OpenGL Notes [a]

Stu Pomerantz

smp@psc.edu

`http://www.psc.edu/~smp`

November 10, 2004

[a]Most material is adapted from: OpenGL ARB, et. al, “The OpenGL Programming Guide”, Third Ed., Reading: Addison-Wesley, 1999

Blending (briefly)

- Conventially, we specify the color of a pixel by setting the amount of red, green, and blue for that pixel.
- A fourth quantity, alpha, is used to denote opacity.
- An alpha value of 0 means no opacity (or completely transparent). An alpha value of 1 means 100% opacity (or completely opaque).
- Blending must be enabled using `glEnable(GL_BLEND)` to inform OpenGL that it should pay attention to the alpha value.
- If blending is not enabled, the alpha value is ignored, and all colors are considered to be 100% opaque.

Blending (briefly)

- When blending pixels are classified in two ways
- The *destination* pixel is the pixel already stored in the frame buffer.
- The *source* pixel is the incoming pixel that's about to be blended with the destination pixel.
- Many possible functions can be used to combine the source and destination pixels into a new pixel in the frame buffer.

The Blending Function

Specify how the source and destination pixels are combined using:

```
void glBlendFunc( GLenum sfactor, GLenum dfactor );
```

- *sfactor* multiplies the source pixel.
- *dfactor* multiplies the destination pixel.
- There are many constants which can be used for sfactor and dfactor. Only a few are discussed here.
- In this context, combining pixels means adding them together. (see below)

Blending

Drawing order is important!

- Imagine drawing an sphere as seen through a (partially) transparent stained glass window.
- Suppose the window is drawn, followed by the sphere.
- Will the sphere be seen ? **No!**
- The window is closer to the eye than the sphere.
- When the sphere is drawn, a value closer to the eye than the sphere (the window) is already in the depth buffer, so the sphere is thrown away.

Blending

Approaches to coping with the drawing order problem:

- If only a few primatives are transparent then draw all the opaque objects, followed by the transparent primatives. Great for the sphere in the window.
- If many primatives are transparent then depth sort the primatives and draw them farthest to closest.
  - Basically, this is the painter's algorithm.
  - Painter's algorithm doesn't work easily for all possible geometric cases (intersecting triangles).
  - Depth sorting many primatives is slow.

Blending

This problem is also called *drawing order independent transparency*

- *Depth Peeling* is a technique for achieving accelerated (e.g. interactive) drawing order independant transparency.
- See the white paper published by nVidia in 2001 by Cass Everitt.

Blending – Example 1

```
glBlendFunc( GL_ONE, GL_ONE );
```

- The source pixel is multiplied by one.
- The destination pixel is multiplied by one.
- The pixels are added component-wise.
- The result is stored in the frame buffer.

Blending – Example 1

`glBlendFunc( GL_ONE, GL_ONE );`

Suppose the source pixel is `(0,255,0,128)`

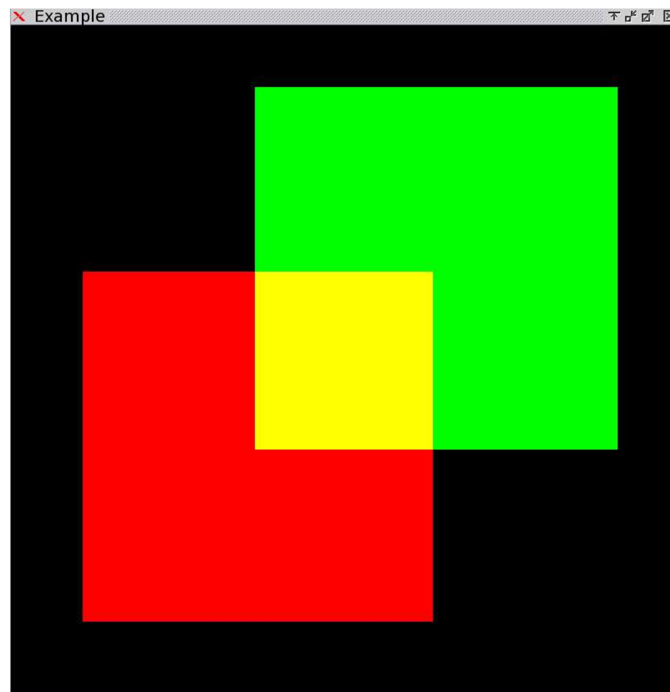
Suppose the destination pixel is `(255,0,0,128)`

- source pixel * source factor = `(0*1,255*1,0*1)` = `(0,255,0,128)`
- destination pixel * destination factor = `(255*1,0*1,0*1)` = `(255,0,0,128)`
- The sum is `(255,255,0)` which is stored in the frame buffer.
- Note that the alpha value isn't even used to combine the pixels here!

Blending – Example 1

```
glBlendFunc( GL_ONE, GL_ONE );
```

- Red is drawn 1st, far, with blending off.
- Green drawn second, near, with blending on.



Blending – Example 1

Literally, this and the following examples use variants of:

```
glColor4f(1,0,0,0.5) ;
draw_quad() ;

glEnable(GL_BLEND) ;
glBlendFunc(GL_ONE, GL_ONE) ;

glColor4f(0,1,0,0.5) ;
draw_quad() ;

glDisable(GL_BLEND);
```

