©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.5.SDA-042

Stereoscopy-based procedural generation of virtual environments

Manlio Scalabrin, Laura Anna Ripamonti, Dario Maggiorini, Davide Gadia; Department of Computer Science, University of Milan, Italy

Abstract

The procedural generation of virtual scenes (like e.g., complex cities with buildings of different sizes and heights) is widely used in the CG movies and videogames industry.

Even if stereoscopy is often used in the visualization of these kind of scenes, nevertheless it is not currently used as a tool in the procedural generation, while a more comprehensive integration of stereoscopic parameters can play a relevant role in the automatic creation and placement of virtual models.

In this paper, we investigate how to use stereoscopy to control the procedural generation of a scene in an open-source modeling software. In particular, we will use the stereoscopic parameters to automatically place objects inside the stereoscopic camera frustum avoiding to reach an excessive parallax on screen, and we will show how to procedurally detect and solve window violations, by means of the automatic placement of a dynamic floating window in the image.

Introduction

In the last years, the production of stereoscopic contents has seen a relevant increment. This has lead to the necessity of developing specific tools to optimize the production pipeline and to improve the quality of the final products. Moreover, a large part of the experts involved in the research and production of stereoscopic contents has urged the necessity to consider stereoscopy as a technical and creative tool which must be included in all the pipeline steps, rather than just an effect to add in post-processing. These aspects have been addressed in several works addressing the production of stereoscopic movies [1, 2, 3] and videogames [4, 5, 6]. Only the integration of stereoscopy in the core of the production pipeline can allow a complete achievement of its expressive and communication power in the production of movies, videogames and Virtual Reality applications [7].

Regarding the production of stereoscopic images using Computer Graphics (CG), the most part of the softwares used for modeling and animation has specific tools for stereoscopy, like e.g. a virtual stereoscopic camera, the visualization of the stereoscopic camera frustum volume and of the convergence plane, the possibility to automatically render and composite the left and right views.

In this paper, we address the specific case of how stereoscopy can be used in the field of procedual modeling.

The procedural generation of virtual models and scenes is widely used in the CG movies and videogames industry [8]. By means of a parametric procedure (a mathematical formula, a set of rules applied to a set of basic shapes, etc.), a complex model or a scene filled with a large number of elements can be created without the need of a long process of manual mesh modeling [9, 10]. Well-known examples of procedurally generated models are plants [11], and cities with buildings of different sizes and heights, streets, etc. [12, 13, 14]. Procedural generation is also used in videogame development, e.g. to automatically create a game level placing platforms and obstacles to achieve a desired level of complexity [15].

Even if very different parameters and rules can be used in the procedural modeling field, to our knowledge stereoscopy has not yet be considered as an actual factor in the generation process, while a more comprehensive integration of stereoscopic parameters can improve in a relevant way the production of effective and visually pleasant virtual scenes.

In this paper, we aim at investigating if some stereoscopic parameters and techniques can be included in a procedural modeling process inside a CG production software. In the next section, we will describe the tools and the test scene we have considered to evaluate the efficacy of the integration of stereoscopy in the procedural creation of a complex CG scene, and we will describe two applications of stereoscopic parameters in the automatic generation of models.

Application of stereoscopy in the procedural generation of CG scenes

Tools and test scene

Even if many CG production solutions are currently available, the choice of softwares providing tools for procedural modeling is currently limited. In many cases, the available softwares are specialized tools focused on the procedural generation of very specific scenes, like cities in the case of *Esri CityEngine* [16], while for general-purpose modeling and animation the most known and widely used software is *Houdini* [17].

After an evaluation of the available procedural-based softwares, we have found that the current solutions do not provide enough access to the stereoscopic parameters (if present) for our purposes. Thus, we have decided to consider for this work an open-source solution, in order to have the maximum flexibility and access to all the internal pipeline stages of the production tool in the development and test of the proposed integration between procedural modeling and stereoscopic parameters, at the cost of a possible minor stability due to the current development stage of the chosen softwares. We have decided to use the well-known open-source *Blender* software [18], which has an integrated stereoscopic pipeline since version 2.75 (released in July 2015), providing stereoscopic camera, stereoscopic preview in the modeling window, and compositing presets to save in different stereoscopic formats. Blender is a general-purpose modeling and animation software, and it does not provide specific procedural techniques. However, these functionalities can be included by

