AwesomeBump v1.0

K. Kolasiński

January 18, 2015

Abstract

This paper explains most important methods and algorithms used in AwesomeBump (AB) tool in version v2.0. I assumed that reader has some mathematical background about derivatives, differential operators, numerical methods and iterative solvers. Note that in v2.0 some of algorithms was modified.

1 Height to normal conversion

This is the simplest problem which I will describe in this text. We can calculate normal vector of surface $z = f(\boldsymbol{r})$ at some point $\boldsymbol{r}' = (x', y')$ using the cross product of tangent vectors of this surface at point $\boldsymbol{r}'$.

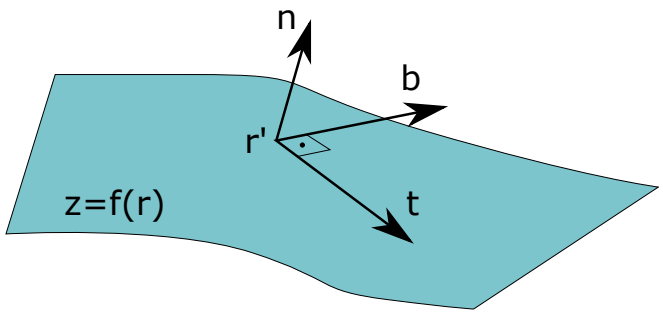

Figure 1: Schematic representation of the surface $z = f(\boldsymbol{r})$ with normal $\boldsymbol{n}$, tangent $\boldsymbol{t}$ and bitangent $\boldsymbol{b}$ vector at point $\boldsymbol{r}'$.

In order to calculate tangent vector $\boldsymbol{t}$ we use the following formula

$$\boldsymbol{t} = (1, 0, \frac{\partial f}{\partial x}\Big|_{r=r'}),$$

numerically (using the finite difference method wikipedia) this will have following form

$$\boldsymbol{t} = (1, 0, \frac{f(x + \Delta x, y) - f(x, y)}{\Delta x}).$$

In order to have control on the amplitude of the normal I introduced the depth parameter α (see Figure 2):

$$\boldsymbol{t} = (1, 0, \alpha\frac{f(x + \Delta x, y) - f(x, y)}{\Delta x}).$$

For $\alpha = 0$ the surface will be completely flat and for $\alpha \gg 1$ surface will be stepper.
After this step $\boldsymbol{t}$ vector is normalized

$$\boldsymbol{t} := \text{normalize}(\boldsymbol{t}).$$

The process of calculation of the $\boldsymbol{b}$ vector is analogical. Normal vector is calculated from the cross product of the tangent and bitangent vector

$$\boldsymbol{n} = \boldsymbol{t} \times \boldsymbol{b} = \text{cross}(\text{t}, \text{b})$$

Here is the full code of the GLSL shader which converts the height map to normal texture.

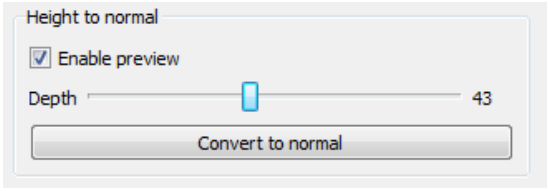

Figure 2: Height to normal depth parameter control slider.

```
subroutine(filterModeType) vec4 mode_height_to_normal(){
    const vec2 size = vec2(1.0,0.0);
    const ivec3 off = ivec3(-1,0,1);
    vec2 tex_coord  = v2QuadCoords.st;
    vec4   hc = texture(layerA, tex_coord);
    float s11 = hc.x;
    float s21 = textureOffset(layerA, tex_coord, off.zy).x;
    float s12 = textureOffset(layerA, tex_coord, off.yz).x;
    vec3 va   = normalize(vec3(size.xy,10*gui_hn_conversion_depth*(s21-s11)));
    vec3 vb   = normalize(vec3(size.yx,10*gui_hn_conversion_depth*(s12-s11)));
    vec3 bump = normalize( cross(va,vb) );
    return vec4(clamp(bump*0.5 + 0.5,vec3(0),vec3(1)),1);
}
```

Note that I used here different notation than before e.g. $\alpha = 10 \cdot \text{gui\_hn\_conversion\_depth}$, and here $\Delta x = 1$. Additionally at the end of the shader the normal map is converted to standard textures values i.e. components have values from 0 to 1. This approach of calculating the normal texture from height map can be found in many places in the internet.

