TESSELLATING OCEAN SURFACES

Anna Puig-Centelles, Francisco Ramos, Miguel Chover
Universitat Jaume I, Castellon, Spain

Oscar Ripolles
Universitat Politecnica de Valencia, Valencia, Spain

Mateu Sbert
Universitat de Girona, Girona, Spain

Keywords: Ocean simulation, Tessellation, GPU, Geometry Shader.

Abstract: Modeling and rendering realistic ocean scenes has been thoroughly investigated for many years. Its appearance has been studied and it is possible to find very detailed simulations with a high degree of realism. Nevertheless, real-time management of the huge heightmaps that are necessary for rendering ocean is still not solved. We propose a new technique for tessellating the ocean surface on GPU. This technique is capable of offering view-dependent approximations of the mesh while maintaining coherence among the extracted approximations when refining or coarsening the mesh. This feature is very important as most solutions previously presented must re-tessellate from the initial mesh.

1 INTRODUCTION

Simulating ocean surfaces is a very complicated challenge, as ocean is composed of different elements that form a very complex system. Most authors simulate an ocean as an unbounded water surface that is represented as a heightmap. In these simulations, the height of each vertex is modified in real time to offer the sensation of wave movement. Figure 1 depicts a snapshot of a mesh simulating ocean movement in a given instant of the animation. Other complex phenomena, such as foam, spray or splashes are usually modeled and rendered using particle systems (Peachey, 1986; Darles et al., 2007).

Managing the mesh representing the ocean still poses a limitation in simulating ocean. A technique that several authors propose is the tessellation of the surface. Moreover, it is possible to find adaptive solutions where some areas of the mesh are more detailed than the others, depending on the criteria used by the application. It is possible to find in the literature many solutions aimed at tessellating by exploiting GPU features, although not many have been applied to ocean surfaces.

This paper introduces a new adaptive tessellation scheme which works completely on GPU. Figure 1 presents an example of our algorithm where those areas closer to the camera are more refined. In addition, our proposed algorithm is capable of maintaining coherence among the extracted approximations as it can re-use the latest calculated approximation when increasing or decreasing the detail. Lastly, the simulation that we propose simulates wave movements with Perlin noise (Perlin, 1985) calculated on GPU.

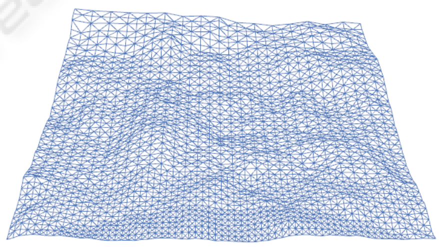


Figure 1: An ocean can be seen as an animated heightmap.

This paper has the following structure. Section 2 presents the state of the art on ocean simulation and tessellation. Section 3 thoroughly describes the tessellation technique that we present. Section 4 offers some results on the presented technique. Lastly, Sec-

Puig-Centelles A., Ramos F., Chover M., Ripolles O. and Sbert M. (2010).
TESSELLATING OCEAN SURFACES.
In *Proceedings of the International Conference on Computer Graphics Theory and Applications*, pages 319-322
DOI: 10.5220/0002827003190322

tion 5 includes some conclusions on the developed techniques and outlines our future work.

2 RELATED WORK

In this state of the art we will describe GPU tessellation techniques that have been developed for rendering ocean scenes. There have been several recent attempts to generate in real-time adaptive water surfaces on graphics hardware. Hinsinger et al. (Hinsinger et al., 2002) rely on an adaptive sampling of the ocean surface and the wave animation, dictated by the camera position. The tessellation and waveform superposition is performed on the CPU and uploaded to the GPU each frame, which is the bottleneck of their approach. Johanson (Johanson, 2004) presented the projected grid concept, which creates a grid whose vertices are even-spaced in post-perspective camera space. This representation provides spatial scalability and the possibility of developing a fully-GPU implementation is described. In paper (Yang et al., 2005), the authors offer GPU-based tessellation by using a previous adaptive scheme for terrain rendering. Their tessellation scheme uses a restricted quad-tree where two neighbouring areas with different resolutions can only vary to a limited extent. Lastly, in (Demers, 2005) Demers et al. presented a simulation which is adaptively tessellated in eye space, mapping a regular grid to the intersection of the ocean plane and the camera viewport.

3 OUR GPU-BASED TESSELLATION SCHEME

Many of the tessellation algorithms presented before modify the detail of the triangles following some criterion applied to the triangle. The calculations involved could consider the distance of the triangle to the camera or its position in the screen. Nevertheless, applying the level-of-detail criterion in a triangle basis implies a limitation for adaptive solutions, like the appearance of noticeable cracks and holes in the mesh. These cracks are due to the introduction of *T-vertices*, which appear commonly in tessellation algorithms when a vertex is positioned on the edge of another triangle.

