Pyramid Algorithm Framework for Real-Time Image Effects in Game Engines

Adrià Arbués Sangüesa[1], Andreea-Daniela Ene[1], Nicolai Krogh Jørgensen[1], Christian Aagaard Larsen[1], Daniel Michelsanti[1(✉)], and Martin Kraus[2]

[1] School of Information and Communication Technology, Aalborg University, Selma Lagerløfs Vej 300, 9220 Aalborg Øst, Denmark
{aarbue15,aene15,njarge12,cala10,dmiche15}@student.aau.dk
[2] Department of Architecture, Design, and Media Technology, Aalborg University, Rendsburggade 14, 9000 Aalborg, Denmark
martin@create.aau.dk

Abstract. Pyramid methods are useful for certain image processing techniques due to their linear time complexity. Implementing them using compute shaders provides a basis for rendering image effects with reduced impact on performance compared to conventional methods. Although pyramid methods are used in the game industry, they are not easily accessible to all developers because many game engines do not include built-in support. We present a framework for a popular game engine that allows users to take advantage of pyramid methods for developing image effects. In order to evaluate the performance and to demonstrate the framework, a few image effects were implemented. These effects were compared to built-in effects of the same game engine. The results showed that the built-in image effects performed slightly better. The performance of our framework could potentially be improved through optimisation, mainly on the GPU.

Keywords: Pyramid methods · Image effects · Depth of field · Blur · Bloom · Game engine · Texture lookup · Compute shader

1 Introduction

Pyramid methods have many applications within the image processing field. These methods are named pyramidal because they are based on the construction of a scale space of filtered levels of different resolutions of an image. From this pyramid structure, image effects can be generated through the use of filters and other image-based computations.

This paper is focused on creating a framework that allows users to develop image effects based on pyramid methods. The framework is implemented in the popular game engine Unity [1], but could ideally be implemented in any game engine that supports rendering to textures. Unity provides its own built-in image effects which are generally based on convolution methods using regular shaders. These are compared with the implemented image effects in terms of visual quality and performance.

A.L. Brooks and E. Brooks (Eds.): ArtsIT/DLI 2016, LNICST 196, pp. 289–296, 2017.
DOI: 10.1007/978-3-319-55834-9_34

We opted for using the free version of Unity because it is a widely used game engine that supplies functionality, which made the implementation and use of the framework easier to manage. Moreover, render to texture was recently added to the free version of the engine (Unity 5.0) enabling the users of this version to use pyramidal image effects. However, currently there is no built-in support for pyramid methods using render to texture functionality, which is the motivation behind implementing the framework.

In order to evaluate the performance of the framework, some image effects have been implemented: blur, depth-of-field, and bloom. The performance was evaluated by comparing the visual quality of the effects to the built-in image effects of Unity. This comparison is a subjective assessment based on side-by-side comparisons. At the same time, the effects were compared in terms of metrics such as memory, GPU, and CPU usage.

This paper has the following structure: in Sect. 2 the state of the art is detailed, explaining various pyramid algorithms that have been proposed in the past; then, in Sect. 3, the framework is explained, and in Sect. 4 several image effects are illustrated. Later on, in Sect. 5, the results are shown, and, after that, they are discussed in Sect. 6. Finally, conclusions are drawn and future work is suggested in Sects. 7 and 8.

2 Previous Work

The pyramid algorithm was proposed by Burt [2] who also showed its advantageous complexity compared to the Fast Fourier Transform (FFT) techniques for blurring. For this reason, pyramid images were used in many computer graphics applications. An example is the work of Kraus and Stengert [3] who applied GPU-based pyramid methods for blurring in their algorithm for depth-of-field rendering.

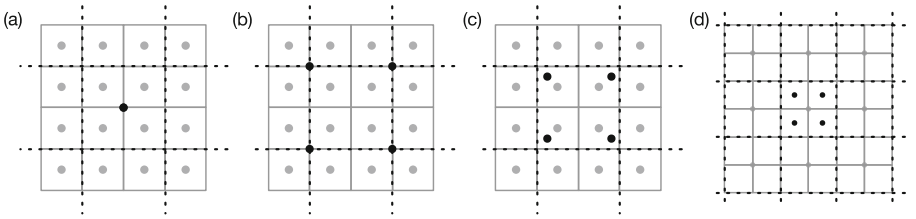

Fig. 1. The dotted lines represent the boundaries of the coarse level pixels, while the grey ones represent the boundaries of the fine level pixels. The grey dots represent the center of the pixels in the current level, while the black ones represent the positions of the bilinear texture lookups for analysis (a, b and c) and synthesis (d). (a) 2×2 analysis box filter using a single bilinear texture lookup. (b) 4×4 analysis box filter averaging the results of four bilinear texture lookups. (c) Biquadratic B-spline analysis filter averaging the results of four bilinear texture lookups. (d) Biquadratic B-spline synthesis filter using four bilinear lookups.