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.5.SDA-042

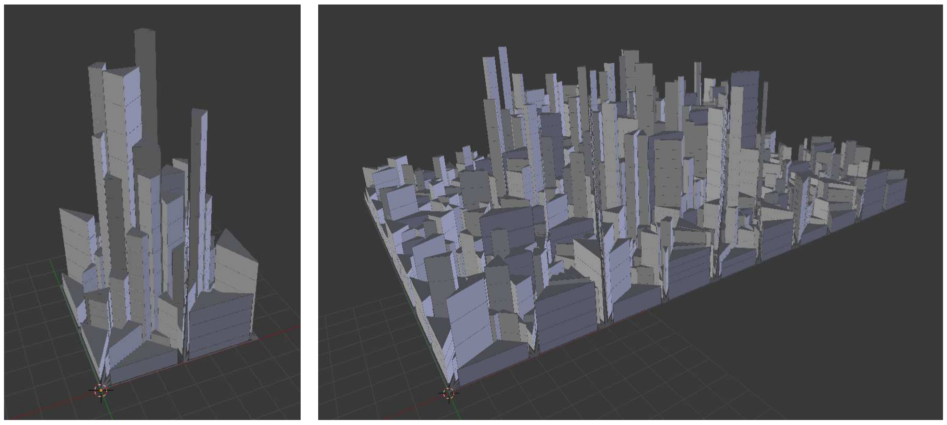

(a) (b)

Figure 1. *An example of geometry generated using Sverchok. Fig. 1(a) shows a single procedural block of a city, while Fig. 1(b) shows the final result created by the engine assembling several blocks with different characteristics.*

means of the recent *Sverchok* add-on [19], which extends the Blender node-based visual creation tool (used for compositing and procedural shader material editing) with operations on the meshes geometry.

Sverchok development is still in beta stage, but its functionalities allow to generate several kind of procedural scenes of different complexity. To test our approach, we have decided to consider a simple city scene. The city is composed assembling several blocks created procedurally with different parameters. For each block, a quadrangular patch is procedurally subdivided in streets and building lots, and the geometry of each new building is then generated on the basis of different parameters regarding heights and size, set by the user. The heights of the building are controlled both locally inside a block, by randomly creating adjacent buildings with different characteristics, than globally in the assembled city, by selecting a *“downtown”* area characterized by higher buildings, and tuning the other heights on the basis of the distance from the *“downtown”* blocks. Once created the geometry, facade textures are automatically applied. Fig. 1 shows examples of the geometry procedurally generated using Sverchok.

The procedure, whose exhaustive description goes out of the scope of this paper, does not include advanced parameters like e.g., density of population or geographical constraints, considered in other works [12, 13, 14, 16], but it allows the generation of an adequately complex scenario for our purposes. The test scene considered in this paper is composed by approximately 2750 buildings (more than 109000 vertices and 171000 triangles considering also the streets). The goal of the paper, i.e. investigating the efficacy of the integration of stereoscopy as one of the possible parameter to consider in the procedural generation, is not dependent on the nature and the complexity level of the generated scene. For this reason, and to achieve a reasonable rendering time with an average lab workstation during the development and test stages, we have decided also to limit the quality of the final rendering of the test scene (see Fig. 2 and 5 for some examples).

Procedural control of parallax

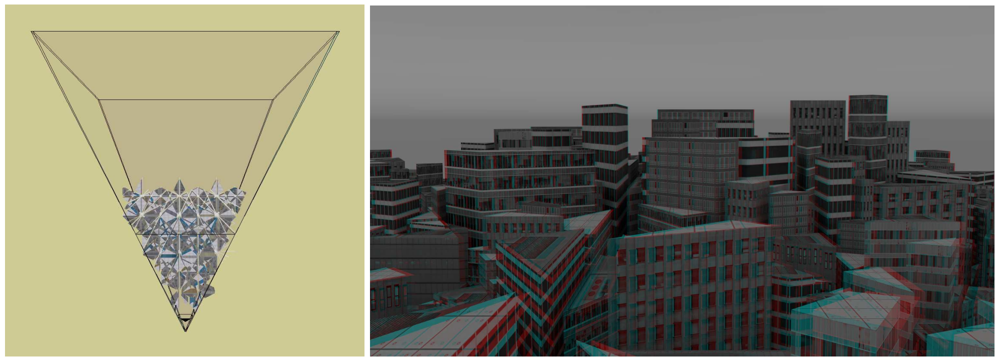
Starting from the generation of the test scene described in the previous subsection, we began to consider the introduction of stereoscopic parameters in the procedural process. The first example of application is conceptually quite straightforward: to control and limit the position of the generated geometry on the basis of the stereoscopic camera frustum volume and of the resulting parallax on screen.