2 Normal to height conversion

This problem is more complicated. As it was in previous section we define a surface function of form

$$z = h(x, y),$$

where h is the height field in our case described by the bump texture which we want to find, and (x, y) are UV coordinates. Mathematically, normal vector $\boldsymbol{n}$ can be calculated by acting with the gradient operator ∇ on a new function

$$g(x, y, z) = z - h(x, y),$$

thus we have

$$\boldsymbol{n}(x, y, z) = \nabla g = \nabla(z - h(x, y)) = (-\frac{\partial h}{\partial x}, -\frac{\partial h}{\partial y}, 1) \equiv -\nabla H, \tag{1}$$

where $\nabla H = (\frac{\partial h}{\partial x}, -\frac{\partial h}{\partial y}, 1)$. Now we act with the divergence operator on the Eq.(1), which gives following partial differential equation

$$\begin{aligned} \nabla \cdot \boldsymbol{n} \quad &= \quad -\nabla \cdot \nabla H \\ \frac{\partial n_x}{\partial x} + \frac{\partial n_y}{\partial y} \quad &+ \frac{\partial n_z}{\partial z} = \quad -\nabla^2 H = -(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} + 0). \end{aligned}$$

From Eq. (1) we have $\frac{\partial n_z}{\partial z} = 0$, which lead to final formula

$$\frac{\partial n_x}{\partial x} + \frac{\partial n_y}{\partial y} = -\left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2}\right). \tag{2}$$

In order to solve this equation numerically I used finite difference approximation, which give following equation (assuming that $\Delta x = \Delta y = 1$, where 1 is texture offset - nearest neighbors)

$$\begin{aligned} \frac{n_x(x + \Delta x, y) - n_x(x - \Delta x, y)}{2\Delta x} + \frac{n_y(x, y + \Delta x) - n_y(x, y - \Delta x)}{2\Delta x} \quad = \\ -\left[\frac{H(x + \Delta x, y) + H(x - \Delta x, y) - 2H(x, y)}{\Delta x^2} + \frac{H(x, y + \Delta x) + H(x, y - \Delta x) - 2H(x, y)}{\Delta x^2}\right] \quad . \end{aligned}$$

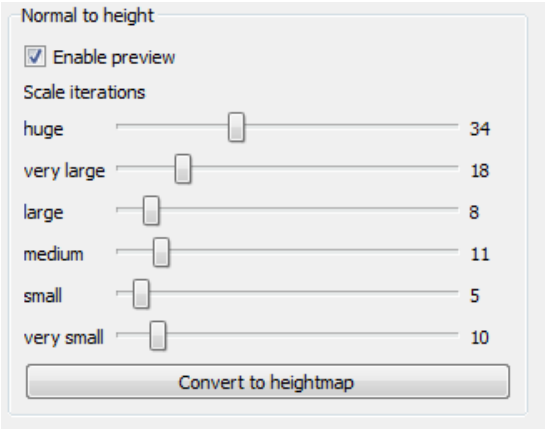

Figure 3: Normal to height control sliders. "Huge" slider corresponds to n_1 number of iterations, "very small" slider sets the n_6 number of iterations in MGI solver which corresponds to $k = 1$.

The equation above is reordered in such way that we can calculate $H(x, y)$ from it, thus we have

$$H(x,y) = \frac{H(x+1,y) + H(x-1,y) + H(x,y+1) + H(x,y-1)}{4}, \tag{3}$$

$$+ \frac{n_x(x+1,y) - n_x(x-1,y) + n_y(x,y+1) - n_y(x,y-1)}{8} \tag{4}$$

in which unknown is $H(x, y)$ function.

This equation can be solve iteratively by calculating $H(x, y)$ for each pixel, then calculating it again, again, and again until the solution for H stop to change. One thing we need to do is to assume proper boundary conditions, which I assumed for simplicity to be periodic boundary conditions. This is quite natural because usually we want to have seamless textures, and additionally periodicity is build-in feature of UV coordinates which simplify many things in GLSL. If you know a little bit of numerical methods you should notice that scheme (3) is the simplest solution in case of iterative solvers but the convergence of it is very poor. Strictly speaking you may need even thousand of iterations to reach the convergence and proper solution thus proper form of $H(x, y)$ texture. Fortunately, we can improve the algorithm above using a simple trick which is called multi-grid iteration (MGI) approach. The MGI method which I will describe in this text will be quite different than it is normally used in engineering problems. We introduce a scale parameter $k = \Delta x = \Delta y$, which means basically that we will calculate derivatives in (2) for further distances (k is usually an integer ≥ 1), which leads to new form of equation (3):

$$H(x,y) = \frac{H(x+k,y) + H(x-k,y) + H(x,y+k) + H(x,y-k)}{4}, \tag{5}$$

$$+ \frac{n_x(x+k,y) - n_x(x-k,y) + n_y(x,y+k) - n_y(x,y-k)}{8}.$$

I will not go into details of multi-grid medthods, but in AB the algorithm is following: $H(x, y)$ is obtained for after n_1 number of iterations for $k = 32$. This H is used as an initial image for $k = 16$ which again is iterated n_2 times, then $k = 8$, then $k = 4$, $k = 2$ and finally $k = 1$ with n_6 number of iterations. Note that the case for $k = 1$ in (5) restores the equation (3). One may check that this approach lead to convergence after around hundred of iterations. Of course this will depend on the size of the input image. The number of iterations $n_1, n_2, ..., n_6$ for each value of scale parameter can be changed in program with horizontal sliders in normal to height conversion tool (see Fig. 3).