Blending – Example 2

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
```

- The source pixel is multiplied by its alpha value.
- The destination pixel is multiplied by one.
- The pixels are added component-wise.
- The result is stored in the frame buffer.

Blending – Example 2

`glBlendFunc(GL_SRC_ALPHA, GL_ONE);`

Suppose the source pixel is `(0,255,0,128)`

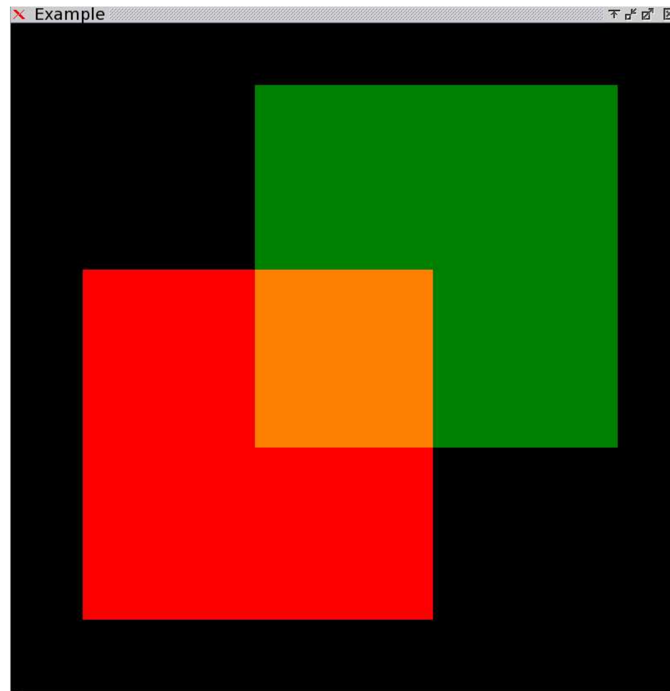
Suppose the destination pixel is `(255,0,0,128)`

- source pixel * source factor = `(0*0.5,255*0.5,0*0.5)` = `(0,128,0)`
- destination pixel * destination factor = `(255*1,0*1,0*1)` = `(255,0,0)`
- The sum is `(255,128,0)` which is stored in the frame buffer.
- Can think of the alpha value as normaized from 0 to 1 for multiplication. $128/255 = 0.5$

Blending – Example 2

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
```

- Red is drawn 1st, far, with blending off.
- Green drawn second, near, with blending on.



Blending – Example 3

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

- The source pixel is multiplied by its alpha value.
- The destination pixel is multiplied by one minus the alpha value of the source pixel.
- The pixels are added component-wise.
- The result is stored in the frame buffer.

Blending – Example 3

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Suppose the source pixel is `(0,255,0,128)`

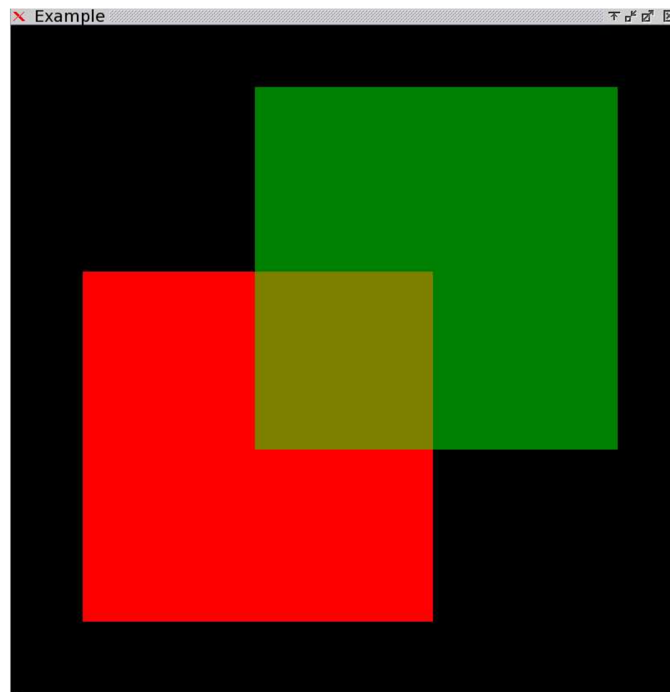
Suppose the destination pixel is `(255,0,0,128)`

- source pixel * source factor =
  `(0*0.5,255*0.5,0*0.5)` = `(0,128,0)`
- destination pixel * destination factor =
  `(255*0.5,0*0.5,0*0.5)` = `(128,0,0)`
- The sum is `(128,128,0)` which is stored in the frame buffer.

Blending – Example 3

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

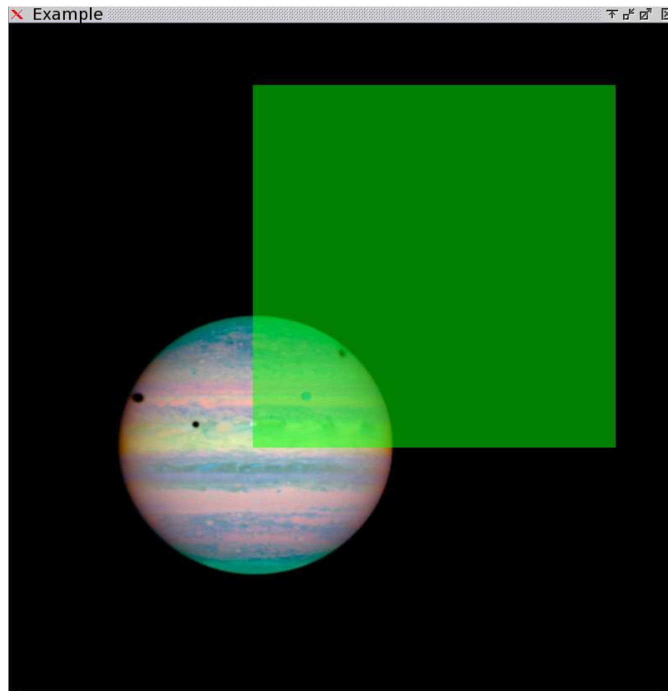
- Red is drawn 1st, far, with blending off.
- Green drawn second, near, with blending on.



