MTAT.03.015 Computer Graphics (Fall 2013) Exercise session VI: OpenGL. Blending. Auxiliary Buffers.

Konstantin Tretyakov, Ilya Kuzovkin

September 14, 2013

In this exercise session we shall learn about *blending* and practice the use of the *depth* and *stencil buffers*. As usual, the code base is provided in the zip archive on the course website as well as in Github. You will need to submit your solution as a zipped archive file.

1 Blending

So far we have been assuming that during rasterization color values are simply written to the pixels. E.g. when you ask OpenGL to rasterize a blue triangle at a particular position, the pixels belonging to that triangle will all be set to blue. Quite often, however, you might want to *blend* the new triangle with whatever is already existing in the frame.

To put it slightly more formally, suppose you are willing to draw with color $\mathbf{s} = (r_s, g_s, b_s)$ onto a pixel that currently has color value $\mathbf{d} = (r_d, g_d, b_d)$. *Blending* means that instead of completely replacing the pixel's current value:

$$\text{new\_pixel\_value} := \mathbf{s},$$

the resulting value of the pixel will be a linear combination of its old and new colors:

$$\text{new\_pixel\_value} := \lambda_1 \mathbf{s} + \lambda_2 \mathbf{d}\,.$$

The coefficients λ_1 and λ_2 determine how much of an effect the original and the new color have on the result. The most common use of blending is imitating transparency via *alpha-compositing*. For that, each color is augmented with the fourth component, typically referred to as α. An $\alpha = 1$ denotes full opacity, $\alpha = 0$ denotes full transparency, and values inbetween correspond to various levels of semitransparency. To implement alpha transparency using blending, we provide the value α_s along with the color $\mathbf{s}$ being drawn and compute the resulting pixel as follows:

$$\text{new\_pixel\_value} := \alpha_s \mathbf{s} + (1 - \alpha_s)\mathbf{d}\,.$$

Note that according to this rule, blending into a pixel a color **s** with $\alpha_s = 1$ will completely overwrite the old value with the new one. Setting a color with $\alpha_s = 0$ will leave the old value intact. Using $\alpha_s = 0.5$ will result in an equal mixture of the old and new colors.

To use blending in OpenGL you have to do three things:

1. Enable it using `glEnable(GL_BLEND)`.

2. Specify the λ_1 and λ_2 coefficients of the blending function using `glBlendFunc(..)`. For example, to implement alpha-transparency you have to do:

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

   Note that there is considerable freedom in configuring the blending coefficients (see the manual for `glBlendFunc`), which allows for some interesting uses. You can also change the addition to subtraction (see `glBlendEquation`).

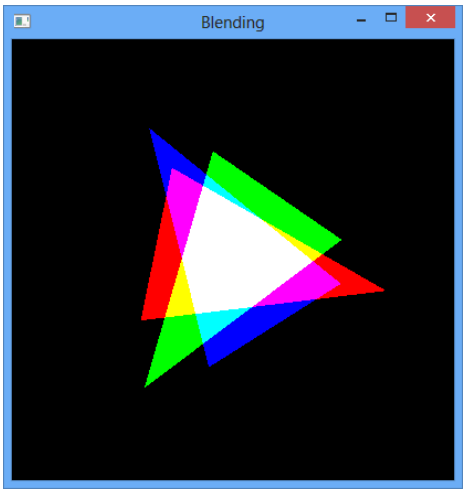
3. Finally, you need to specify the alpha component for all your colors using a four-component color function, e.g.:

   ```
   glColor4f(1.0, 0.0, 0.0, 0.5) // Semi-transparent red
   ```

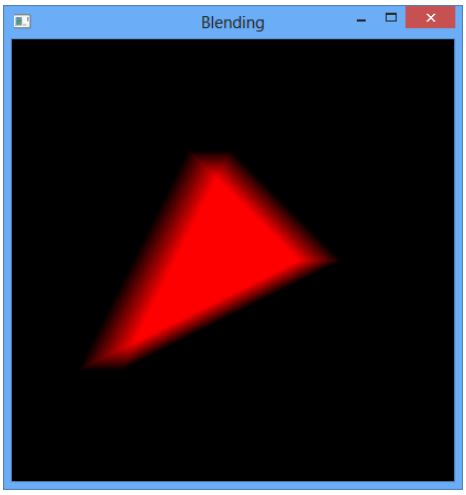
In general, blending is slightly less efficient than writing pixels directly (much less so in older graphics cards), so it is considered good practice to only enable blending for those primitives where it is required. In particular, though, rendering of anti-aliased ("smooth") lines and polygons[1] is only possible with alpha-transparency blending switched on.

[1] The technical details of anti-aliasing is a topic for one of the further lectures, but you must have noted the calls to `glEnable(GL_LINE_SMOOTH)` and `glEnable(GL_POLYGON_SMOOTH)` in the code of some previous practice sessions already. Those affect the rasterizer so that it will produce some pixels with semi-transparent alpha values to make the lines and edges look smoother on the screen. Obviously, for this to have any visual effect, alpha values must be taken into account, i.e. alpha blending must be enabled.

Exercise 1 (0.5pt). Open the project `1_Blending`. The first scene, described in the `draw_triangles` function, renders three rotating triangles of different colors. First try making the triangles semi-transparent using alpha-compositing. See what happens if you set different alpha-values for different vertices of the blue triangle. Finally, to get points for the exercise configure blending to produce the picture below:



Exercise 2 (0.5pt). Besides alpha-transparency, blending can be used to achieve effects such as *motion blur*, *soft shadows* and *depth of field*. In the second scene of the project (function `draw_motionblur`), implement the motion blur effect for a horizontally moving triangle. To do that you essentially need to average several consecutive frames of triangle movement. Currently, the function renders 20 copies of the triangle, each shifted a bit to the right, one on top of the other. For the motion blur effect you have to configure blending so that each copy is multiplied by 1/20 and added to whatever is currently on the screen. The result should look like the picture below.

