MovieFX: Blending based bokeh

Blending based effect for any hardware

Petr Schreiber

TBGL programming

for advanced

23rd August 2010

Introduction

There is an interesting trend which can be observed in the cut scenes present in many videogames released in the last years. Almost every 3D game now shows increased focus on cinematic feeling, which is achieved by extensive use of motion capture and specific work with camera, which includes not only its movement in the scene, but also simulating some of the optical properties as well.

This article provides one of the possible implementations of scene background blur, which is often referenced as *bokeh* in photography. You can see basic example on Pic. 1, where the bird is clearly separated from the out of focus background.

Pic. 1: Sharp bird and blurred background

Discussed technique also helps to stress more attention on the objects in the foreground, while still providing visually very compelling render of the scene behind with acceptable hardware requirements.

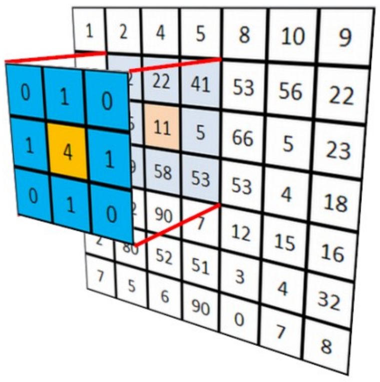
Please note that what follows is not fully featured depth of field simulation, but more an approximation usable for example in the mentioned cut scenes.

Possible approaches

There are many ways to achieve the blurred background, the most straightforward one would be to use pre-rendered image which was blurred in image editor. While this will indeed result in the fastest rendering, it is not well suited for cases when we need the background dynamic. That means, cases where camera flies around some object or where there are some moving objects in the background would mean using huge number of pre-rendered frames, which is both resource intensive and not very flexible when we need to do many changes to the background.

For linearly blurred background, it would be possible to render the scene to viewport smaller than the target render surface, capture the image to texture and then stretch the texture fullscreen. While this is probably the fastest approach, the result doesn't look very natural.

Another option would be to use OpenGL imaging extensions [1], which allow casting some convolution filters on image. The problem is that most implementations have this feature realised in software, so it can become very slow for higher resolutions. It is also not standard feature, but an extension programmer needs to check for. In the end, convolution is generally quite intensive operation, highly depending on size of the convolution kernel, which affects not just one pixel, but also its nearest surroundings, as shows Pic. 2.



Pic. 2 Convolution filter floating over 2D data[2]

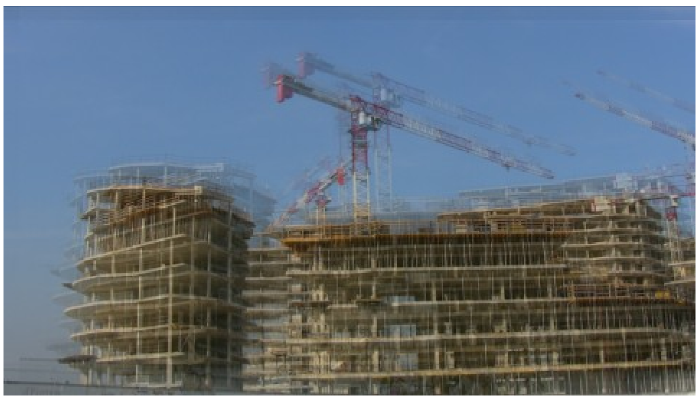
Next possibility is to take advantage of shader programming, which is common practice on modern hardware. We can render scene to some kind of buffer, and then perform Gaussian blur on each of the pixels [3]. This approach is used most widely, yet it gets slow on lower end hardware and it is unusable on graphic card without recent shader model.

Similar approach would be to use OpenCL to perform the image convolution. This is very common use of the GPU, which provides significant boost over CPU based solution, yet it requires decent new hardware as well.

Blending based approach

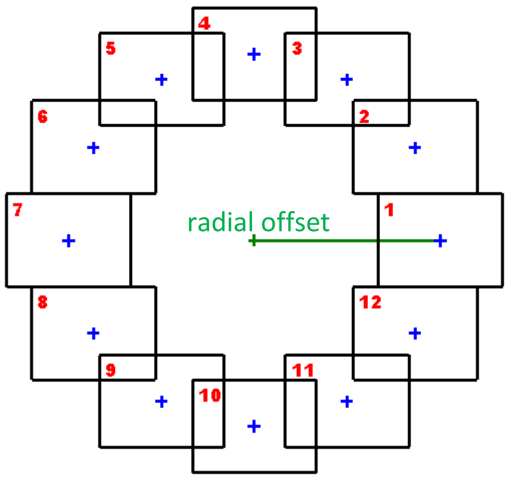
This article presents slightly different approach comparing to those mentioned in previous chapter. It uses just the resources present on all GPU hardware, which are textures and blending. It can be achieved using TBGL without any need for auxiliary OpenGL code.

