Combining displacement mapping methods on the GPU for real-time terrain visualization

Article *in* The Journal of Supercomputing · September 2016

DOI: 10.1007/s11227-016-1869-6

3 authors, including:

Mariano Pérez
University of Valencia
23 PUBLICATIONS **97** CITATIONS
SEE PROFILE

Juan M. Orduña
University of Valencia
127 PUBLICATIONS **743** CITATIONS
SEE PROFILE

Noname manuscript No.
(will be inserted by the editor)

Combining Displacement Mapping Methods on the GPU for Real-Time Terrain Visualization

Cesar González · Mariano Pérez · Juan M. Orduña

Received: date / Accepted: date

Abstract Real-Time terrain visualization plays an important rule in multiple popular applications. In these applications, displacement mapping algorithms (both per-vertex and per-pixel methods) can be used to improve the accuracy and performance of terrain rendering. Per-vertex methods are usually implemented by means of hardware tessellation, and per-pixel techniques like parallax mapping apply changes at the pixel level using the fragment shader. However, parallax mapping has not still been used in real-time terrain visualization applications due to different reasons.

In this paper, we propose a comparison study of different combinations of per-vertex and per-pixel methods. The performance evaluation results reveal that any of the implemented schemes improve the performance of terrain rendering, with respect to the performance yielded by the exclusive use of hardware tessellation. These results validate the proposed schemes as efficient alternatives for real-time terrain visualization applications.

Keywords Terrain visualization · Real-Time rendering · GPU shaders

1 Introduction

Real-Time terrain visualization is a very active research field in the area of computer graphics, and it plays an important role in multiple applications like Geographic Information Systems (GIS) [9,10], computer games [13] or civil or military simulators [8,15,7]. Figure 1 shows an example of an image displayed by a GIS application.

This work has been supported by Spanish MINECO and EU FEDER funds under grant TIN2015-66972-C5-5-R.

Cesar González, Mariano Pérez · Juan M. Orduña at Departamento de Informática - Universidad de Valencia - SPAIN
Tel.: +349644489
E-mail: {mariano.perez,juan.orduna}@uv.es

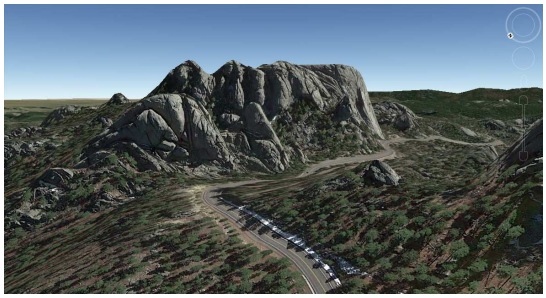

Fig. 1 Example image displayed by a Geographic Information System application

These applications should display a high visual quality terrain model at interactive frame rates. Terrain datasets used in these applications usually exceed the rendering capabilities of currently available hardware graphics, and their interactive rendering requires that applications adjust their geometry complexity in a view-dependent manner. Traditionally, most of the techniques used managed the level-of-detail required at every point of the terrain surface by executing CPU-based algorithms [11]. However, modern GPU features provide a more efficient way to carry out this task using GPU-based algorithms [12,6,5,2].

One way to reduce geometry complexity is implementing displacement mapping algorithms on the GPU [14], using per-vertex methods and/or per-pixel methods. Both methods use displacement maps (or height maps) to displace vertices of a tessellated mesh (per-vertex methods) or to displace texture coordinates of pixels (per-pixel methods) on a base mesh. Per-vertex displacement algorithms are closely related to tessellation and subdivision algorithms. They are used in real-time terrain rendering applications to generate a detailed geometry from a coarse mesh in a view-dependent manner. They tessellate a base mesh depending on the camera distance and moving the generated vertices along their normal according to the value stored in a height map. They are actually implemented on the GPU using hardware tessellation [17], which has become a de-facto standard nowadays due to its high fidelity interactive terrain rendering [1]. Per-pixel displacement algorithms, like parallax mapping [14], enhance the appearance of the rendered image by applying changes at the pixel level using the fragment shader, without increasing the number of polygons. They use information included in a displacement map with small height variations of the base surface to modify the texture coordinates of the color texture applied to the model. In an intuitive way, parallax mapping corrects the coordinates of the color texture mapped on the base mesh, approximating them to its coordinates as if the full resolution surface had been rendered as a polygonal mesh instead. However, these algorithms are rarely used in real-time terrain rendering. The main reason is that parallax mapping was originally designed for real surfaces with small variations on the base surface (such as stone walls or floors), but a real terrain surface can show great height variability, since terrains typically have a fractal structure. This issue can become a se-

rious problem, obtaining a wrong estimation, specially when the the virtual camera is placed near the ground surface.