Image blurring with pyramidal methods consists of two steps: analysis and synthesis. In the first one, a pyramid is generated by iteratively downsampling the filtered image by a factor of two in each dimension. In the second step, a level of the analysis pyramid is chosen, based on the desired blur width, and the image is upsampled through a synthesis filter until reaching the original dimensions.

As illustrated by Kraus and Stengert [4], many pyramid filters based on bilinear interpolation may be used for both analysis and synthesis. Although the users of our framework may implement their own filters, we decided to provide three different analysis filters (2×2 box filter, 4×4 box filter, and biquadratic B-spline) and one synthesis filter (biquadratic B-spline). Figure 1 shows how the mentioned filters are implemented through bilinear texture lookups.

3 Framework

The implemented framework aims at supporting developers in creating pyramid-based image effects. The framework is intended for experienced and advanced users of Unity3D with the assumption that users have some level of shader programming experience. At the same time, the framework supplies a few ready-to-go image effects in order to demonstrate its use as well as to help less experienced programmers.

As mentioned, the framework uses compute shaders to perform all the image processing and to produce the image effects.

The implemented framework takes over a lot of the management required to instantiate and maintain frame buffer objects, implemented as render textures in Unity. These textures are ARGB32 textures when running Unity in gamma space; in linear space they are floating point textures because of color quantisation issues due to gamma correction and these are, of course, more expensive. Moreover, the framework provides access to methods that analyse and synthesise an image. Each image is sampled to a new texture with the nearest power of two (PoT) resolution and stored in a list. The general procedure of the framework is shown in Fig. 2.

Creating an instance of the framework exposes all the functionality of it to the image effect the developer wants to create. This step also generates the

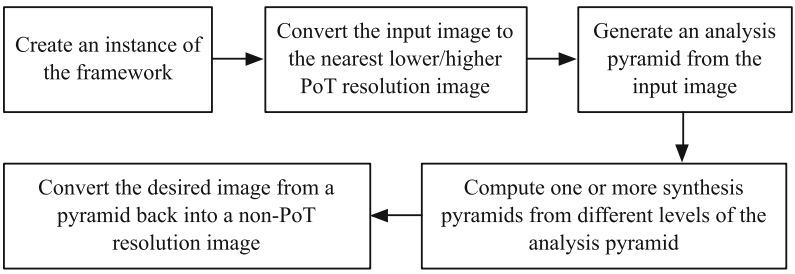

Fig. 2. Steps performed by the framework.

analysis pyramid based on the constructor that is used to generate the instance. The analysis pyramid is generated by taking an image in native resolution and downsampling it into a new texture with the nearest lower PoT resolution or copying it into a higher PoT resolution, depending on user preferences. The image is offset from the edge of the PoT texture and clamped to the edge by copying the edge pixels. This texture is then analysed using one of the analysis kernels described in Sect. 2.

Once the analysis pyramid has been computed, the users can generate one or more synthesis pyramids depending on their need. These pyramids can be generated through different constructors specifying source and, optionally, target levels or resolutions. The pyramid is generated using the specified synthesis filter described in Sect. 2. At this point, users can implement their desired image effects using the provided pyramids and once completed, the image will be converted back to a native resolution image.

The idea behind the framework is that it stores the pyramids allowing users access from anywhere in any script by name or resolution. Simultaneously, it ensures that the users are not able to generate duplicate pyramids. The framework consists of one class which handles the creation of all the textures needed for the pyramids as well as the dispatching of the compute shader. The compute shader then handles the transfer of pixel data between textures and the computation of effects. Another important feature of the framework is that it supports custom kernel creation, meaning that users are able to specify a kernel of the size they desire.

4 Applications of the Framework

The following sections present the three image effects provided with our framework, namely blur, depth of field, and bloom (Fig. 3).