Blending & Texturing

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

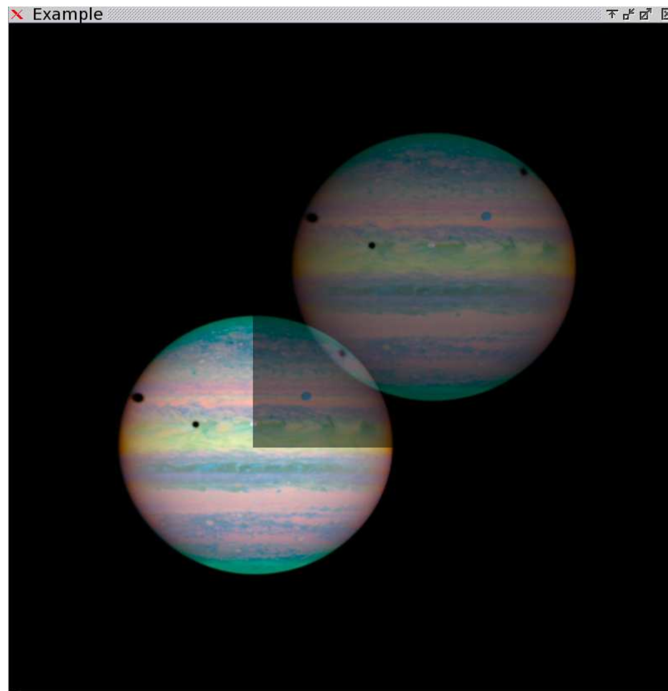
- Lower left picture is drawn 1st, far, with blending off.
- Green quad drawn second, near, with blending on.



Blending & Texturing

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

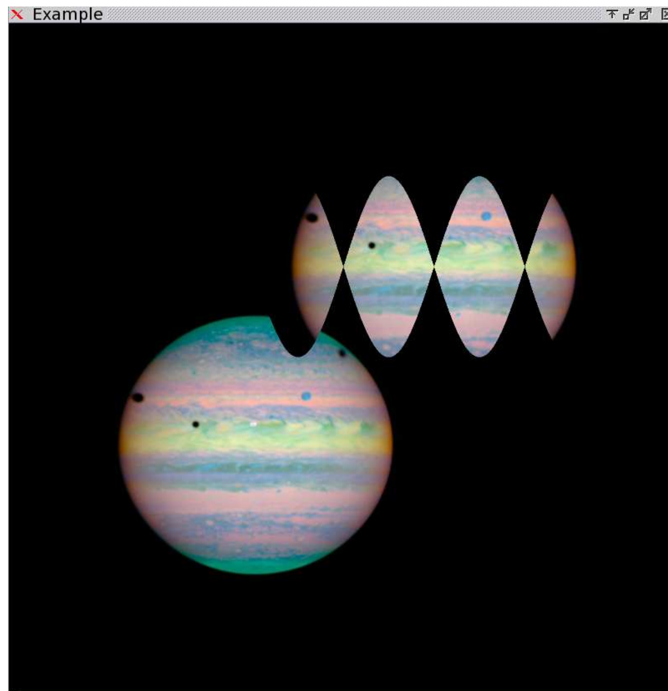
- Lower left picture is drawn 1st, far, with blending off.
- Upper right drawn second, near, with blending on. $\alpha = 0.5$;



Blending & Texturing

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

- Lower left picture is drawn 1st, far, with blending off.
- Upper right drawn second, near, with blending on. α varies.



Blending & Texturing

Billboarding is generally referred to as the process of:

- wrapping a texture around a quad
- using the alpha channel to make unwanted parts of a texture transparent
- the result is a non-rectangular picture.
- Billboarding is very fast & commonly used in video games. Its easy to spot a billboard because it transforms as a 2D plane rather than the 3D object its usually representing.

Blending & Texturing

Looking behind the billboard:

Bump Mapping

Bump mapping, also referred to as *per pixel lighting*, is a technique which allows the programmer to specify a surface normal **for each pixel** rather than just the geometric primitive (which will usually have many pixels associated with it).

Bump mapping provides us with a way creating the illusion of complex geometry without having to create and process complex geometry.

Look at the specs for GPUs and you will see that the number of *texels* (texture elements) per second is roughly an order of magnitude larger than the number of vertices per second.

nVidia GF6800 Ultra: 600 Million vertices – 6.4 Billion Texels per second.

Bump Mapping

- Per pixel normals are represented in *tangent space* using, for instance, a second RGB texture.
- So bump mapping is like multitexturing.
- It is different than multitexturing because we are combining textures (one representing normals, the other a conventional texture) in a more sophisticated way.
- *Tangent space* refers to a coordinant system that is relative to the current plane. That is, in tangent space, the normal vector (0,0,1) is coincident with the normal of the geometry on which we are pasting the the bump map.

Bump Mapping

- Can think of bump maps as perterbing the direction of the geometric normal at each pixel.
- The per pixel normals are encoded in an RGB texture map.

RGB Encoding	Tangent Space	Comment
127 127 127	0 0 0	The “origin”
127 127 255	0 0 1	Straight up in the direction of geometric normal
127 127 0	0 0 -1	Straight down, opposite of The geometric normal

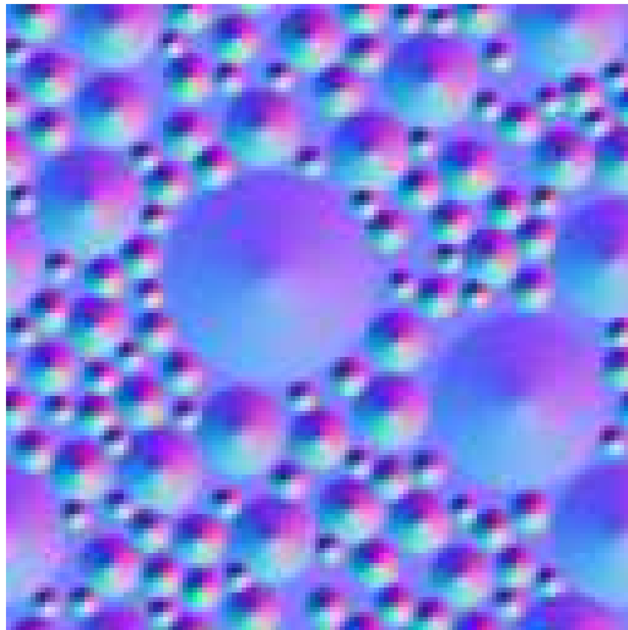
Bump Mapping

RGB Encoding	Tangent Space	Comment
0 127 127	-1 0 0	-X direction in the plane
255 127 127	1 0 0	+X direction in the plane
127 0 127	0 -1 0	-Y direction in the plane
127 255 127	0 1 0	+Y direction in the plane

Notice all the “blue” in bump maps.

Bump Mapping

RGB view of a bump map.



