Precomputed procedural content level of detail

V.V. Sanzharov[1], V.A. Frolov[2,3]
vsan@protonmail.com|vova@frolov.pp.ru
[1]Gubkin Russian State University of Oil and Gas, Moscow, Russia;
[2]Keldysh Institute of Applied Mathematics RAS, Moscow, Russia;
[3]Moscow State University, Moscow, Russia

In this paper we address the problem of determining the detail level of precomputed procedural content for photo-realistic rendering. We propose a rasterization prepass with an algorithm based on mipmapping to calculate the detail levels. Suggested solution is applied to computing resolutions of procedural and image textures and mesh subdivision levels.

Keywords: *procedural textures, photo-realistic rendering, level of detail.*

Introduction

Procedural content is a basis of todays cinema and VFX (visual effects) industrial pipeline because it allows to reduce the cost of manual content creation. This is also important in recent applications of photo-realistic rendering to the synthesis of images and video sequences for the purpose of using them as training data for artificial intelligence algorithms in computer vision [25] where manual content creation is infeasible because of the dataset sizes.

The procedural approach in material modeling is the synthesis of texture maps, which can be used to specify the values of any material model parameters, geometry (for example, displacement maps), and light sources. Methods of procedural generation are often classified as implicit or explicit [5]. Explicit methods calculate the whole texture in advance. These textures can be used further by the lighting modeling system as conventional images. Implicit methods define the texture as a function of some arguments, for example, from the coordinate in the texture or world space and calculate it "on the fly". Basically, the algorithm responds with a value to a query about particular sample. This approach implies the creation of some algorithmic description of the desired visual result.

Procedural texture synthesis overview

Implicit texture synthesis techniques have been used for quite a long time, one of the first major breakthroughs being Perlin noise algorithm [23] back in 1985, and are continually being developed (for example [7, 14, 20]). These methods have several advantages - texture maps are not stored in memory and don't have fixed resolution (the texture is infinite and has no seams). Also, textures can be parametrized with arbitrary values (for example, world space position) and therefore applied to geometry without uv-unwrapping. By changing the input parameters it's possible to get a large number of different texture variants. Disadvantages include: the complexity of the process of algorithmic formalization, the need for careful manual adjustment of parameters to achieve a desired result and independent texture samples evaluation. This restricts the generalization possibilities and therefore limits the variability in resulting textures. Still, implicit methods are widely used in computer graphics and are implemented in many software products (Houdini, Blender, Maya, 3ds Max, etc.).

In explicit texture synthesis texel values are interdependent and therefore can't be evaluated independently. A detailed review of early explicit methods is presented in [26]. Example based and most statistical texture synthesis algorithms are classified as explicit. Example based methods are being actively developed by improving existing algorithms [10, 11], applying neural networks [6, 18] and combining existing algorithms with neural networks [4, 17]. One of the main advantages of these methods is that they don't suffer from the problem of formalizing the desired result which is represented by an example image passed as an input to the algorithm. Among the shortcomings is a considerable amount of computations usually required for these methods, which directly depends on the output texture resolution. As reported in [13] running time for texture synthesis algorithms ranges from seconds to minutes and may even reach hours for large resolutions.

Existing problems

Resolution influences the size of the texture in memory, time needed to execute texture synthesis algorithm and image quality in rendering. Thus, for photo realistic rendering it is crucial to determine the texture resolution which will provide the best perceived quality with the smallest possible size in memory and computation time. Another problem arises when rendering system needs to support existing implementations of implicit texture synthesis methods in closed-source software like Maya and 3ds Max. This usually means that the renderer should call a virtual function provided by the software API to get the texture sample value. So, if the address space where the renderer is running differs from the address space of the procedural texture implementation (for example, the renderer runs on the GPU or on a different computer), it's simply not possible to query these proce-

Работа опубликована при финансовой поддержке РФФИ, грант 18-07-20045\18

dural textures during rendering like it should be done with implicit texture synthesis. Therefore, to support them the renderer developers need to take on a different approach. The first option is to implement analogous procedural textures in the rendering system and match their input parameters with existing software. This has to be done for every product the rendering system wishes to integrate into. And the second (and less expensive) option is to treat these procedural textures as explicit - that is precompute them before rendering. And again, this leads to the same problem - the need to determine the resolution.