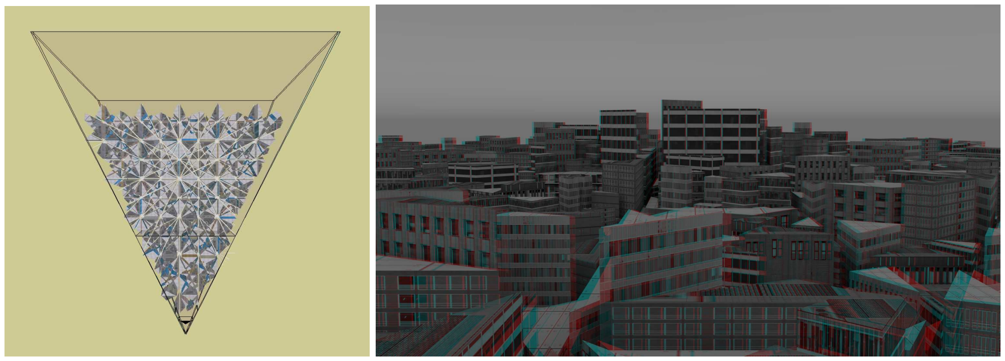
The amount of parallax on screen depends on the interaxial distance of the stereoscopic camera setup, on the distance of the convergence plane from the camera, and on the magnification and size of the screen used for the visualization of the final image [20]. Regarding a particular object, its parallax on screen increases depending on its distance from the camera. It is well known [1, 2] that the parallax on screen is one of the most important parameter to control, because if it exceeds the maximum positive parallax value *MPP* (2.56 in/6.5 cm), than the resulting image will be painful to view.

In the proposed approach, we suggest to calculate the position of the new procedurally generated vertices in *camera space* (i.e., using the camera position and its local coordinate system as a reference), in order to limit their z coordinates (i.e., their distance from the camera) to the distance corresponding to the maximum positive parallax *MPP*. Thus, this approach automatically prevents configurations with possible visual discomfort, but it gives also the possibility to the user to tune the amount of geometry created in the scene and, as a consequence, the final depth range of the scene.

Most of the modeling softwares providing stereoscopic support allow the visualization of the stereoscopic camera frustum volume, of the convergence plane, and of the maximum parallax plane, on the basis of the parameters chosen by the user regarding the stereoscopic camera and the target visualization setup. However, these visual aids are usually given just to help the user during the manual positioning of the objects in the scene, and not all of them are accessible as parameters from the software GUI and/or the development API. As a consequence, it is not possible to easily use these values in a procedural generation approach. In our tests, we had to develop a specific Python script in Blender in order to calculate the position of the maximum parallax plane of our stereoscopic setup: with this value available, we could add a procedure in the procedural city generation in Sverchok in