Bump Mapping

To do bump mapping use an OpenGL 1.4 feature called *texture combiners* which provide a texture environment for more sophisticated mixing of textures.

Steps to do bump mapping:

- Setup texture environment for bump map (texture combiners).
- Setup texture environment for a second texture (if desired).
- disable lighting!
- calculate the light direction at each vertex and transform into RGB tangent space at each vertex of the geometry.
- color the vertex with this color.

Bump Mapping – Setup

- Setup texture environment for bump map (texture combiners).

```
glActiveTextureARB(GL_TEXTURE0_ARB) ;
glBindTexture(GL_TEXTURE_2D,mybumptex) ;
glEnable(GL_TEXTURE_2D) ;

glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
          GL_COMBINE_EXT);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT,
          GL_DOT3_RGB_EXT);
```

Bump Mapping – Calculating the Light Direction

After the geometry to be bump mapped has been transformed into place do:

- `glGetFloatv(GL_MODELVIEW_MATRIX, mv_mat)`
- `glGetLightfv(GL_LIGHT0, GL_POSITION, lp_vec)` (gives us the position of the light in eye coordinates).

Recall that normal vectors are **not** transformed the way geometric vertices are transformed. Normal vectors are transformed by the **inverse** of the geometry transformation (the geometry transform is also known as the modelview matrix).

Recall that the inverse of an orthonormal matrix is its transpose.

Bump Mapping – Calculating the Light Direction

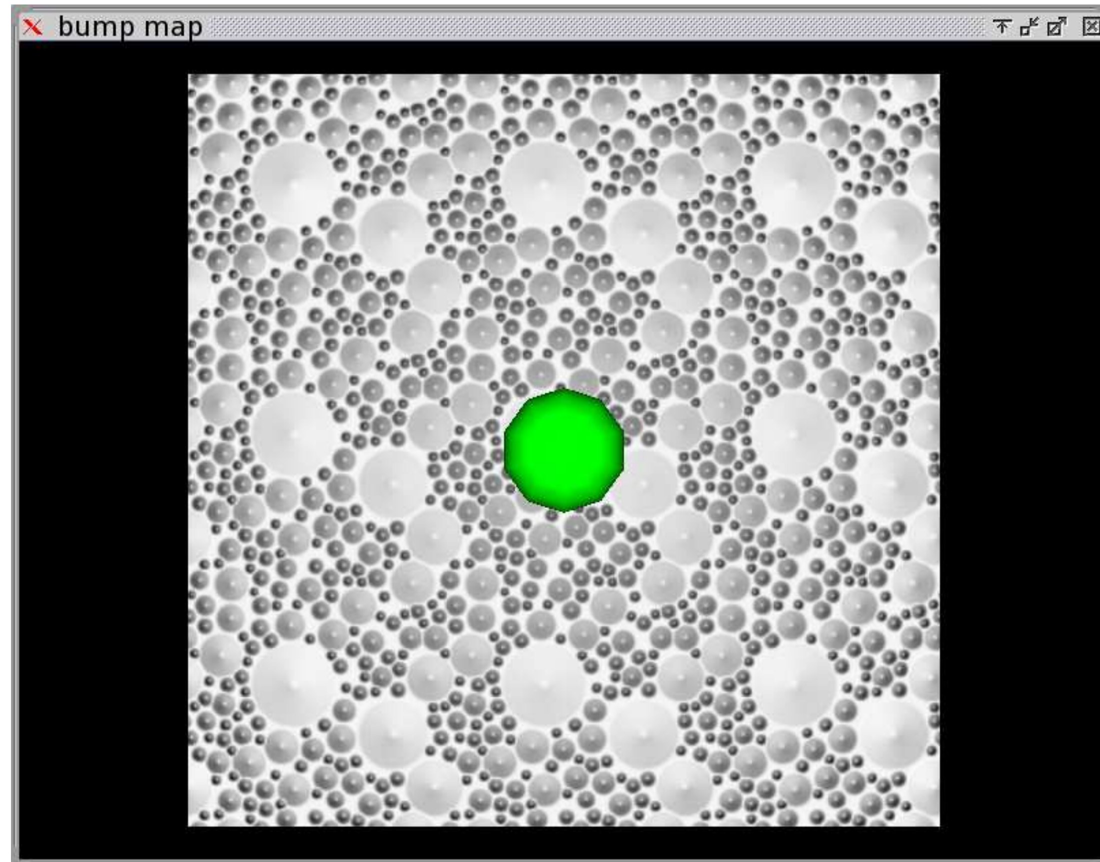
- The position of the light in eye coordinates (`lp_vec`) is transformed as a normal, so calculate the inverse of the modelview matrix (`mv_mat`), call it `mv_mat_inv`.
- Transform `lp_vec` by `mv_mat_inv` to get `lp_vec_obj`, the light vector in an object relative coordinate system.
- Rotate `lp_vec_obj` into tangent space by transforming it by M.

```
float M[16] = {1.0, 0.0, 0.0, 0.0,
               0.0,-1.0, 0.0, 0.0,
               0.0, 0.0, 1.0, 0.0,
               0.0, 0.0, 0.0, 1.0};
```