In a previous work [3], we proposed a primary combination of hardware tessellation and a modified version of basic parallax mapping, making possible the use of the de-facto standard of hardware tessellation in real-time terrain rendering applications. In this paper, we propose a comparative study of three possible versions of parallax mapping [14] combined with hardware tessellation, in order to find the best strategy for real-time terrain rendering applications. The performance evaluation results show that the proposed schemes improves the performance of real-time terrain rendering applications in regard to the performance yielded when using hardware tessellation only.

The rest of the paper is organized as follows: section 2 describes the proposed approach for real-time terrain rendering. Next, section 3 shows the performance evaluation of the proposed models. Finally, section 4 shows some conclusion remarks.

2 Displacement mapping Implementations

We propose an approach combining per-vertex displacement mapping using hardware tessellation with per-pixel displacement mapping using parallax mapping techniques. The parallax mapping method described in [3] corresponds to a basic parallax mapping method with offset limiting [14]. In this section, we describe other possible versions of the parallax mapping method. In order to make this paper self-contained, we also describe the hardware tessellation procedure which takes place in shader model version 5.

2.1 Tessellation

Hardware tessellation requires that the base mesh of the tile is defined as a set of patch primitives. The tessellation technique uses height maps (textures with elevation information), in addition to color texture, in order to add geometry from a coarse mesh according to the distance to the virtual camera, performing a smooth view-dependent level-of-detail representation. Following the OpenGL nomenclature, the Tessellation Control Shader (TCS) should determine how many times the patch must be subdivided on the basis of the camera distance. It is also responsible for ensuring continuity across boundaries patches, forcing the shared edges between the patches to use the same level of tessellation. Figure 2 shows part of our code (using GLSL language) for this shader. The control points and texture coordinates are passed unaltered to the next stage.

The Tessellation Primitive Generator (TPG) subdivides the patch based on the tessellation level values computed by the TCS, and the Tessellation Evaluation Shader (TES) computes the vertex values for each generated vertex. The vertices position are updated by vertically moving the generated vertices according to the values stored in the height map. It must be noted that the displacement must be done in the normal direction to the reference surface where

```
if (frustrumTest == true) {
  gl_TessLevelOuter[0] = getTessLevel(distEdge0toCamera);
  gl_TessLevelOuter[1] = getTessLevel(distEdge1toCamera);
  gl_TessLevelOuter[2] = getTessLevel(distEdge2toCamera);
  gl_TessLevelOuter[3] = getTessLevel(distEdge3toCamera);
  gl_TessLevelInner[0] = getTessLevel(distCentertoCamera);
  gl_TessLevelInner[1] = getTessLevel(distCentertoCamera);
}
tcTexCoords[gl_InvocationID] = vTexCoords[gl_InvocationID];
gl_out[gl_InvocationID].gl_Position =
   gl_in[gl_InvocationID].gl_Position;
```

Fig. 2 Tessellation Control Shader code

the texture values are coded. However, the earth surface is locally planar, and the geographical information (composed of color textures and height maps) is provided as an orthographic projection of the raw data gathered by the measurement devices; therefore, the reference surface is an horizontal plane (a flat surface) and the normal vector is a vertical vector at every point. Figure 3 shows part of the code for this shader in our implementation.

```
vertexPos = computeVertexPosition(gl_TessCoord);
vec4 heightNormal = texture(uHeightNormalmap,tcTexCoords);
vertexPos.z = uHeightScale * heightNormal.r;
teEyeVector = normalize(uEyeWorldPos-vec3(uModelMatrix*vertexPos));
teNormal = normalize(heightNormal.gba * 2f - 1f);
teHBase = heightNormal.r;
teTexCoords = tcTexCoords;
gl_Position = uModelViewProjectionMatrix * vertexPos;
```

Fig. 3 Tessellation Evaluation Shader code

In this implementation we combine the height map and the normal map in only one texture, as shown in the above code (second line). The shader outputs the camera position and the normal vector in the worlds coordinate system, as well as the height of the base mesh. Our implementations of the parallax mapping algorithms will use this information in the fragment shader code, as explained in the next subsections.