DOI: 10.2352/ISSN.2470-1173.2016.5.SDA-042


(a) $sff = 0.5$


(b) $sff = 0.8$

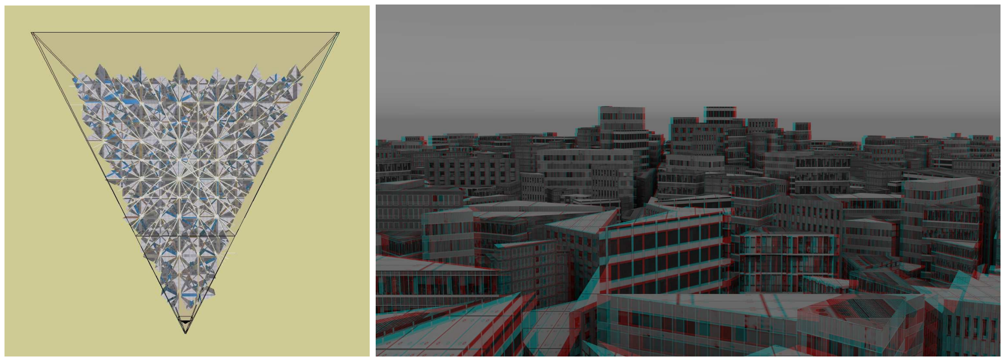

(c) $sff = 0.95$

Figure 2. *The procedural engine fills the stereoscopic camera frustum with geometry according to the value of* sff *(see on the left a top view of the generated scene). With* $sff = 1.0$*, the parallax on screen of the models on the far background will be approximately equal to the maximum parallax on screen achievable before divergence occurs. Images on the right can be viewed with anaglyph red-cyan glasses. Original anaglyphs have been created for a large screen FullHD projection. No correction for window violations has been applied.*

order to calculate the positions using camera space, and to limit the positions inside the stereoscopic camera frustum.

In particular, we have introduced a new parameter called sff (for *stereoscopic frustum filling*), and a new procedural rule for the generation of z coordinates of the new geometry, by limiting the range of coordinates to the set Z_{sff} defined as :

$$Z_{sff} = \{z \mid z \in [z_{min},\ \mathrm{sff} \cdot z_{MPP}]\} \qquad (1)$$

where $sff \in [0.0, 1.0]$, and z_{MPP} is the distance from the stereoscopic camera to the maximum parallax plane. Fig. 2 shows some examples of the effect of the changes of sff parameter.

This approach gives to the user an active control on the final effect generated by the procedural engine. This control can be extremely useful in case the stereoscopic material has to be re-edited, re-rendered, or adjusted for a different visualization setup, because it allows to adjust the amount of parallax on screen of the background objects with a single parameter. In future works, a possible extension of this technique can be the automatic setup of a multirigging stereo camera setup for $sff > 1.0$.

In equation 1, z_{min} is the closest distance where a vertex can be created by the procedural engine. Different choices can be made to set its value: for example, it can be set to the position

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.5.SDA-042

of the convergence plane, to avoid the automatic positioning in the negative parallax area and, as a consequence, problems with window violations [1, 3], or it can be just set to the value of the camera *near* plane. The next subsection will address the detection of window violations, and the automatic application of a *floating window* [3] by the procedural scene generator.

Procedural Floating Window

Window violation is a well-known problem in the stereoscopic production field [1]. However, an appropriate use of the dynamic floating window technique [3] can avoid visual discomfort in the final visualization. The method is based on the application of black masks at the borders of the frame to cover the visual information leading to retinal rivalry. These masks are dynamic (i.e., their characteristics can be adapted frame by frame), and different interesting creative solutions have been proposed to visually engage the viewer, like the application of the masks also to the top and bottom borders, or the tilting of the floating window to obtain asymmetrical masks [3].

The application of the dynamic floating window is already included as a standard tool in many softwares for stereoscopic production: in most of the cases, the black masks are applied to the stereoscopic frame in post-processing. In this paper, we have also investigated how window violations can be detected and managed in the procedural generation of a complex stereoscopic scene. As suggested in the previous subsection, a possible straightforward solution can be to avoid the procedural positioning of geometry in the negative retinal rivalry areas, in order to avoid any possible window violation. However, in many cases this approach can be too restrictive, limiting the stereoscopic effect and quality of the final images.

To this aim, we propose a method to procedurally detect and solve window violations, by means of the automatic application of the dynamic floating window technique in the scene. The approach is based on considering the black masks as actual 3D models in the scene, and to appropriately set their positions and size in each frame in order to make them occlude the scene geometry in window violation condition in the final rendering of the stereoscopic frame. The proposed method, which considers only lateral not-tilted masks, integrates stereoscopic parameters with the modeling tools already available in most CG production softwares, and it is based on four steps, described in the next sections.

Stereoscopic camera frustum as a 3D model

The proposed method is based on the application of *boolean operators* between the stereoscopic camera frustum and the geometry of the scene. The use of boolean operators is common in CG since *Constructive Solid Geometry (CSG)* [21]; in current modeling tools, it is possible to apply them also to complex meshes, or to use them for advanced selection of vertices.