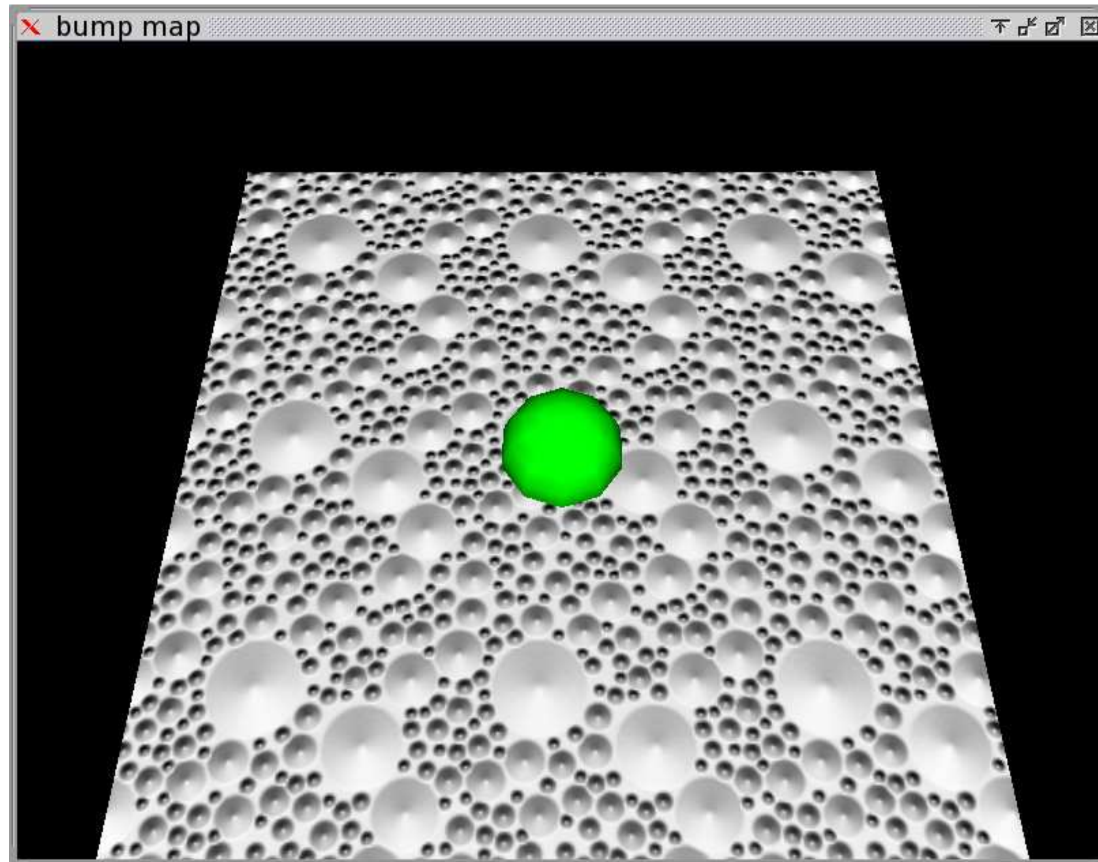
Bump Mapping – Calculating the Light Direction

- Call the result of transforming `lp_vec_obj` into tangent space `lp_tan`.
- Finally, rescale each component of `lp_tan` by multiplying it 0.5 and adding 0.5 ;
- Call the result is used in `glColor3f()` for a vertex.
- This color calculation is done separately for each vertex of the geometry to be bump mapped.