2.2 Parallax mapping using a tessellated mesh as base surface

Simple parallax mapping [4] obtains the target texture coordinates uv_1 by the projection on the base surface of the intersection point of the real surface (high resolution surface) with the eye vector, assuming that the height field $h(uv)$ is near constant everywhere in the neighborhood of uv_0. Figure 4 illustrates this process. As described in subsection 2.1, the base surface is usually flat in the

case of terrain rendering. Therefore, the intersection point can be calculated simply from the direction of eye vector **v** and the height value $h(uv_0)$. Figure 5 shows the implementation for this flat base surface.

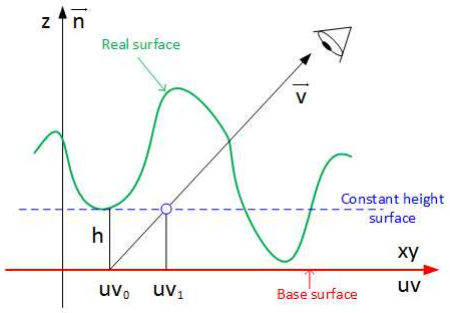

Fig. 4 Finding the intersection point between the eye and the base surface (flat mesh)

```
float h = texture(uHeightNormalmap, teTexCoords).r;
vec3 v = normalize(teEyeVector);
newTexCoords = teTexCoords + h * v.xy / v.y;
```

Fig. 5 Implementation of the parallax mapping method for a flat base surface

However, parallax mapping is not well-suited for terrain rendering when a flat base surface is used. The great variability of terrain data results in a wrong estimation of the corrected texture coordinates, specially when the camera is near to the terrain surface and the distance to this surface is similar or smaller to the height values stored in the height map. In that case, the new texture coordinates obtained (uv_1) can be far from the original texture coordinates (uv_0) or even they may be infinite, so the assumption of the height field being near constant is not satisfied in practice.

Nevertheless, this problem can be avoided using a base surface closer to the real surface, greatly reducing the displacement distance h, and in turn the difference between uv_1 and uv_0 values. Therefore, we use the mesh generated in the tessellation stages of the graphic pipeline, which is a lower resolution model of the terrain model than the real surface (full resolution model), as the base surface in the parallax mapping algorithm. Figure 6 illustrates the process when using this new base surface. This Figure shows how the new texture coordinates uv_1 are now obtained from the intersection of the eye vector and a parallel plane to the base surface shifted to $h(uv_0)$. The intersection point is calculated from the eye vector **v**, the normal vector of the base mesh **n** and the difference between the height map value $h(uv_0)$ and the base mesh height $H(uv_0)$.

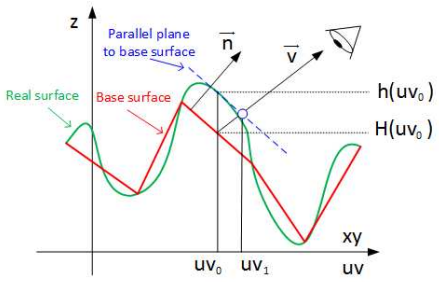

Fig. 6 Finding the intersection point between the eye and the base surface (tessellated mesh)

Figure 7 shows the parallax algorithm implementation for this new base surface. It must be noted that the height map is defined in a planar domain, but we now need to compute height variations of the real surface from every planar triangle of the new base surface in its normal direction. This is the reason why the tessellation evaluation shader must output the height H and the normal vector direction n at every vertex of the base surface, as it can be shown in figure 3.

```
float h = texture(uHeightNormalmap, teTexCoords).r;
vec3 v = normalize(teEyeVector);
vec3 n = normalize(teNormal);
newTexCoords = teTexCoords+(h-teHBase)*n.z*v.xy/max(dot(v,n),0.5);
```