The shader which solves Eq. (2) using MGI method is following:

```
subroutine(filterModeType) vec4 mode_normal_to_height(){
    float scale    = hn_min_max_scale.z; // scale parameter
    vec3 off       = scale*vec3(vec2(-1,1)*dxy,0);
    vec2 tex_coord = v2QuadCoords.st;

    float hxp      = texture(layerA,tex_coord + off.yz).x;
    float hxm      = texture(layerA,tex_coord + off.xz).x;
    float hyp      = texture(layerA,tex_coord + off.zy).x;
    float hym      = texture(layerA,tex_coord + off.zx).x;

    float nxp      = 2*(texture(layerB,tex_coord + off.yz ).x-0.5);
    float nxm      = 2*(texture(layerB,tex_coord + off.xz ).x-0.5);
    float nyp      = 2*(texture(layerB,tex_coord + off.zy ).y-0.5);
    float nym      = 2*(texture(layerB,tex_coord + off.zx ).y-0.5);

    float h = (nxp-nxm+nyp-nym)/8.0*scale + (hxp + hxm + hyp + hym)/4.0;     // Main equation
    return vec4(h);
}
```

In above, the hn_min_max_scale is an uniform variable which in z-th component contains the value of k. After the iterations for all values of k the heightmap is normalized to be in range from 0 to 1. In order to do the normalization the maximal and minimal value of the $H(x, y)$ is calculated. This part of calculation is done on CPU side:

```
...
GLint textureWidth, textureHeight;
glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_WIDTH, &textureWidth);
glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_HEIGHT, &textureHeight);
float* img = new float[textureWidth*textureHeight*4];
glGetTexImage(  GL_TEXTURE_2D,0,GL_RGBA,GL_FLOAT,img);
float min[3] = {img[0],img[1],img[2]};
float max[3] = {img[0],img[1],img[2]};
for(int i = 0 ; i < textureWidth*textureHeight ; i++){
   for(int c = 0 ; c < 3 ; c++){
        if( max[c] < img[4*i+c] ) max[c] = img[4*i+c];
        if( min[c] > img[4*i+c] ) min[c] = img[4*i+c];
    }
}
...
program->setUniformValue("min_color",QVector3D(min[0],min[1],min[2]));
program->setUniformValue("max_color",QVector3D(max[0],max[1],max[2]));
...
```

Then normalization shader is applied:

```
subroutine(filterModeType) vec4 mode_normalize_filter(){
    vec4 color = texture( layerA, v2QuadCoords.xy);
    color.rgb =  ( color.rgb - min_color )/(max_color-min_color) ;
    color.a = 1;
    return color;
}
```

3 Basemap to normal map

In order to calculate normal map from diffuse texture the sobel filter is applied on gray scaled diffuse image:

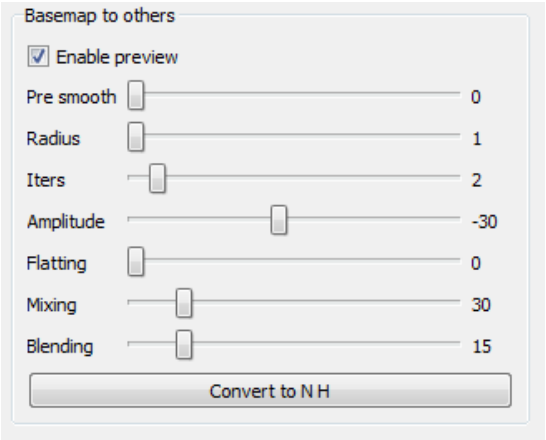

Figure 4: Base map conversion sliders.

```
const mat3 sobel_kernel =
        mat3(-1.0, 0.0, +1.0,
             -2.0, 0.0, +2.0,
             -1.0, 0.0, +1.0);


subroutine(filterModeType) vec4 mode_sobel_filter(){
    float sobel_x = 0;
    float sobel_y = 0;
    vec4 ocolor= vec4(0);
    for(int i = 0 ; i < 3 ; i++){
        for(int j = 0 ; j < 3 ; j++){
            sobel_x  += texture(layerA,v2QuadCoords.xy+vec2(i-1,j-1)*dxy).r*sobel_kernel[i][j];
            sobel_y  -= texture(layerA,v2QuadCoords.xy+vec2(i-1,j-1)*dxy).r*sobel_kernel[j][i];
    }}
    vec3 n = normalize(vec3(gui_basemap_amp*vec2(sobel_y,sobel_x),1));
    ocolor.rgb = clamp(n*0.5+0.5,vec3(0),vec3(1)); // convert to values from 0 to 1
    ocolor.rgb = vec3(1-ocolor.r,ocolor.gb);
    return ocolor;
}
```