In order to avoid crack problems, some authors have proposed to apply the refinement criterion only to the edges of the triangle. Therefore, if an edge needed refinement, then both triangles sharing the edge would act accordingly. Schmiade et al. described some edge-based patterns for tessellating triangles which completely avoided the appearance of T-vertices (Schmiade, 2008). Figure 2 presents, on the left side, an initial rectangular triangle where its hypotenuse and catheti (or *legs*) are depicted anticlockwise as H, C_1 and C_2. Next, the seven tessellation patterns introduced by Schmiade et al. are presented (labeled from 1 to 7), where the edges of the original triangle that need refinement are depicted in red. Each pattern shows the tessellation that would be necessary depending on the combination of edges that need refinement.

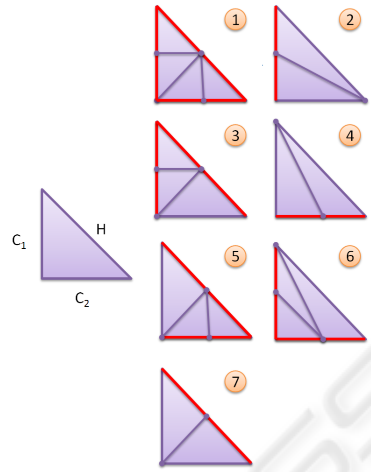


Figure 2: Tessellation patterns (Schmiade, 2008).

In our proposal we will use these patterns, as they can assure that no T-vertices are introduced and that the continuity of the mesh is maintained. It is worth mentioning that we will not store precomputed patterns on GPU memory as other solutions do (Boubekeur and Schlick, 2005). We just code in the *Geometry Shader* the seven cases that we follow so that the coordinates of the new vertices can be easily calculated from the coordinates of the two vertices that define the edge.

3.1 Our Proposed Algorithm

The main feature of the framework that we are presenting is the possibility of refining or coarsening the mesh while maintaining *coherence*. By coherence we refer to the re-use of information between changes in the level-of-detail. In such way, the latest extracted approximation is used in the next step, optimizing the tessellation process and improving the performance.

When *refining* the mesh, the algorithm checks each edge to see whether they need refinement. Depending on the combination of edges that need more detail, the algorithm selects a pattern (see Figure 2).

Each of these generated triangles stores the spatial coordinates, the texture information and any other information needed for rendering. In addition, it is necessary to output for each new triangle two pieces of information:

- A number indicating the tessellation *depth* for the triangle.
- A number coding the tessellation patterns that have been applied, keeping this information in $patternInfo$.

The need of storing these two values is due to the fact that we must know how a particular triangle was created in order to know how we should modify it when swapping to a lower level of detail. On the one hand, the $depth$ value will simply indicate how many tessellation steps have affected a particular triangle. On the other, the $patternInfo$ number is used to store all the patterns that have been applied to refine a triangle. This value is obtained following Equation 1. In this equation $appliedPattern$ refers to the type of the latest pattern, the one that we have used to create this triangle. The value $numberOfPatterns$ refers to the number of available patterns that we can apply in our tessellation algorithm. In our case we will use the 7 patterns presented in Figure 2 and, as a consequence, $numberOfPatterns$ will be equal to 7.

$$patternInfo = patternInfo + appliedPattern \cdot numberOfPatterns^{depth} \tag{1}$$

This $patternInfo$ value will not be the same for all the triangles belonging to the same parent. When tessellating a triangle, for example with the pattern used when the hypotenuse and both legs are refined (see pattern 1 in Figure 2), four triangles are output. Nevertheless, only one of these triangles will be needed when diminishing the detail. Three of them will be discarded and the other one will be modified to recreate the geometry of the parent triangle. Therefore, the $patternInfo$ value will be calculated with Equation 1 for the triangle that survives and will be 0 for the ones that will be eliminated. We will see the coarsening process in more detail in the following subsection.

3.1.1 Coarsening the Mesh

A different process should be applied when diminishing the detail of the mesh. The $depth$ and $patternInfo$ values presented above are the elements that enable our algorithm to recover less detailed approximations without having to start again from the coarsest approximation. It is important to underline that this is one of the main features of the method we are proposing.

The first step would be to find out whether the processing triangle can be discarded or if it is the triangle in charge of retrieving the geometry of the parent triangle. A positive $patternInfo$ value indicates that it is the latter case, the triangle will calculate the spatial coordinates of its parent triangle.

In order to retrieve the spatial coordinates of the parent triangle we must know which pattern was applied to create the existing triangle. This is due to the fact that for each pattern we will perform different calculations for retrieving the three vertices of the parent triangle. In this situation the $patternInfo$ value helps us to know which pattern was applied, as the latest pattern can be obtained with the next equation:

$$latestPattern = \frac{patternInfo}{numberOfPatterns^{depth-1}} \tag{2}$$

Once we know which pattern was applied, we calculate the position of the vertices and we output the new geometry with the new $patternInfo$ value obtained with Equation 3. The way we calculated this value assures that we will be able to continue coarsening the mesh or refining it without any problem.

$$patternInfo = patternInfo - latestPattern \cdot numberOfPatterns^{depth-1} \tag{3}$$

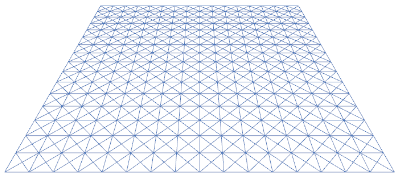
3.2 Camera Movement

We have described how the tessellation process works but we have considered that the location of the plane splitting the area where the mesh is and is not tessellated remains unaltered. Nevertheless, in a real case this line keeps moving all the time as it is usually related to the camera position.