The idea behind the technique is somehow simple. I am sure the reader can remember at least one movie where some person was hit to head, drunk or present in other similar state, affecting the way he sees the world. In such a case, we can observe the image is repeated to make impression of damaged vision, as seen on Pic. 3.



Pic. 3 "Damaged vision"

Very similar approach can be used for the purpose of blurred image. It is only necessary to make sure the image shift is more regular, in fact circular. This is shows Pic. 4 in exaggerated manner to highlight two interesting characteristics.



Pic. 4 Layout of image layers for blur

The radial offset tells us how much the image will be blurred, how far it will be "exploded" beyond the normal. Image with no blur would have radial offset equal to zero.

The second characteristic is the number of images composing the circle of blur. On the image we can see 12 of them, but it can be any number. Higher the count, better the blur impression is. Generally, it is safe to say that you need more images with bigger radial offset and vice versa. For typical applications, based on the tests, the ideal compromise is having about 15 layers of images.

Where to actually take the image for blurring? It is as simple as rendering the part of scene we want to blur directly to texture instead to screen.

It is unlucky that there is still quite a big number of graphic cards, which have problems with textures which are not sized to power of two dimensions. For this reason the program should first check for the NPOT support, create target texture accordingly and if necessary, adjust the viewport size for background rendering.

```
' You need to create variable to which we will pass data later
Dim texInfo As TBGL_tTexturingInfo

' To fill variable fields, just use following
TBGL_TexturingQuery( texInfo )

' Use the passed variable for tests in your code
If texInfo.NPOTSupport = %TRUE Then

  ' -- Grab screen at full detail
  bokehInfo.Width = width
  bokehInfo.Height= height

Else
  ' – Find closest match
  TBGL_EvaluatePOTMatch(width, height, bokehInfo.Width, bokehInfo.Height)
End If
```

```
 ' -- Make the target texture safely at checked size
  TBGL_MakeTexture( String$ ( bokehInfo.Width*bokehInfo.Height*3, 255 ),
                    %TBGL_DATA_RGB,
                    bokehInfo.Width, bokehInfo.Height, bokehInfo.textureID,
                    %TBGL_TEX_LINEAR )
```

Code 1: Creating texture to render to in safe way

Before you actually render the scene to texture, it is necessary to setup viewport to size of the texture. This might differ from windows client dimensions in case the NPOT textures are not supported and the code above picked nearest closest power of two match.

To make sure this will pose no problem, we need to use combination of two commands – *tbgl_Viewport* and *tbgl_RenderMatrix3D*.

```
Sub BokehRender_BeginCapture(bokehInfo As tBokehInfo)
  ' -- Set viewport to texture size
  TBGL_Viewport(0, 0, bokehInfo.Width, bokehInfo.Height, %TBGL_PARAM_PIXELS)

  ' -- But maintain original window ratio for perspective
  TBGL_RenderMatrix3D(%TBGL_CLIENTAREA)
End Sub

Sub BokehRender_EndCapture(bokehInfo As tBokehInfo)
  ' -- Capture the image
  TBGL_RenderToTexture( bokehInfo.textureID, 0, 0, bokehInfo.Width, bokehInfo.Height )

  ' -- Set viewport to original size
  TBGL_Viewport(0, 0, 1, 1, %TBGL_PARAM_RELATIVE)
  TBGL_RenderMatrix3D
End Sub
```

Code 2: Auxiliary procedures for maintaining correct aspect ratio for the rendering

Now just the final two steps remain – actual rendering of the background *bokeh* and the focused foreground.

Rendering the captured image to create illusion of *bokeh* presents few challenges. The first one is how to compose the images together. Here comes in the blending, which is designed for such a cases. But it is not that simple.