4.1 Blur

The blur effect is meant to obscure details in an image by averaging pixels over a larger area. Using pyramids, the blur effect can be obtained by applying analysis

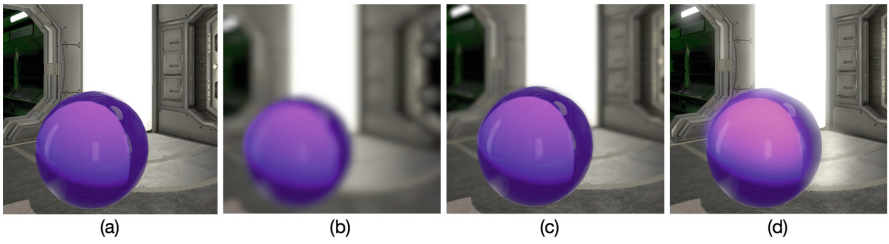

Fig. 3. The three image effects implemented using the framework, compared with the original image (a): blur (b), depth of field (c), and bloom (d).

and synthesis steps to an image. The results may change based on the number of levels of the generated pyramid and the kernels used to analyse and synthesise.

Our framework not only allows three different analysis filters and one synthesis filter (see Sect. 2 for details), but it also supplies support for custom kernels as mentioned in Sect. 3.

4.2 Depth of Field

Depth of field is defined as the difference between the nearest and the farthest planes between which pixels in an image appear sharp. Generally, pixels are considered sharp when they are in a position where the circle of confusion is not distinguishable from a point. Further explanations can be found in the work of Demers [5].

Using the framework described in Sect. 3 a simulation of depth of field was implemented. It is important to point out that this effect is not a physically accurate implementation of depth of field, but rather an approximation that exposes simple attributes which allow for modification of the effect.

The effect was implemented using the Reverse-Mapped Z-Buffer approach [5] by generating an analysis pyramid, with the framework, and using this to derive different amounts of blur. Each level of blur was derived by synthesising from lower levels of the analysis pyramid. In our effect, three blur widths were used.

From these three levels of blur, as well as the depth texture, depth of field was simulated based on the specified parameters of the effect (see Fig. 4). For each pixel in the image, its depth is looked up in the depth texture. Based on the depth, the pixel color is linearly interpolated from two textures. This happens for all pixels where the depth is within the A regions. In Fig. 4 this interpolation is expressed through the grey gradient. The rest of the image is copied either from the original image (in focus) or the maximum blur texture.

4.3 Bloom

Bloom is an effect generated by producing fringes of light extending from bright areas of an image. It produces an imaging artifact also seen in real world cameras, where bright light bleeds to nearby areas on a film.

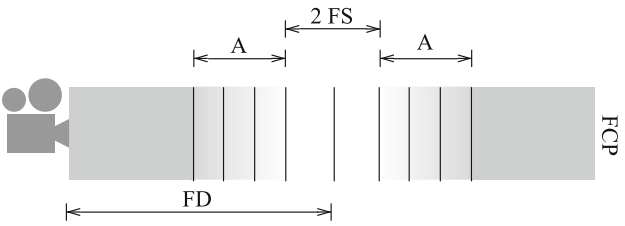

Fig. 4. Parameters of the implemented depth of field effect. FD: distance from camera to focus plane; FS: distance from focus plane to start of blur; A: distance over which the blur will gradually increase; FCP: far clipping plane.

Implementing this effect using the framework, as previously described, is achieved by taking the source texture and locating all the bright areas. They are located by taking the average value of the three color channels for each pixel and checking whether it is above a user-specified threshold. In this case the pixel is copied to a bloom texture, otherwise it is set to black. The bloom texture is then sampled to a lower PoT texture and synthesised based on the desired bloom strength. This synthesised texture is then combined with the source texture using the following formula:

$$z = 1 - (1 - x)(1 - y)$$

where x, y, and z represent a pixel of the original image, of the blurred image after bright colors extraction, and of the resulting image respectively.

5 Results

The framework and its effects were tested on a Lenovo Y-50 laptop with a Nvidia GeForce 860 m graphics card and an Intel Core i7 4710HQ 2.5 GHz processor with 16 GB of RAM running in full HD resolution (1920×1080) in gamma space.

All the results are gathered over 1000 frames while skipping the first 60 frames as these could potentially cause anomalies in the performance. From this, the average time used per frame can be calculated as well as the processing time of the framework. The image effects are also compared to the built-in counterparts in terms of visual quality through side-by-side comparisons.

In Fig. 5, the average processing time and the average frame time of the results are displayed, along with the memory usage of the framework and of the built-in effects in Unity.

To clarify, *Only Blit* is the Unity scene running with just a texture being blitted to the screen and this is logged to see what the basic cost of the scene is. *Framework* is the framework running without applying any image effect but simply computing one analysis pyramid and one synthesis pyramid. The rest are the built-in image effects of Unity compared to the image effects implemented with our pyramid framework.