In this paper we suggest an approach to estimate resolution for procedural textures precomputation for photo-realistic rendering systems. Proposed solution can also be used to resize bitmap textures and with some modifications applied to other preliminary computations like estimation of subdivision level for meshes subject to displacement.

Related work

Existing approaches for the most part are concerned with level of detail (and therefore resolution) estimation for image textures in real-time rendering. Conventional solutions to the problem of texture resolution selection are based around mipmapping [28] and involve storing several copies of an image with different resolutions - mipmap pyramid, out of which appropriate mip level is selected at rendering time usually based on the projected size of a textured object. There are methods enabling parallel synthesis of different miplevels for some of the explicit procedural textures [8, 16, 27]. Some methods focus on performing relatively small amount of preliminary computations which are then used to speed up subsequent synthesis of the whole image pyramid [3]. An optimization to mipmapping called clipmapping [24] is based on the idea that for a large texture not all data of a mipmap is used in rendering of a particular frame so it is sufficient to only store parts of a mipmap which are potentially visible - clipped mipmap. More complex approach - virtual texturing [1, 15, 22] is based around splitting texture into uniform tiles and loading into video memory only the tiles needed to render a particular frame using visibility information. Virtual texturing also received hardware support [2] and is available in Vulkan API and as OpenGL ARB_SPARSE_TEXTURE extension [30].

Limitations of existing solutions

Existing solutions to determine the texture resolution focus on application to real-time rendering and are designed to be implemented as a part of a rendering pipeline. Since we need to estimate resolution for precomputed procedural textures before rendering, these approaches don't solve the problem, at least by themselves. The other challenge lies with the rendering pipeline integration - to implement any of these methods in an existing rendering system, the rendering system itself would obviously require changes which can be substantial. For example, when the mesh has more than one texture applied to it, using virtual texturing requires using texture atlases [21]. Since rewriting already established parts of a complex system is often undesirable, we consider this as a disadvantage of direct implementation of these methods in the renderer. There are also other possible problems with the approaches mentioned earlier like visual artifacts caused mainly by filtering [1, 22]. For some texture types, for example glossiness textures and normal maps, mipmap filtering can lead to significant losses in detail [29]. This is not acceptable in photo-realistic rendering since details are one of the foundations of a realistic image.

Suggested approach

The essence of the suggested solution is to separate all the necessary precomputations from the rendering system into a simple scene rasterization prepass followed by texture synthesis. This prepass would determine the resolution of the each texture in a scene from mip levels. And mip levels are computed using a slightly modified implementation of the approaches suggested in [1] and provided in OpenGL API specification [30]. Now let's consider the whole proposed process.

Implementation details

First of all, we will need to store function pointers and input parameters for all procedural textures used in the scene we are going to render. This way we'll be able actually synthesize them after we determine the needed resolution.

The next step is to rasterize the scene geometry with the information on the assigned materials. Rasterized scene allows to determine texture coordinates gradient for all textured objects and thus the mip level. The equations for gradient (1, 2) are similar to those for mip level calculation in [1] and OpenGL API specification [30].

$$G_x(uv) = \frac{R\_rast_x}{R\_rend_x} * tex\_res * \frac{\partial u}{\partial x} \quad (1)$$

$$G_y(uv) = \frac{R\_rast_y}{R\_rend_y} * tex\_res * \frac{\partial v}{\partial y} \quad (2)$$

R_rast_x, R_rast_y – rasterization resolution, R_rend_x, R_rend_y – photo-realistic rendering resolution, tex_res – texture resolution, u, v – texture coordinates, x, y – rasterized image coordinates.

However, for procedural textures tex_res is unknown and it is actually the value we want to determine. So, let's assume for the moment that it is an arbitrarily high resolution which we want to reduce. Also we need to take into account ratio be-

The publication of the conference proceedings was supported by the RFBR, grant No. 18-07-20045\18

tween rasterization resolution (R_rast_x, R_rast_y) and actual photo-realistic rendering resolution (R_rend_x, R_rend_y) since these can be different - rasterized image only serves a utility function and there is no need for it to be high resolution while photo-realistic rendering obviously can have arbitrary resolution.