It is important to note that when blending the image, to maintain its brightness, the multiple copies must be rendered at fraction of the brightness. So for 12 layers, you should render each image at 1/12 of intensity of original picture. This is easily achieved by approach shown in Code 3.

```
  brightness = 255/ bokehInfo.numImages
  TBGL_Color brightness, brightness, brightness
```

Code 3: Using TBGL_Color function to manipulate brightness

Second problem is depth information. As the scene we rendered to texture will serve as background and everything else should go in front of it, it would be ideal to make it depth neutral. This is made possible by disabling the depth mask, which stops writing to depth buffer for the time of rendering our layers.

```
' -- Set proper blending mode
TBGL_BlendFunc(%GL_ONE, %GL_ONE)

' -- Set 2D rendering mode for the layers
TBGL_RenderMatrix2D(-1,-1,1,1)
TBGL_ClearFrame

' -- Disable lighting for natural color and
' -- depth mask to make possible rendering forgeground later
TBGL_PushStateProtect(%TBGL_LIGHTING Or %TBGL_DEPTHMASK)
  ' -- Texturing and blending is required to properly draw the image
  TBGL_PushState(%TBGL_TEXTURING Or %TBGL_BLEND)

  ' – Bind texture we rendered to
  TBGL_BindTexture(bokehInfo.textureID)

  ' – Fix brightness for the image layers
  brightness = 255/bokehInfo.numImages
  TBGL_Color (brightness, brightness, brightness)

   ' -- Render the images radially distributed around the center of screen
  aStep = 360/bokehInfo.numImages
  For f = 1 To 360 Step aStep
    TBGL_PushMatrix

      TBGL_Rotate(angle)
      TBGL_Translate(0.001*bokehInfo.radialOffset, 0)
      TBGL_Rotate(-angle)

      ' -- Cached polygon to hold image
      TBGL_CallList(bokehInfo.displayListID)

    TBGL_PopMatrix
  Next

  TBGL_PopState
TBGL_PopStateProtect

'-- Return rendering back to 3D
TBGL_RenderMatrix3D
```

Code 4: Rendering the background bokeh

Code 4 shows the complete code to draw the *bokeh*. You can see we used 3 transformation commands for positioning the frame. We could simply use *Sin()* and *Cos()* with just single *tbgl_Translate*, but the way shown in code makes sure the calculations are performed on GPU, while the mentioned trigonometric functions would have to be run on CPU, not speaking of necessity of bothering the interpreter with extra *DegToRad()* call for conversion of degrees to radians.

As the code restores the original render matrix, you can render the scene in foreground right after it without any further steps necessary.