Fig. 7 Implementation of the parallax mapping method for a flat base surface

This process does not change the geometry of the surface rendered, which still remains the mesh created by the tessellation shaders, but the results obtained are quite similar to the full resolution surface rendered as a polygonal surface instead. However, this algorithm has a significant flaw. As the eye vector becomes parallel to any triangle of the base mesh, the new texture coordinates become very large. This problem is reduced by limiting the offset so that it never becomes larger than a certain threshold value. This is why the implementation shown in Figure 7 includes the offset limiting with threshold 0.5 (last line). However, if the Welsh approximation [16] is used and a threshold of value 1 is adopted, then the division by $dot(v, n)$ is eliminated and the resulting code, shown in Figure 8, is simplified.

```
newTexCoords = teTexCoords + (h - teHBase) * n.z * v.xy;
```

Fig. 8 Implementation of the parallax mapping with offset limited to 1

2.3 Iterative Parallax mapping

Parallax mapping tries to offset the texture coordinates towards the corresponding full resolution surface point. It is unlikely that this target point is reached in a single attempt. The accuracy of the solution, however, can be improved repeating the process a few times [14]. The implementation of this version of the parallax mapping algorithm using a tessellated base mesh is shown in figure 9.

```
vec3 v = normalize(teEyeVector);
vec3 n = normalize(teNormal);
vec2 uv = teTexCoords;
for ( int i = 0; i < PAR_ITER; i ++ ) {
   float h = texture(uHeightNormalmap, uv).r;
   uv += (h - teHBase) * n.z * v.xy;
}
newTexCoords = uv;
```

Fig. 9 Implementation of the iterative parallax mapping method

It must be noted that the eye vector $\mathbf{v}$, the normal vector $\mathbf{n}$ and the height of the base surface H are not updated at every iteration, since the pixel position is always the same and it always belongs to the same fragment of the base mesh.

2.4 Binary search

The binary search method [14] consists of performing a binary search to find the real intersection point between the height field and the line in the direction of the view ray. The search starts assuming two initial endpoints, one below and the other one above the height field. The search halves the interval between the two points in each iteration step, putting the next point at the middle of the current interval, and keeping the half interval where one endpoint is above and the other one below the height field. The original binary search algorithm takes as initial points the minimum and the maximum possible values stored in the height map. However, these endpoints are usually far from each other, and therefore it is necessary to iterate many times for obtaining good results and avoiding aliasing problems. This issue obviously reduces the algorithm performance. However, we use the mesh generated by the tessellation shaders as base surface, as described above. This surface is close to the real surface and their vertices are points belonging to the real surface. Thus, the fragments inside the triangles of the base mesh must have texture coordinates limited by their vertices texture coordinates. Therefore, our implementation of the binary search method starts at the intersection point of the view ray with this base surface, and we displace endpoints left and right a proportional distance

to the size of the mesh triangles *tsize* in the direction of the view ray. These two endpoints are close to the real surface, and this implementation yields better results than the original binary search method. Figure 10 shows our implementation of the binary search method, in which the last step applies the secant method [14] to reduce the aliasing effect.

```
vec3 v = normalize(teEyeVector);
vec3 UVH1 = vec3(teTexCoords, teHBase) + tsize * v;
vec3 UVH2 = vec3(teTexCoords, teHBase) - tsize * v;
for ( int i = 0; i < PAR_ITER; i ++ ) {
  vec3 UVHm = (STH1 + STH2) * 0.5;
  float h = texture(uHeightNormalmap, UVHm).r;
  if (h < UVHm.z) { d1 = h - UVHm.z; UVH1 = UVHm; }
  else            { d2 = h - UVHm.z; UVH2 = UVHm; }
}
newTexCoords = vec2(UVH2 + (UVH1 - UVH2) * d2 / (d2 - d1));
```

Fig. 10 Implementation of the binary search method

3 Performance Evaluation

For evaluation purposes, we have implemented a generic terrain visualization application which performs a real-time fly over the terrain model using different visual quality parameters, and it allows taking and storing screen captures of the rendered terrain from the user point-of-view. The terrain database corresponds to an area located at the mountain range of Spanish Pyrenees. This dataset has been split in 3x5 regular tiles, each one of them being a heightmap image of 1024x1024 pixels with a sample spacing resolution of 5 meters. This involves a total surface of 15360 meters wide by 25600 meters long. The color textures have a resolution of 2048x2048 pixels. The total size of the test data is about 400 MB in its decompressed form. Tests were run on a PC-platform based on an Intel Core i7-4790 CPU processor and 12 GB of RAM, and we have performed the tests with three different NVidia graphic cards: GeForce GTX 590, GeForce GTX 650 and GeForce GTX 970.