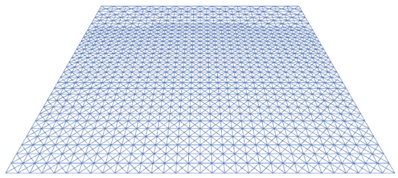
In these cases, a slightly different process is applied to correct the appropriate triangles. This algorithm checks each triangle to see whether, with the new criterion, their parent triangle would need a different tessellation. In that case, the algorithm will coarsen the triangles and refine them again. It is important to underline that both processes (coarsening and refining again) are executed at the same time and can be continuously executed while the camera is moving. In this particular case the $depth$ value is very important as it enable us to know how many tessellation coarsements and refinements must be executed in those areas that are affected by the camera movement.

4 RESULTS

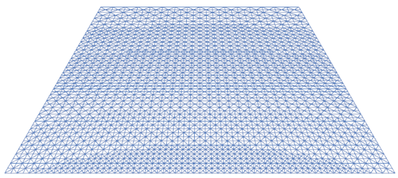
In this section we will study the performance of our tessellation method. Our scheme was programmed with GLSL and C++ on a Windows Vista Operating



(a) Second tessellation step (1,024 triangles).



(b) Third tessellation step (3,644 triangles).



(c) Fourth tessellation step (5,756 triangles).

Figure 3: Sample tessellation using a distance criterion.

System. The tests were carried out on a Pentium D 2.8 GHz. with 2 GB. RAM and an nVidia GeForce 8800 GT graphics card.

First, we offer some visual results of the tessellation algorithm. Figure 3 presents some snapshots of a tessellation case where an initial mesh composed of four triangles is refined according to the distance to the camera. In the more refined meshes it is possible to see how the tessellation is not uniform, as those areas of the mesh which are closer to the observer are more tessellated than those that are farther. These figures show how the tessellation process increases the detail of an input mesh without introducing cracks or other artifacts. Figure 1 shown at the beginning of this article presented the most detailed tessellation (Figure 3(c)) with Perlin noise applied to simulate waves.

In order to test the performance of our technique, Figure 4 presents the time needed for tessellating, animating and rendering the example depicted in Figure 3. This Figure shows how the noise calculations involve increasing the rendering time in 10% while the tessellation supposes an increase of 60%.

5 CONCLUSIONS

We have presented a method for simulating ocean in real-time. The presented approach is based on the use of a new adaptive tessellation scheme which exploits

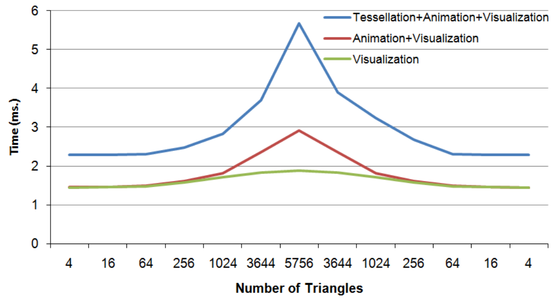


Figure 4: Performance obtained using a distance criterion.

coherence among extracted approximations. Accordingly, by storing some information, we are capable of reusing the latest extracted mesh when refining and coarsening the surface. For future work we are focused on the inclusion of more effects like refraction or the interaction of objects with the surface.

ACKNOWLEDGEMENTS

This work has been supported by the Spanish Ministry of Science and Technology (TIN2007-68066-C04-02, TIN2007-68066-C04-01) by Bancaja (P1 1B2007-56) and by ITEA2 (IP08009).

REFERENCES

Boubekeur, T. and Schlick, C. (2005). Generic mesh refinement on GPU. In *Proc. of the EUROGRAPHICS conference on Graphics hardware*, pages 99–104.

Darles, E., Crespin, B., and Ghazanfarpour, D. (2007). Accelerating and enhancing rendering of realistic ocean scenes. In *Proc. of WSCG 2007*, pages 287–294.

Demers, J. (2005). The making of clear sailing, secrets of the NVIDIA demo team. http://www.nzone.com/object/nzone_clearsailing_makingof1.html.

Hinsinger, D., Neyret, F., and Cani, M. (2002). Interactive animation of ocean waves. In *Eurographics symposium on Computer animation*, pages 161–166.

Johanson, C. (2004). Real time water rendering-introducing the projected grid concept. Technical report, Master of Science Thesis, Lund University.

Peachey, D. R. (1986). Modeling wave and surf. *Computer Graphics*, 20(4):75–83.

Perlin, K. (1985). An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296.

Schmiade, T. (2008). Adaptive GPU-based terrain rendering. Master's thesis, University of Siegen.

Yang, X., Pi, X., Zeng, L., and Li, S. (2005). Gpu-based real-time simulation and rendering of unbounded ocean surface. In *Int. Conf. on Computer Aided Design and Computer Graphics*, pages 428–433.