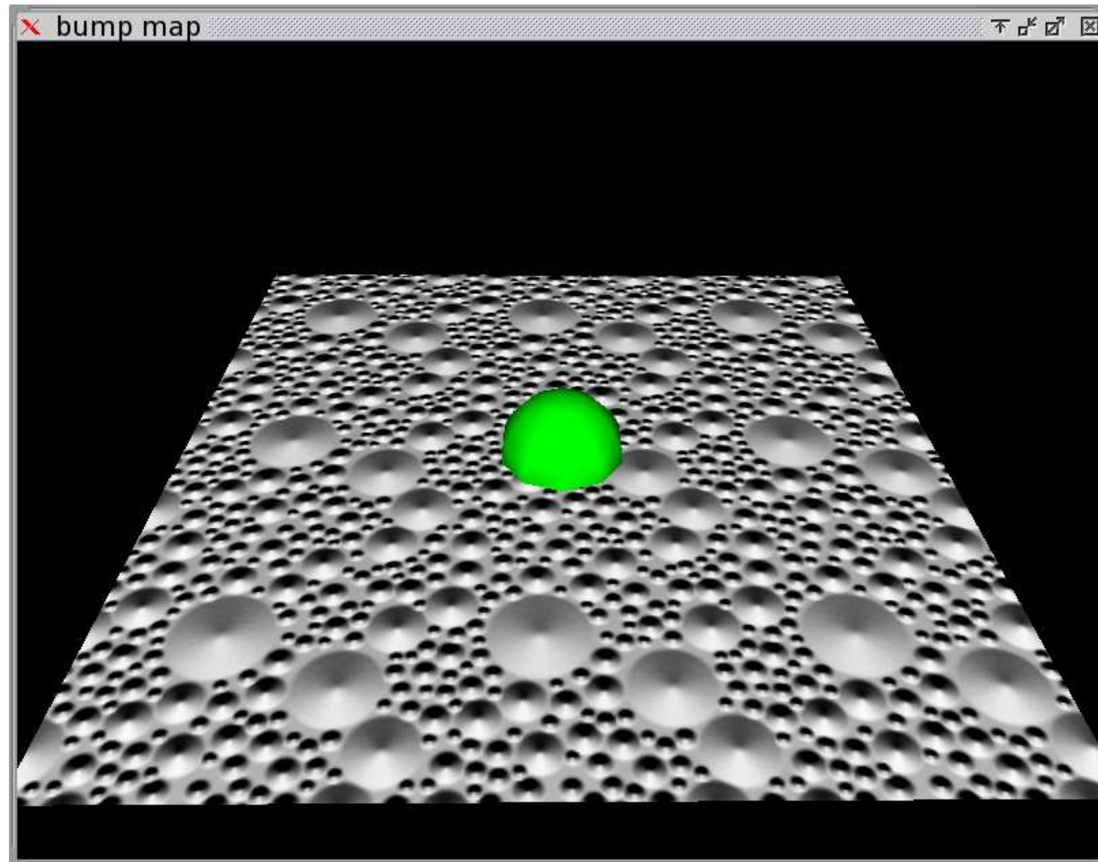
Bump Mapping – Results



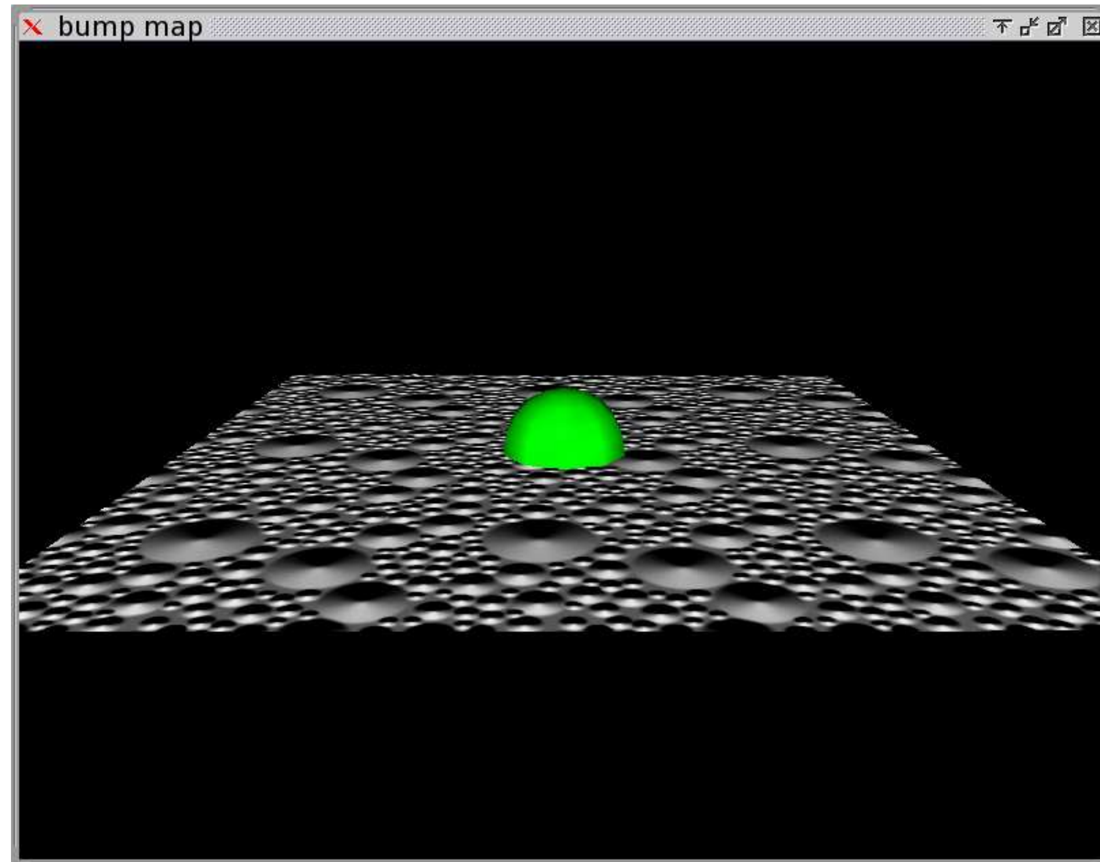
Bump Mapping – Results



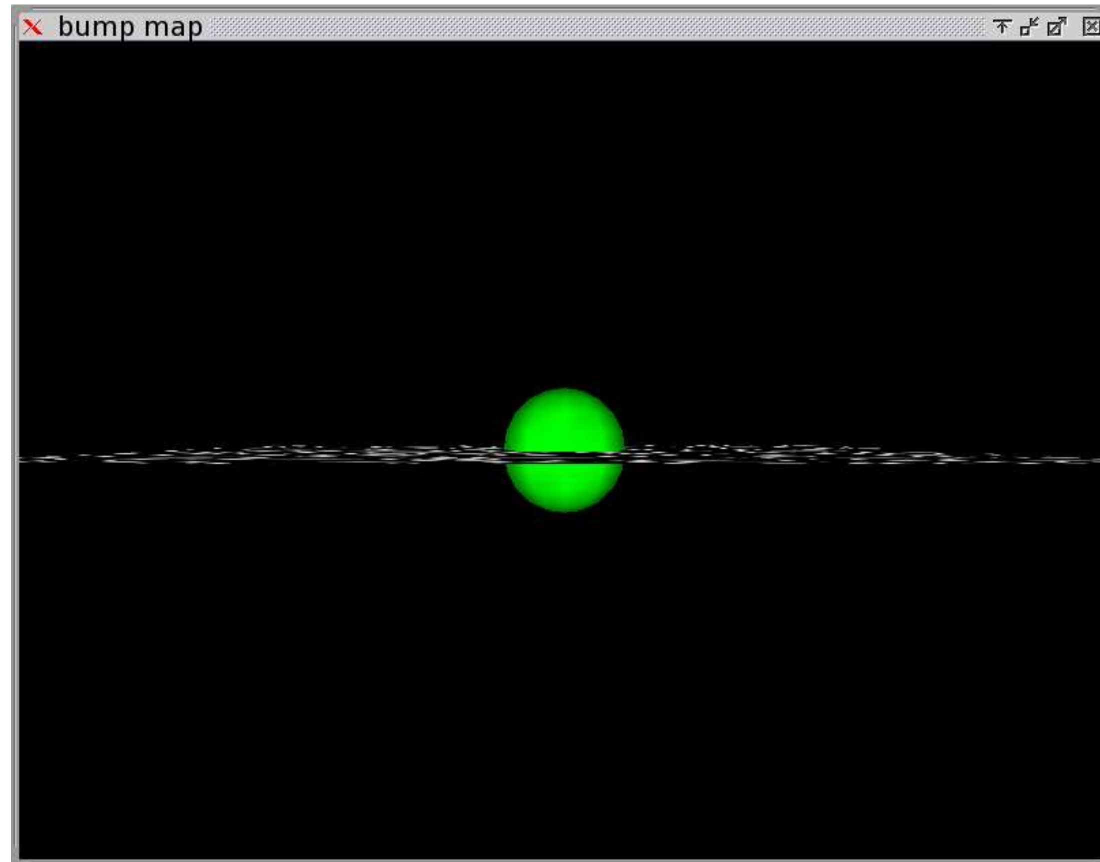