The gradients are then used to compute the mip level mip_lvl (3) analogous to OpenGL specification [30] and [1]:

$$mip\_lvl = log_2[max(G_x, G_y)] \quad (3)$$

After mip levels are calculated we find the maximum mip level for each texture in the scene - the same textures can have different materials mip levels in different materials and on different geometry. So, we basically retain only the maximum level from the mip pyramid for each texture. Required resolution R is then calculated with (4).

$$R = \frac{tex\_res}{2^{mip\_lvl}} \quad (4)$$

After substituting the mip_lvl in (4) with (3) and (1, 2) we get the following (5):

$$R = \frac{tex\_res}{max(\frac{R\_rast_x}{R\_rend_x} * \frac{tex\_res}{\frac{\partial u}{\partial x}}, \frac{R\_rast_y}{R\_rend_y} * \frac{tex\_res}{\frac{\partial v}{\partial y}})} \quad (5)$$

It's obvious, that tex_res can be eliminated, yielding the final equation for the resolution (6):

$$R = max(\frac{R\_rend_x}{R\_rast_x} * \frac{1}{\frac{\partial u}{\partial x}}, \frac{R\_rend_y}{R\_rast_y} * \frac{1}{\frac{\partial v}{\partial y}}) \quad (6)$$

All that is left is to pass the obtained resolution to the stored texture synthesis functions.

The same procedure can be applied to the «ordinary» image textures to resize them to the 1-to-1 pixel-to-texel ratio for the sake of possible memory saves. It's straightforward to incorporate different maximum allowed mip levels for different texture types, for example, to force the reflection or bump textures to be generated in higher resolution than diffuse textures.

Limitations and special cases

There are several special cases which cannot be tackled by suggested approach directly. The first such case is a scene with transparent objects - with straightforward rasterization they can potentially occlude other objects and interfere with detail level calculation. This situation can be handled by performing two rasterization passes for detail level estimation - one for all non-transparent objects and the other for transparent objects.

The second more complex case concerns textured objects visible as reflections (possibly after multiple reflections/refractions). This is a very difficult situation for rasterization and multiple bounces are nearly impossible to handle. Within the framework of our approach it can be addressed heuristically - for a reflected object we can safely assume that it won't be larger than the screen resolution and resize the textures accordingly. Obviously, it's not very efficient but still allows for possible memory savings. Another heuristic would be to assign to not directly visible objects the level of detail computed for the largest visible reflective object (i.e. mirror) in the scene.

A promising way for solving the challenge of reflections more rigorously is using techniques like ray differentials [9] with an upcoming real-time ray tracing technology [31].

Application to other precomputed effects

Proposed solution can also be applied to other precomputed effects, in particular to compute subdivision level for meshes. Since in the essence, texture mip level and geometry subdivision level both represent the same notion of detail level. For example, in [19] clipmapping ideas were applied to mesh subdivision in terrain generation from a height map. The goal of precomputing mesh subdivision instead of evaluating it at render time is 1) to do the computation only once in the case of rendering on multiple machines and 2) to keep the rendering system implementation simpler. To utilize the rasterization prepass described earlier to calculate the per-mesh subdivision factor some modifications are required. First of all, the gradient should be calculated for the positions, not the texture coordinates. This way we can obtain the detail level per mesh. To get different subdivision levels on the same big mesh (like terrain), it needs to be split into regions, for example as proposed in [19].

Results and discussion

We tested our approach on 3 scenes (fig. 1) which use procedural and image textures. First we rendered scenes with image textures in their respective native resolution and all procedural textures with manually picked reasonable resolutions. Rendered images obtained this way served as references. Then, we rendered the same scenes with the same image and procedural textures but the rasterization prepass was executed to determine the resolution for textures before actual rendering. Scenes were rendered with our own C++/OpenCL path tracing implementation on the GPU. The comparison of the results with the references in terms of overall memory needed for textures (table 1) showed several times lower memory size with rasterization prepass. Visual perception comparison (fig. 2, fig. 3, fig. 4) shows no perceptible differences as confirmed by mean square error (MSE) values (table 2). We also tested our approach in application to computation of the subdivision level (fig. 5, fig. 6) using non-adaptive sqrt 3 [12] subdivision algorithm. First, terrain generated from height map was subdivided before displacement with the same level across all three parts (separate meshes). Next, modified rasterization prepass was used to determine the subdi-

Работа опубликована при финансовой поддержке РФФИ, грант 18-07-20045\18

vision levels per mesh. Applying suggested prepass allowed to reduce the amount of memory needed for geometry from 185 Mb to 89 Mb while maintaining image quality.