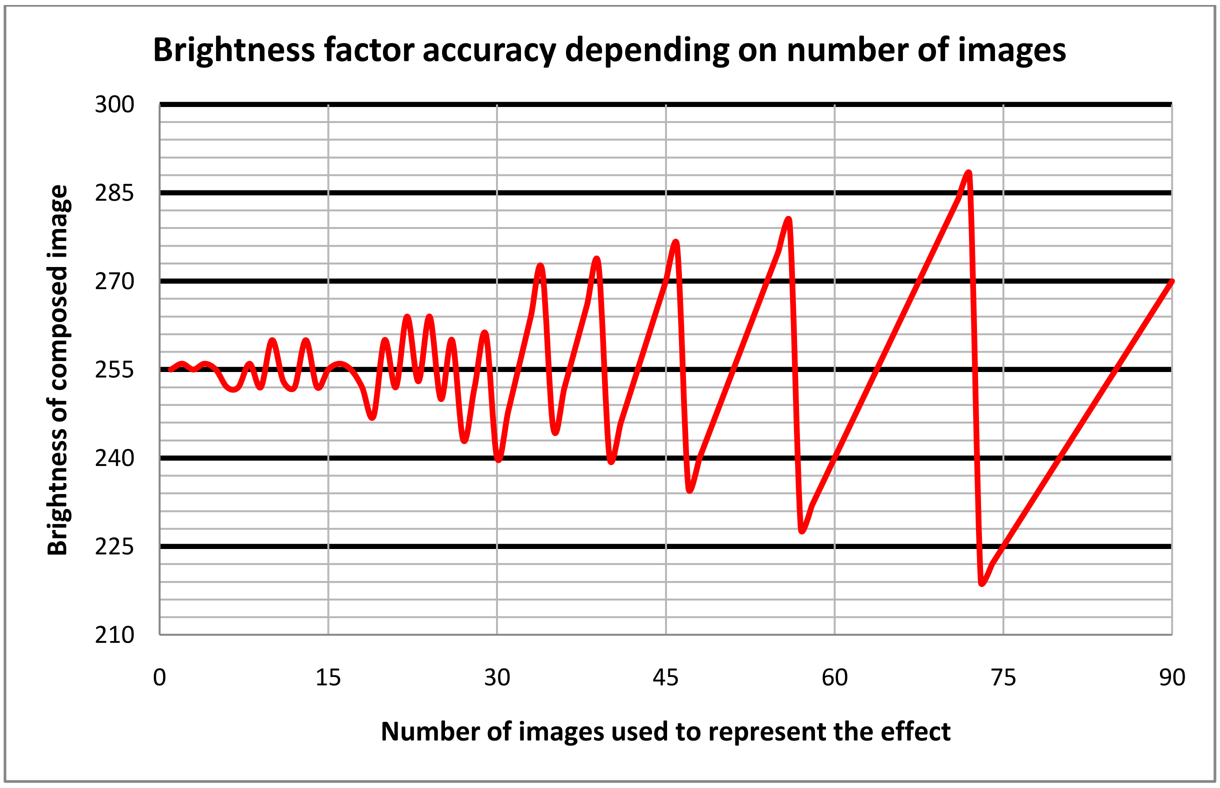
Problems of the approach

The demonstrated approach has three main issues to be aware of.

The first two are related to the colour precision. As noted in previous chapter, each image composing the final effect is rendered at fraction of the brightness. The problem is that currently standard image representation uses 8-bit numbers to describe each colour component. They are integer which means that not every number of images we choose can be used to reconstruct the original image brightness accurately.

For example, if we pick 15 images to represent the effect, we get the brightness calculated as 255/15, which equals to 17. If we multiply this number by number of images, we get back the original 255 brightness.

The problem is when the fraction evaluates as floating point number. This gets rounded to integer, so it results in same brightness factor for different number of images. As the result, the final composed image is not as precisely bright as the original image, and when dynamically changing the number of images, there can be observed nonlinear brightness level of the background.

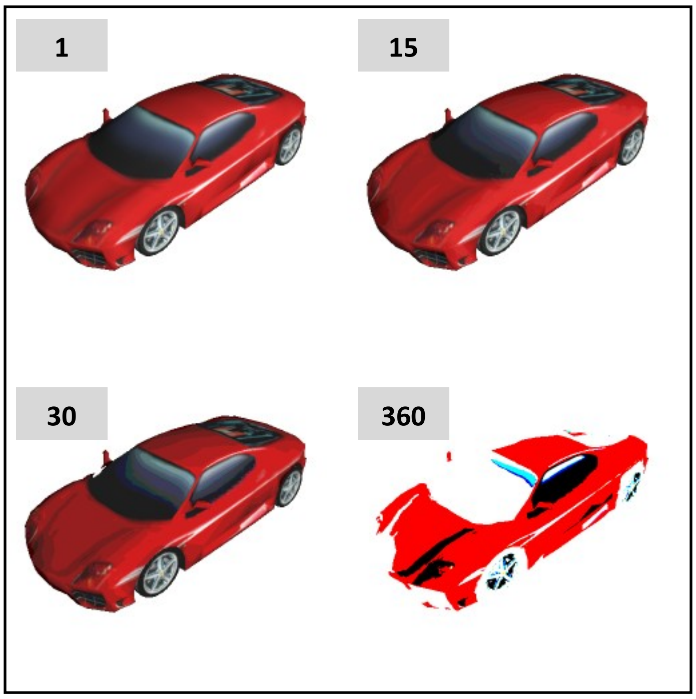


Pic. 5

To get the original brightness with blur, we would have to use 3, 5, 15, 17, 51 or for example 85 images. While the Pic. 5 looks dramatic, even other values are quite suitable for use until 25, after which the jumps get bigger.

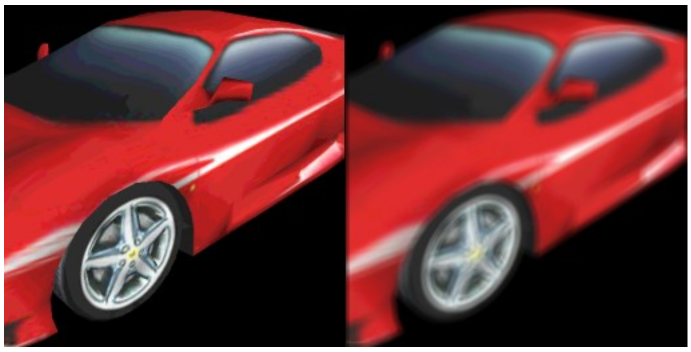
The second problem is correct colour representation of *bokeh*. Thanks to number rounding, the image can lose its visual fidelity with growing number of individual low brightness image layers.