In particular, we want to consider the left and right negative retinal rivalry frustums as actual 3D models in our procedurally generated city. Because of the limitations in the access and management of the parameters of the stereoscopic setup, described in the previous subsection, we again had to develop a specific Python script in Blender in order to create the meshes corresponding to the frustums. To this aim, we use some basic trigonometry in camera space using the camera FOV, the known z coordinate of the convergence plane, and the previously calculated z coordinate

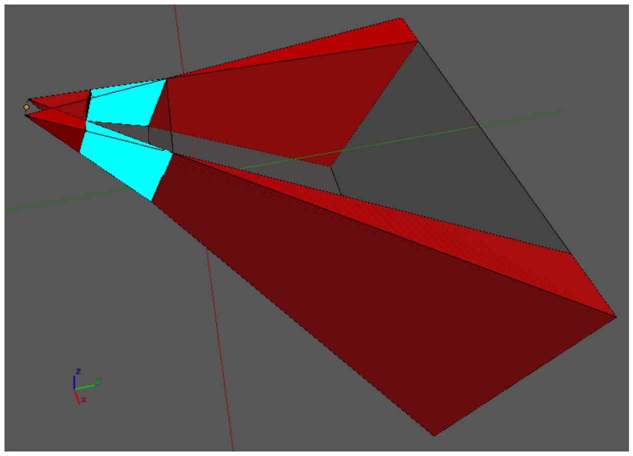


Figure 3. The meshes corresponding to the retinal rivalry frustums of the stereoscopic camera. The negative rivalry areas, used for the automatic placement of the black masks, are shown using a lighter color.

of the maximum parallax plane, in order to determine the three-dimensional coordinates in camera space of the vertices of the frustums. The implemented script uses these vertices to create inside the scene 3D models corresponding to the stereoscopic camera frustum. These models are not considered during the rendering stage, but they are just used in the boolean operations with the geometry of the scene. Fig. 3 shows the generated meshes corresponding to the positive and negative retinal rivalry frustums. In the proposed technique, we have considered the negative retinal rivalry areas, shown with a lighter color in Fig. 3.

Determining window violations

Once created the frustums meshes, we can determine the occurrence of window violations by finding the presence of geometry (models, or part of models) inside the negative retinal rivalry frustums. This control can be achieved using an *intersection* boolean operation between the procedurally generated geometry of the scene, and the left and right negative rivalry meshes.

If M is the set of the models procedurally generated in the scene (in our test scenes, the buildings and the streets of the city), then we can describe the operation as:

$$\mathrm{m}_{wv} = \mathrm{m} \cap \mathrm{RF}^{-}, \ \forall \mathrm{m} \in \mathrm{M} \tag{2}$$

where RF^{-} is one of the negative rivalry frustums (the operation must be applied separately for both the left and right rivalry areas). It must be noticed that m_{wv} can be a different model than the input one, with a different geometry and topology. The boolean operator gives as result only the part of the model which is inside the frustum; if the input model is only partially inside, then new vertices and edges are created, splitting the model in order to discard the part of geometry which is outside the volume. This is an important aspect, because it can affect the positioning of the lateral mask, which must be placed just in front of the closest vertex to the camera inside the negative retinal rivalry frustum.

This operation can be computationally expensive, if thousand of models have been procedurally generated in the scene. However some optimizations can be implemented, like applying frustum culling to discard models outside the whole stereoscopic

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.5.SDA-042

camera frustum, and not considering models placed behind the convergence plane, as they are surely not involved in window violations. Moreover, if a model produces a window violation on the left of the frame, it cannot be in the same situation on the right, and viceversa. Therefore, if equation 2 is applied considering the left RF^-, and it does not return an empty set, then the operation with the right RF^- can be ignored.

Determining the position of the lateral masks

Having determined the geometry producing window violations, the following step is to calculate the z coordinates in camera space of the two lateral masks to be created in the scene. For each negative rivalry frustum, the corresponding mask must be placed in front of the closest vertex to the camera, in order to occlude all the geometry causing the violation.