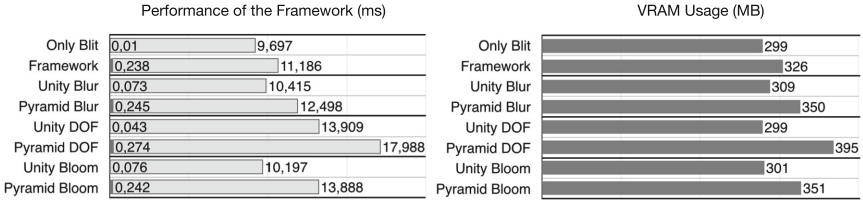

Fig. 5. Graphs showing the performance of the framework (left) and the total video memory usage on the graphics card when running the image effects. For the graph on the left, the dark grey bars represent the average CPU time in milliseconds as a part of the average frame time (light grey bars).

Looking at the average frame time from only blitting to running the framework (downsampling to 32×32 pixels), it can be seen that the basic cost of the framework is around 1.5 ms. For each of the image effect comparisons the framework performs slightly slower, where the biggest difference is in the bloom effect where the framework had the relatively worst performance. For the average processing time, the framework is fairly consistent; note that the depth of field effect is slower due to the creation of more synthesis lists.

Regarding the video memory usage, it can be seen that while the built-in effects barely use any additional memory, the framework footprint is directly tied to the amount of textures needed for each effect, as well as how many times the source texture is analysed and synthesised. Again, the depth of field effect is the most expensive effect, also due to the additional textures created.

Visually, the implemented image effects look relatively similar to the built-in image effects. However, as the procedure is different from one another (*i.e.* the built-in effects have more adjustable parameters), there are visible differences, especially in the bloom image effect.

6 Discussion

As the built-in effects of Unity are heavily optimised for the engine, we did not expect the image effects implemented with our framework to outperform them. This also proved to be the case as displayed in Fig. 5 where every effect was slower both on the CPU and the GPU. However, the current framework has room for improvement as it is a work in progress.

Although results are averaged over 1000 frames, they have some inaccuracy due to how Unity handles multithreading. Running two tests in a row could produce significantly different results, especially for the processing time, whereas average frame time generally stayed the same. The test would have to be iterated multiple times in order to get more accurate results. However, for the purpose of doing this initial testing of the framework, the collected results give an adequate representation of the performance.

Figure 5 shows, that the framework uses significantly more processing time than the built-in Unity effects due to the generation of the pyramids. However, when comparing this to the average frame time, it is an insignificant contribution to the overall performance cost. This in turn tells that the main performance bottleneck is the GPU implementation.

As it has been established that the framework's CPU cost is not a major factor of the performance, it would most likely be possible to optimise the framework by moving some logic and calculations from the GPU to the CPU side wherever possible.

As expected, the video memory usage of the framework is higher compared to the built-in effects. This is due to the use of multiple textures of different resolutions for the analysis and synthesis pyramids.

7 Conclusions

A pyramid algorithm framework capable of performing real-time image effects by using compute shaders in Unity has been developed. The framework is implemented based on previous work within the field and has functionality to support implementation of custom kernels. We showed how the framework performs by implementing certain image effects and compare them to their built-in counterpart in Unity. The performance was slightly worse than the built-in effects, an expected result considering that the framework is a work in progress.

8 Future Work

The performance of the framework could be improved in various ways. As mentioned earlier, the performance bottleneck is the GPU. The compute shader used by the framework applies many logic operations, such as branching logic, which could potentially lower the performance quite significantly. To avoid this, some of the kernels used in the computer shader would have to be split into separate kernels and dispatched appropriately from the CPU.

Moreover, the framework should supply some support for creating textures in order to make texture generation easier for the users. Currently, textures are made manually for each image effect. When implementing multiple image effects, some memory usage could be avoided by handling the textures in the framework instead of each image effect separately.

References

1. Unity Manual. http://docs.unity3d.com/Manual/UnityManualRestructured.html. Accessed Sept 2015
2. Burt, P.J.: Fast filter transform for image processing. Comput. Graph. Image Process. **16**(1), 20–51 (1981)
3. Kraus, M., Strengert, M.: Depth-of-field rendering by pyramidal image processing. In: Computer Graphics Forum, vol. 26. No. 3. Blackwell Publishing Ltd. (2007)
4. Kraus, M., Strengert, M.: Pyramid filters based on bilinear interpolation. In: GRAPP (GM/R), pp. 21–28 (2007)
5. Demers, J.: Depth of field: a survey of techniques. In: GPU Gems: Programming Techniques, Tips, and Tricks for Real-time Graphics, Chap. 23, pp. 375–390. Addison-Wesley Professional (2004)