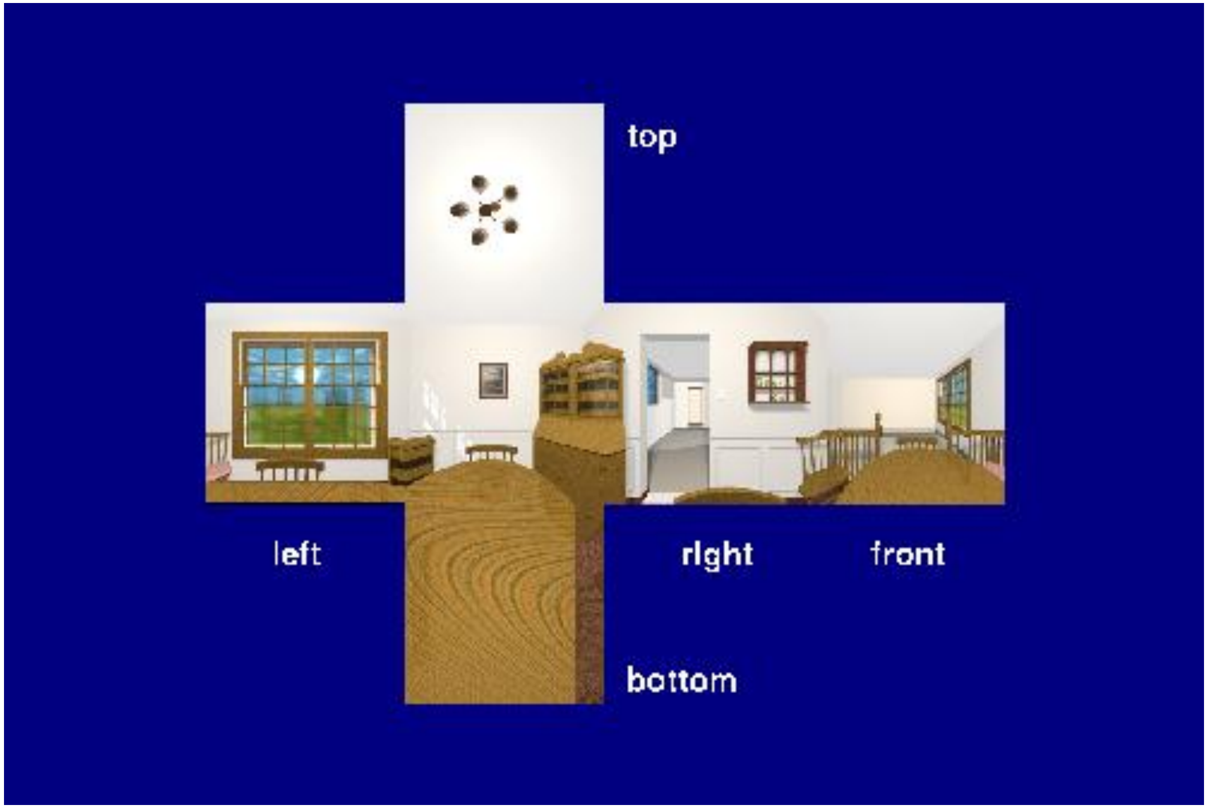
- The other possibility we mentioned at the start was setting the **specular colour according to a texture map**.
- Environment mapping uses this idea to **reflect a surrounding environment** in a shiny object.
- The idea is that a **shiny object reflects its surrounding** (or environment) and if a **bitmap of this is pre-stored as a texture map**, **then texture mapping will create the desired effect** for us.
- It uses the form of two-part mapping whereby the **O mapping uses the reflection of the view vector**.



Environment Mapping

- This **texture value** is then **used for the colour of the specular reflection**. Results in a mirror like effect.
- Because the **value picked from the environment map is dependent on the view direction**, we get a **different reflection from different viewpoints**.
- **Often the approach taken is to use a box** as the intermediate surface.
- The maps for the inside of the box might be constructed either by **taking six photographs of a room interior**, or **rendering 6 views of the computer graphics scene** from different viewpoints.
- Environment mapping is really a **cheap and cheerful means of carrying out ray tracing**.

Environment Mapping



Initial environment maps

Environment Mapping

Rendered object.