The closest distance can be found applying a sorting algorithm on the z coordinates of the set of vertices passed by the boolean operation described in the previous subsection, or a more efficient method can be applied, by running a preliminar *depth rendering pass*, an operation which saves in a *z-buffer* the depth information for the objects based on the distance from the camera. With this approach, it is sufficient to search in the *z-buffer* the point with minimum depth from the camera.

Determining the size of the lateral masks

In the final step, the two lateral masks are created at the positions determined in the previous operation, and they are opportunately sized in order to occlude only the geometry causing the window violation. The same operation must be applied separately for the left and right mask.

First of all, a quadrangular patch, parallel to the convergence plane, is created at the previously found minimum distance from the camera. The initial dimensions of the patch are set in order to cover the whole height of the final render in any situation, while the width is initially set to cover the whole convergence plane. A basic shader material is assigned to the patch, which returns a full black color, not applying any illumination model. As a consequence, this will lead to a full black color in the final rendering of the scene.

Once created the patch, we have applied another *intersection* boolean operation between the patch and the negative rivalry frustum mesh. If $ptch_{z_{min}}$ is the patch generated at the minimum distance from the camera for the left or right frustum, then we can describe the operation to create the final lateral mask fw as:

$$\mathrm{fw} = \mathrm{ptch}_{z_{min}} \cap \mathrm{RF}^- \qquad (3)$$

The output of this operation is a smaller patch, having the correct width to cover only the geometry of the scene involved in the window violation, and giving as result in the rendered image a lateral black mask as in the post-processing floating window technique. Fig. 4 shows the left and right masks generated as 3D models in the scene, while Fig. 5 shows a rendering of the procedural city test scene before and after the automatic application of the floating window technique in Sverchok.

Conclusion

In this paper, we have investigated some possible uses of stereoscopic parameters as active parameters in the procedural

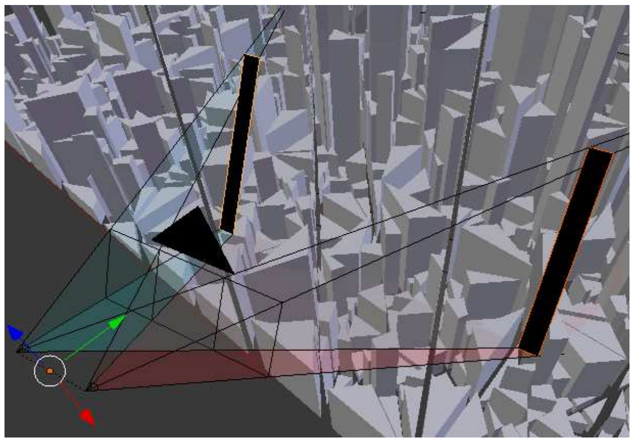


Figure 4. The resulting 3D lateral black masks placed in the negative retinal rivalry frustums, occluding the geometry causing the window violation. The patches size and positions in depth are calculated separately for the left and right frustums. No frustum culling has been applied in the picture.

generation of virtual scenes. From the preliminary results, the proposed methods seem effective and scalable enough to manage the automatic generation of CG scenes with different complexity, while keeping under control the final stereoscopic effect.

We have decided to try to integrate these methods inside a CG production software, in order to analyze also the current limits in the stereoscopic pipeline provided by the most diffused production softwares. It seems quite evident that even if several stereoscopic tools and visual aids have been included in CG softwares, there are still some limits in the access to all the parameters involved in the stereo setup, limiting in some aspects the possible use of stereoscopy as a creative tool since the preliminar stage of construction of a virtual scene. For example, many parameters, like the distance from the camera to the maximum parallax plane, are calculated but they are not visible to the user through the GUI, and they are also not included in the software development APIs: in the development of the techniques presented in this paper, we had to develop scripts to re-calculate some of these parameters, in order to use them for our procedural tools. To exploit all the potentialities of stereoscopy as a tool in the procedural modeling field, a better integration and extension of the stereoscopic pipeline in the CG production softwares is surely needed.

References

[1] B. Mendiburu, 3D Movie Making: Stereoscopic Digital Cinema from Script to Screen, Focal Press, 2009.

[2] B. Mendiburu, Y. Pupulin and S. Schklair, 3D TV and 3D Cinema: Tools and Processes for Creative Stereoscopy, Taylor and Francis, 2012.