Bump Mapping – Results



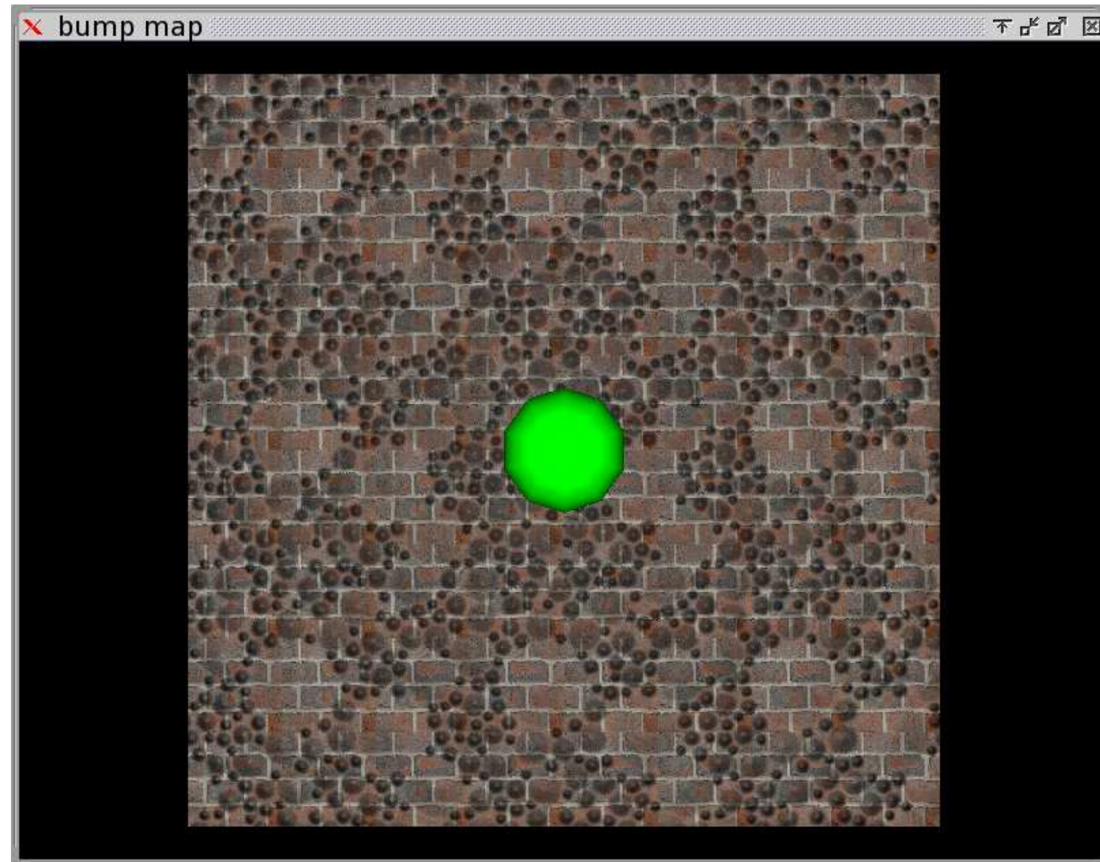
Bump Mapping – Results



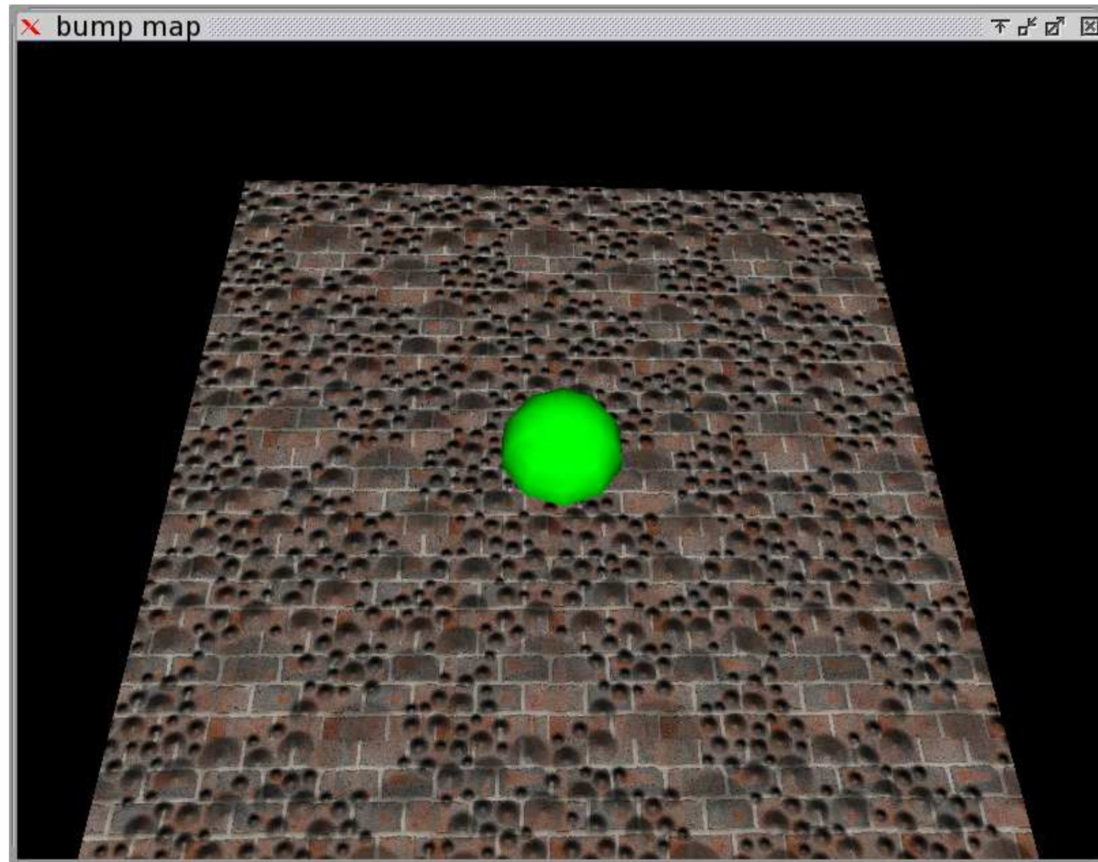
Bump Mapping – Results



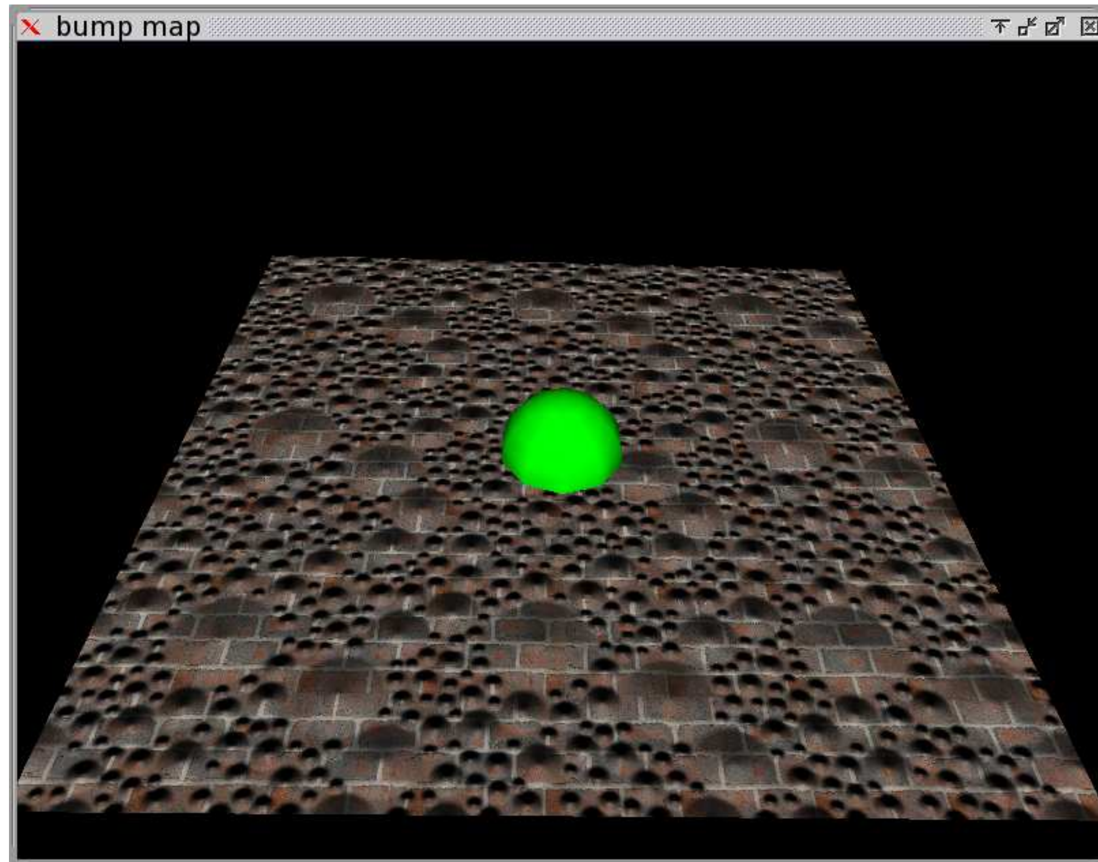