Conclusion

Proposed approach allows for tangible memory savings compared to naive approach of simply synthesizing procedural textures with a same fixed resolution and passing images into rendering system as they are.

Scene, rendering resolution	Ref., Mb	Ours, Mb	Ratio
Arch, 1920 x 1080	276	156	1.77
Arch, 3840 x 2160	276	173	1.60
Bathroom, 1024 x 1024	521	142	3.67
Bathroom, 2048 x 2048	521	178	2.93
Alley, 1024 x 1024	527	129	4.09
Alley, 2048 x 2048	527	146	3.61

Table 1. Memory used by textures

Scene	MSE
Alley	1.67
Bathroom	3.59
Arch	3.02

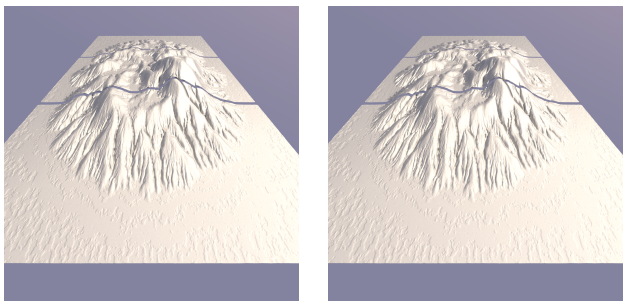
Table 2. Mean Square Error (MSE) for rendered images (computed after tone mapping, i.e. in low dynamic range)

Figure 1. Test scenes - Arch, Bathroom, Alley. Arch and Alley - primarily procedural textures, Bathroom - high resolution bitmaps

Figure 2. Reference vs With resized textures, Bathroom

Figure 3. Reference vs Difference vs With resized textures. Bathroom scene, in top row - bitmap diffuse texture, bottom row - bump map computed from bitmap

Figure 4. Reference vs Difference vs With resized textures. Alley scene, procedural noise-based texture on an object with one of the texture coordinates stretched out



geom. size in memory: 185 Mb 89 Mb

Figure 5. Terrain subdivision with uniform subdivision levels vs subdivision levels computed by prepass, terrain mesh generated from height map

The publication of the conference proceedings was supported by the RFBR, grant No. 18-07-20045\18

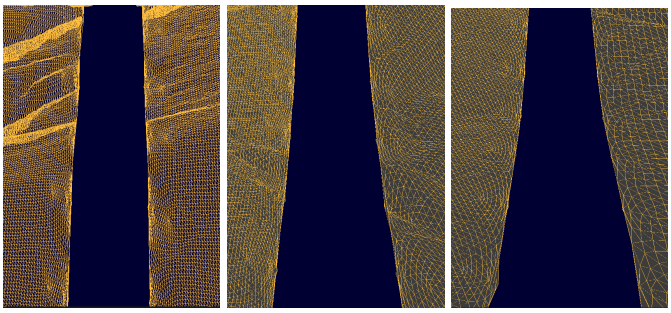


Figure 6. Neighboring mesh grids fragment. Left to right: no prepass applied - uniform subdivision levels; reduced by prepass - the closest and the middle meshes; reduced by prepass - the middle and the farthest meshes. The farther the mesh is from the camera - the smaller its subdivision levels and therefore more sparse mesh grid

Acknowledgments

This work was sponsored by RFBR 16-31-60048 «mol_a_dk».

References

[1] Barrett S. *Sparse virtual textures* http://silverspaceship.com/src/svt/

[2] Bilodeau B., Sellers G. and Illesland K. *Partially Resident Textures on Next-Generation GPUs* // Game Developers Conference, 2012.

[3] Dong Y., Lefebvre S., Tong X. and Drettakis, G. *Lazy Solid Texture Synthesis* // Computer Graphics Forum, 2008, 27: 1165-1174

[4] Dong J., Wang L., Liu J., Sun X. *A Procedural Texture Generation Framework Based on Semantic Descriptions* // arXiv:1704.04141, 2017

[5] Ebert D.S., Musgrave F.K., Peachey D., Perlin K., and Worley S. *Texturing and Modeling: A Procedural Approach.* // Morgan Kaufmann, 1998.