[3] B. R. Gardner, Dynamic floating window: new creative tool for three-dimensional movies, J. Electronic Imaging, 21(1), 011009 (2012).

[4] J. Schild, Deep Gaming - The Creative and Technological Potential of Stereoscopic 3D Vision for Interactive Entertainment, CreateSpace Independent Publishing Platform, 2014.

[5] I. Bickerstaff, Case study: the Introduction of Stereoscopic Games on the Sony PlayStation 3, Stereoscopic Displays and Applications XXIII, Proceedings of SPIE 8288, 828815 (2012).

[6] J. Weaver and N. S. Holliman, Interlopers 3D: experiences designing

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.5.SDA-042

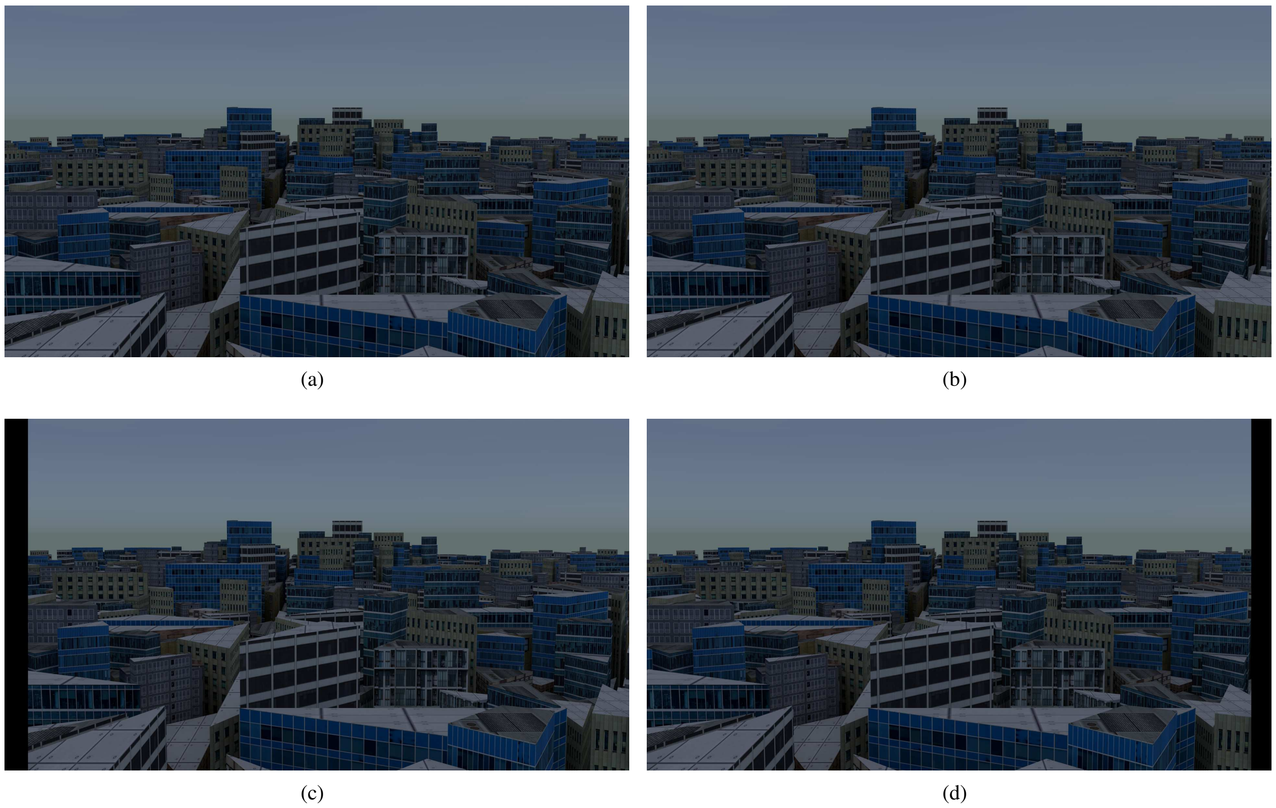


Figure 5. *The result of the automatic application of the floating window technique on a rendering (with $sff = 0.95$) of the virtual city procedurally created in Blender using the Sverchok add-on.*

a stereoscopic game, Stereoscopic Displays and Applications XXV, Proceedings of SPIE 9011, 90110F (2014).