Bump Mapping – Results



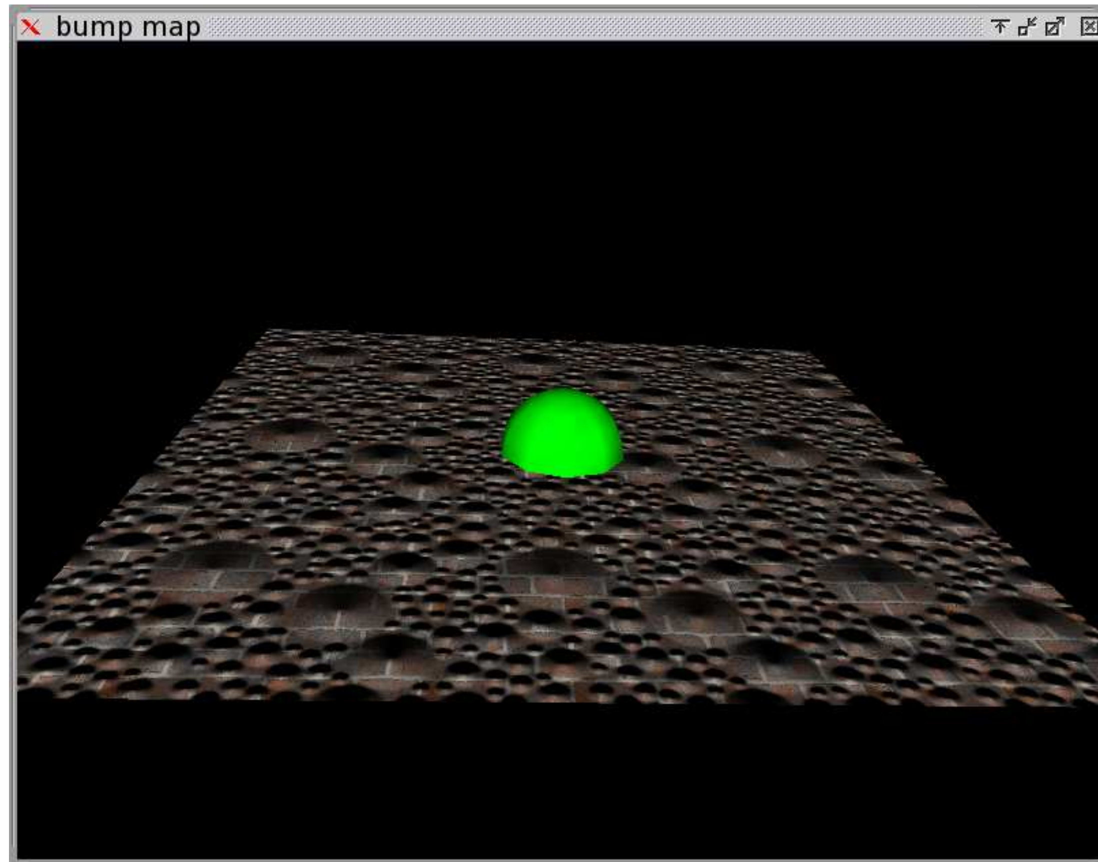
Bump Mapping – Results



Bump Mapping – Results



Bump Mapping – Results



Bump Mapping – Results