This can be most effectively visualized with zero *radial offset*, which means no blur. With this setup the layers are just drawn one over each other, as seen on Pic. 6.



Pic. 6: Colour degradation with growing number of layers

This again gets much less dramatic once nonzero *radial offset* is involved. In such a case further interpolation takes place, and the single colour regions are softly washed out.



Pic. 7: Image with 15 image layers without (left) and with blur (right)

The last problem with *blending based bokeh* approach might be its full screen per-pixel blending itself. With growing number of the layers, the number of operations increases. According to performed tests, this factor is less and less limiting with the currently available hardware. Using low end *NVIDIA GeForce 9500GT*, it is still possible to get smooth frame rate at 1080p display resolution.

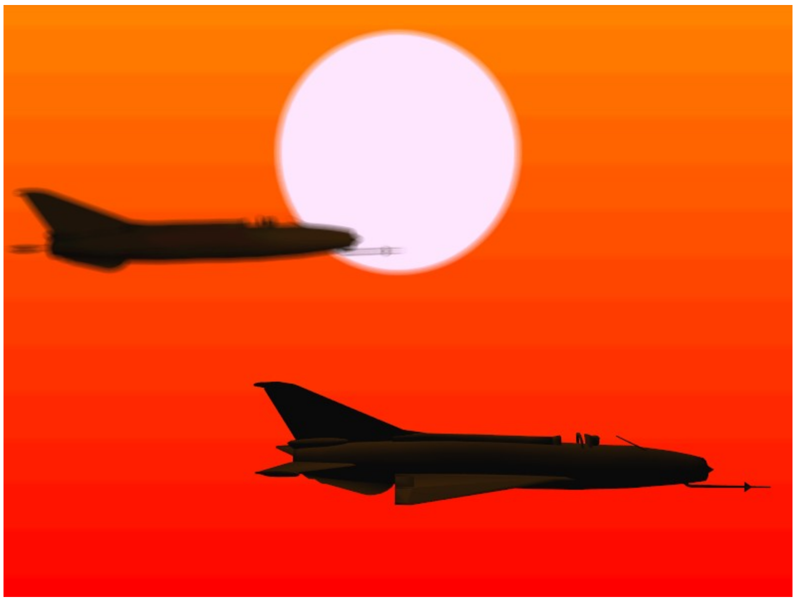
Conclusion

The approach discussed in this article provides the reader with complete implementation of technique approximating *bokeh* rendering for game cut scenes.

Its advantage is wide hardware compatibility, easy implementation and possible parameterization affecting both quality and final look of the effect.

The disadvantages of the solution lie mainly in the area of colour precision.

Part of the article are two example scripts, which demonstrate the technique. The scripts use dedicated include file for *bokeh* rendering, which the readers are welcomed to use in their ThinBASIC applications.



Pic. 8: Example application using the described technique

The first script shows scene displayed on Pic. 8, which shows focused object in front of completely dynamic blurred background.

The second script renders just the effect itself, which can be parameterized using trackbar manipulation to observe the *bokeh* closely. The script intentionally allows setting extreme values, so the reader can observe the problems which can occur in some extreme conditions.

Thanks to Adam Schreiber for promptly providing the model of airplane used in the example script

Model of F360 used in second demo and images in the article is free art from http://www.psionic3d.co.uk

References

[1]**Richard, Wright S., Lipchak, Benjamin a Haemel, Nicholas.** *OpenGL(R) SuperBible: Comprehensive Tutorial and Reference (4th Edition).* : Addison-Wesley Professional, 2007. 0321498828.

[2]**Schreiber, Petr.** *Realizace vybraných výpočtů pomocí grafických karet.* Brno : VUT, 2010.

[3]**Guinot, Jérôme.** Image Filtering with GLSL - Convolution Kernels. *oZone3D.Net Tutorials.* [Online] 8. 1 2006. [Citace: 2010. 8 23.] http://www.ozone3d.net/tutorials/image_filtering.php.