The parameter gui_basemap_amp is responsible for the amplitude of the calculated normals (Amplitude slider in Fig. 4). Note that if gui_basemap_amp=0 then the image will be completely flat. After this step image can be presmoothed using the two-pass guassian blur filter (Presmooth slider, where the value of slider corresponds to the width of the Gaussian mask i.e. standard deviation of the gaussian distribution function).

After this step normals are manipulated by normal expansion filter, which is executed n-times, where n is controlled by the Iters slider:

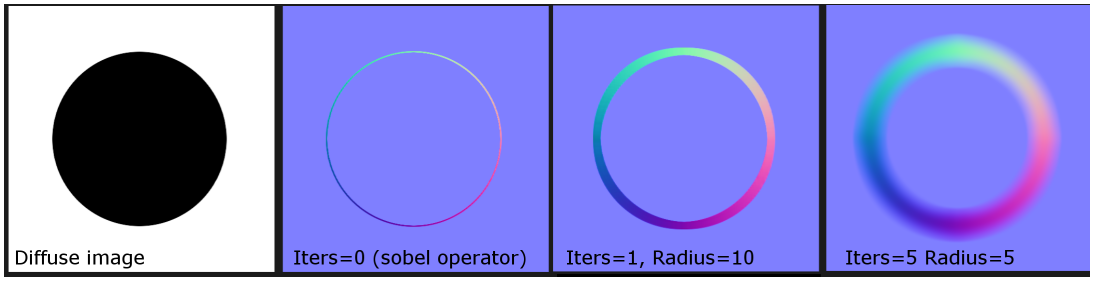

Figure 5: Normal extraction filter example outputs. From left: a) input image, b) image obtained from sobel filter (case when the parameter Iters is equal 0), c) the same but normal filter is applied (case when Iters$\geq$ 1). d) Large number of iterations leads to blurred image.

```
subroutine(filterModeType) vec4 mode_normal_expansion_filter(){
    vec3 filter  = vec3(0);
    float wtotal = 0.0;
    int radius   = gui_filter_radius;

    // normal expansion
    if(gui_combine_normals == 0){

    for(int i = -radius ; i <= radius ; i++){
    for(int j = -radius ; j <= radius ; j++){
        vec2 coords = vec2(v2QuadCoords.xy+vec2(i,j)*dxy);
        vec3 normal = normalize(2*texture(layerA,coords).xyz-1);

        float w = mix(length(normal.xy),
                      1/(20*gaussian(vec2(i,j),gui_filter_radius)*length(normal.xy)+1),
                      gui_normal_flatting+0.001);
        wtotal  += w;
        filter  += normal*w;
    }}
    filter /= (wtotal);//normalization

    return vec4(0.5*normalize(filter)+0.5,1);
    } else { ...
    ...
    }
}
```

Basically this code calculates the weighted arithmetic mean of normal vector where the weights are obtained from the slope of the normal at given point. For example when the value of expression length(normal.xy) is big then the slope is big. This filter makes that the edges become extruded along the edges. See example below:

The obtained image from the previous step is then mixed with the normal image obtained from sobel operator. There are two type of blending: a) slope-based (Mixing slider) and b) standard blending (Blending). See the code below:

```
subroutine(filterModeType) vec4 mode_normal_expansion_filter(){
    if(gui_combine_normals == 0){
        ....
    }else{ // blending and slope-based mixing
        vec3 n      = normalize(texture(layerA,v2QuadCoords.xy).xyz*2-1);//get normal
        float slope = (1.0/(exp(+10*gui_mix_normals* length(n.xy))+1)); // slope based blending param
        vec4 a = texture(layerA,v2QuadCoords.xy);//processed normal image
        vec4 b = texture(layerB,v2QuadCoords.xy);//normal image obtained from sobel filter
        return  mix(mix(a,b,slope),a,gui_blend_normals);
    }
}
```

Finally, when the "Convert to NH" button is pressed, the image visible in the right window is copied to the Normal texture tab, then Height texture is calculated using the algorithm presented in (3).

4 Specularity texture calculation

In order to calculate specularity from the diffuse texture the difference of gaussian (DOG) filters is applied. For more information see wikipedia and elsewhere in the internet.

5 Ambient occlusion texture

In AB ambient occlusion is calculated using well known Screen Space Ambient Occlusion approximation. For more information see wikipedia and elsewhere in the internet.