[7] D. Marini, R. Folgieri, D. Gadia and A. Rizzi, Virtual Reality as a communication process, Virtual Reality, Springer, 16(3), 233-241 (2012).

[8] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin and S. Worley, Texturing & Modeling: a procedural approach, Third Edition, Morgan Kaufmann 2003.

[9] P. Merrell and D. Manocha, Model Synthesis: A General Procedural Modeling Algorithm, IEEE Transactions on Visualization and Computer Graphics, 17(6), 715-728 (2011).

[10] D. Ritchie, B. Mildenhall, N. D. Goodman and P. Hanrahan, Controlling Procedural Modeling Programs with Stochastically-Ordered Sequential Monte Carlo, ACM Transactions on Graphics, 34(4), 105:1-105:11 (2015).

[11] P. Prusinkiewicz and A. Lindenmayer, The Algorithmic Beauty of Plants, Springer-Verlag, electronic version, 2004.

[12] P. Müller, P. Wonka, S. Haegler, A. Ulmer and L. Van Gool, Procedural Modeling of Buildings, ACM Transactions on Graphics, 25(3), 614-623 (2006).

[13] G. Chen, G. Esch, P. Wonka, P. Müller and E. Zhang, Interactive Procedural Street Modeling, ACM Transactions on Graphics, 27(3), 103:1-103:10 (2008).

[14] M. Schwartz and P. Müller, Advanced Procedural Modeling of Architecture, ACM Transactions on Graphics, 34(4), 107:1-107:12 (2015).

[15] D. Maggiorini, M. Mannalá, M. Ornaghi and L. A. Ripamonti, FUN PLEdGE: a FUNny Platformers LEvels GEnerator, Proceedings of the 11th Biannual Conference on Italian SIGCHI Chapter, ACM, 138-145 (2015).

[16] City Engine, http://www.esri.com/software/cityengine (2015).

[17] Houdini, http://www.sidefx.com/ (2015).

[18] Blender, http://blender.org (2015).

[19] Sverchok Blender add-on, http://nikitron.cc.ua/sverchok_en.html (2015).

[20] L. Lipton, Foundations of the Stereoscopic Cinema, Van Nostrand Reinhold, 1982.

[21] J. F. Hughes, A. van Dam, M. McGuire, D. F. Sklar, J. D. Foley, S. K. Feiner and K. Akeley, Computer graphics: principles and practice, Third edition, Addison-Wesley, 2013.

DOI: 10.2352/ISSN.2470-1173.2016.5.SDA-042

Author Biography

Manlio Scalabrin has received his B.S. in Digital Communication from University of Milan in 2015. His research interests focus mainly on procedural modeling and animation techniques.

Laura Anna Ripamonti has graduated in Engineering and Managerial Sciences at Politecnico di Milano and then got a Ph.D. in Computer Science at University of Milan. Currently, she is Assistant Professor at the Department of Computer Science of the University of Milan, where she is co-directing the PONG - Playlab fOr inNovation in Games, and a master degree on Video Games for Computer Science. Her research interests focus on video games, with a special interest for MMORPGs (Massively Multiplayer Online Role-Playing Games) and Applied Games. In particular, she investigates the relations connecting game design issues to social interaction and AI applications.

Dario Maggiorini is Associate Professor at the University of Milan. He joined the Department of Computer Science in 2003 where he is now co-directing the PONG - Playlab fOr inNovation in Games. In the past, he has been working on the Quality of Service in IP networks, multimedia content delivery, application-level networking, and software architectures for service provisioning. Currently, his research interests focus mainly on software and network architecture to support entertainment applications and on content, service, and gaming provisioning in distributed environments.

Davide Gadia has received his M.Sc. (2003) and Ph.D. (2007) in Computer Science from the University of Milan. Currently he is assistant professor at Department of Computer Science at University of Milan. His research interests focus mainly Computer Graphics, Video Game programming and Virtual Reality. He is also interested in the processing of stereoscopic images and videos. He is involved in the research activities of the Pong Laboratory and of the Virtual Theater, an advanced VR visualization system installed at the University of Milan.