[6] Gatys L.A., Ecker A.S., Bethge M. *Texture Synthesis Using Convolutional Neural Networks* // arXiv:1505.07376, 2015

[7] Gilet G., Sauvage B., Vanhoey K., Dischler J.-M., and Ghazanfarpour D. *Local random-phase noise for procedural texturing* // ACM Trans. Graph., 2014, Vol. 33(6), Article 195

[8] Han C. et al. *Multiscale texture synthesis* // ACM Trans. Graph., 2008, Vol. 27(3), p. 51.

[9] Igehy H. *Tracing Ray Differentials* // Proceedings of ACM SIGGRAPH, 1999, pp. 179-186.

[10] Jamriška O., Fišer J., Asente P., Lu J., Shechtman E., and Sýkora D. *LazyFluids: appearance transfer for fluid animations* // ACM Trans. Graph., 2015, Vol. 34(4), Article 92

[11] Kaspar A., Neubert B., Lischinski D., Kopf J. *Self Tuning Texture Optimization* // Computer Graphics Forum 34(2), 2015

[12] Kobbelt L. *√ 3-subdivision* // Proceedings of ACM SIGGRAPH, 2000

[13] Kolář M. , Debattista K. and Chalmers A. *A Subjective Evaluation of Texture Synthesis Methods* // Computer Graphics Forum 36: 189-198, 2017

[14] Lefebvre L., Poulin P. *Analysis and synthesis of Structural Textures* // Proceedings of Graphics Interface 2000, pp. 77-86

[15] Lefebvre S., Darbon J., Neyret F. *Unified texture management for arbitrary meshes* // Technical Report RR5210-, INRIA, may 2004

[16] Lefebvre S., Hoppe H. *Parallel controllable texture synthesis* // ACM Trans. Graph., 2005, Vol. 24(3), pp. 777-786.

[17] Li C., Wand M. *Combining Markov Random Fields and Convolutional Neural Networks for Image Synthesis* // arXiv:1601.04589, 2016

[18] Li Y., Fang C., Yang J., Wang Z., Lu X., Yang M.-H. *Diversified Texture Synthesis with Feed-Forward Networks* // IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, p. 266-274

[19] Losasso F., Hoppe H. *Geometry clipmaps: terrain rendering using nested regular grids.* // ACM Trans. Graph., 2014, Vol. 23(3), pp. 769-776.

[20] Maung D., Shi Y., Crawfish R. *Procedural textures using tilings with Perlin Noise.* // 60-65. 10.1109/CGames.2012.6314553

[21] Mayer A.J. *Virtual texturing* // Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Oct. 2010.

[22] Mittring M. et al. *Advanced virtual texture topics* // ACM SIGGRAPH 2008, 2008, p. 23-51.

[23] Perlin K. *An Image Synthesizer* // Proceedings of ACM SIGGRAPH, pp. 287–296, 1985

[24] Tanner C.C., Migdal C.J., Jones M.T. *The clipmap: a virtual mipmap* // Proceedings of ACM SIGGRAPH, 1998, p. 151-158.

[25] Tsirikoglou A., Kronander J., Wrenninge M., Unger J. *Procedural Modeling and Physically Based Rendering for Synthetic Data Generation in Automotive Applications* // arxiv.org:1710.06270v2, 2017

[26] Wei L.-Y., Lefebvre S., V. Kwatra, G. Turk *State of the Art in Example-based Texture Synthesis* // Eurographics 2009, EG-STAR 2009

[27] Wei L. and Levoy M. *Order-independent texture synthesis* // Tech. Rep. TR-2002-01, Stanford University CS Dept.

[28] Williams L. *Pyramidal parametrics* // ACM SIGGRAPH Computer Graphics, vol. 17, no. 3, pp. 1–11, Jul. 1983

[29] Yan L.Q., Hašan M., Jakob W., Lawrence J., Marschner S., Ramamoorthi R.*Rendering glints on high-resolution normal-mapped specular surfaces* // ACM Trans. Graph., 2014, Vol. 33(4), p. 116.

[30] *Khronos registry* https://www.khronos.org/registry

[31] *Nvidia RTX* https://developer.nvidia.com/rtx

Работа опубликована при финансовой поддержке РФФИ, грант 18-07-20045\18