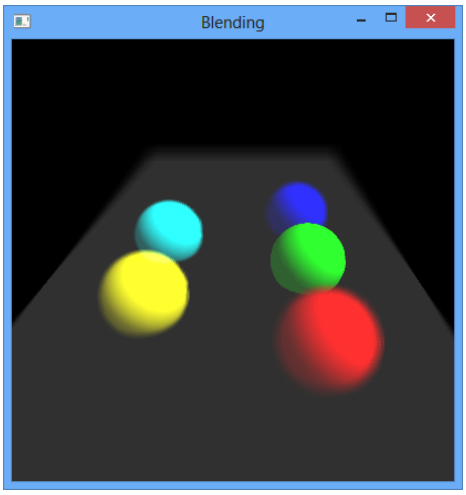


Exercise 3* (1pt). The *depth-of-field* effect aims to imitate the way real camera lenses work: objects located at around the *focus distance* are sharp, and objects further away or closer than the focus distance are blurred. The effect can be implemented in a manner similar to motion blur. The only difference is that the average is not taken over moving frames, but over slightly jittered camera projections. Read about the approach in detail in Chapter 10 of the "OpenGL Red Book"[2] (or find other descriptions in the internet). Implement this effect on a simple scene with several objects at different distances from the camera. Use blending to implement it[3]. An example image you may strive for is shown below. Notice how the green sphere is exactly at the focus distance. Also observe that the DOF is very small, so spheres even slightly out of focus look considerably blurred[4].

[2] `http://www.glprogramming.com/red/chapter10.html`

[3] Most (if not all) descriptions you will find in the internet will refer to the use of *accumulation buffer* for averaging multiple frames. Accumulation buffer is, however, removed from the latest versions of OpenGL specification in favor of blending and pixel shaders. Hence, to keep up with the times, I require you to use blending in this exercise as well.

[4] This situation is actually typical for close-up photography and some people deliberately imitate small DOF in photos by blurring foreground and background to imitate miniature scenes. See `http://en.wikipedia.org/wiki/Tilt%E2%80%93shift_photography`



Finally, note that blending is a general concept, not specific only to OpenGL. For example, the same notions apply to Allegro (see `al_set_blender`). HTML5 provides limited blending support via `globalCompositeOperation`.

2 Depth Buffer

In the lecture we learned about the *Z-buffer* (a.k.a *depth buffer*) algorithm. The simple take-away message was that you should always have it enabled when working with 3D graphics. Let us now practice the more explicit use of the depth buffer on an (admittedly somewhat artificial) example[5].

Exercise 4 (0.5pt). Open the project `2_DepthBuffer`. It contains a rendering of a model, that should be familiar to you from exercise session 4. Suppose we would like to render only the part of the model, that lies closer to the camera than distance 6 (which is the distance to the center of the model). Image below illustrates the desired effect.

[5] Think how you could achieve the same effect without resorting to depth buffer manipulations

To achieve this effect using the depth buffer do the following:

1. Draw the floor rectangle (`draw_floor`) as usual.
2. Switch off drawing to the screen using `glColorMask`. Now you will only render to the depth buffer.
3. Draw a large rectangle (e.g. 100×100) perpendicular to the viewer and at distance 6 from it. Hint: recall that in the camera frame the z axis points straight out of the screen.
4. Switch drawing to the screen back on and render the model as usual.

Implement this in `2_DepthBuffer`.

3 Stencil Buffer

Another useful auxiliary buffer is the *stencil buffer*. Like the depth buffer, it keeps a separate value for each pixel, however the values are integers. You can specify an operation used to update the stencil buffer using `glStencilOp`.

For example, `glStencilOp(GL_INCR, GL_INCR, GL_INCR)` specifies that the value in the stencil buffer has to be increased each time a pixel could be drawn to it. Hence, if you start with a zeroed stencil buffer and render the whole scene, each value in the stencil buffer will count how many triangles were rendered at that pixel.

In addition to the buffer update rule, you may specify a *stencil test*, which lets you decide on a pixel-per-pixel basis, which pixels should be drawn, depending on their stencil values. You enable the stencil test using `glEnable(GL_STENCIL_TEST)` and specify the type of test using `glStencilFunc`.

Stencil buffer allows to implement numerous interesting effects, such as shadows, reflections, contours or object intersections. Here we shall use it to implement a simple planar reflection.

Exercise 5 (1pt). Open project `3_StencilBuffer` and modify it as follows:

1. Modify `glutInitDisplayMode` to allocate memory for the stencil buffer. Add a line to enable the stencil test.
2. Modify `display` function to implement the following algorithm:
   (a) Clear the color, depth and stencil buffers
   (b) Render the floor *only* to the stencil buffer (disable writing to depth or color buffers temporarily), setting stencil buffer values to 1 for all pixels of the floor.
   (c) Set the stencil test to pass only for pixels that have stencil buffer value of 1. Render the figure of a man, mirrored about the $y = 0$ plane. Due to stenciling, only "floor" pixels will be affected.
   (d) Set the stencil test to always pass. Render the floor, using alpha-transparency with a coefficient of 0.8 for the floor.
   (e) Finally, render the man standing on the floor as normal.

The result should look as follows.