We have analyzed five different techniques: only tessellation, tessellation + parallax mapping with offset limited to 0.5, tessellation + parallax mapping with offset limited to 1 (Welsh approximation), tessellation + iterative parallax mapping (with 2 iterations) and tessellation + binary search parallax mapping (with 3 iterations). The considered methods have been tested using different levels of detail (tessellation factors), in order to study how they scale with the number of triangles. The terrain data set has 3x5 tiles and the underlying geometry of every tile has been set to 16 subdivisions, which translates into a mesh of 7681 triangles with the minimum tessellation factor (1). The number of triangles linearly scales as the tessellation factor increases, until it reaches a total of 31457280 triangles at the maximum tessellation factor (64).

We have obtained the Mean Square Error (MSE) produced by the five considered techniques as a function of the number of triangles drawn in the scene, which in turn depends on the tessellation level, that is, the level of detail used by the tessellation algorithm for generating the mesh. Figure 11 a) show these results. This Figure shows that any of the proposed techniques yield a higher accuracy (a lower MSE) than tessellation when exclusively applied, particularly for low levels of detail.

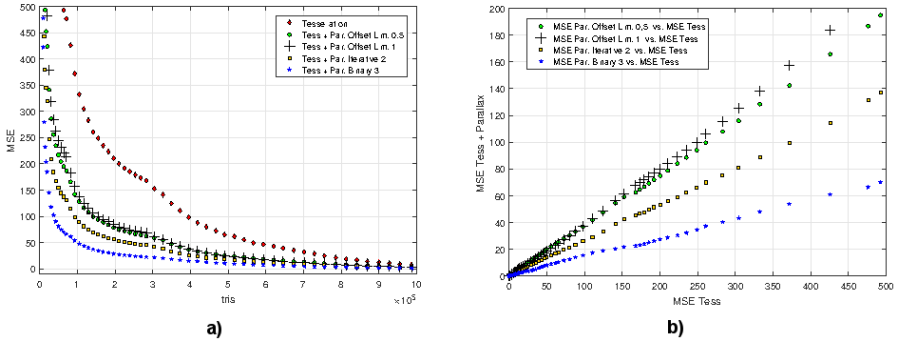

Fig. 11 a) MSE as a function of the number of triangles in the mesh b) Proportion between the MSE yielded by each technique.

Figure 11 b) shows a quantitative measurement of the improvements achieved by the proposed techniques, in regard to the performance obtained by tessellation. In this figure, each point represents a fixed number of triangles. For that number of triangles, the X-axis value is the MSE yielded by the tessellation technique, and the value in the Y-axis is the MSE yielded by every proposed technique (tessellation + parallax mapping with offset limiting, tessellation + iterative parallax mapping and tessellation + binary search parallax mapping). The inverse of the slope of every plot is around 2,4 in the case of parallax with offset limited to 1, around 2.5 in the case of parallax with offset limited to 1, around 3.6 in the case of iterative parallax mapping, and around 7.1 in the case of binary search parallax mapping. These results mean that the proposed techniques yield an improvement of the qualitative results of around 140%, 150%, 250% and 610% better (in terms of MSE), respectively, with respect to the ones yielded by tessellation. That is, the Binary Search method (combined with tessellation) provides the best results, followed by the Iterative method and the Offset method. The threshold limit used in the latter one does not add significant differences in the performance of that method.

We have also measured the frame rates (the inverse of the time required for rendering each frame) obtained with the three considered graphic cards. However, due to space limitations we only show here the results for the NVidia GTX 970 card. The results for the other two cards were (proportionally) very similar, showing the robustness of the evaluation. It must be noted that the

frame rate value shown by this metric is not the real frame rate as seen by the user, since the application code running in the CPU is the actual bottleneck. If we had considered the real frame rate, we would have not been able to appreciate significant differences between the five considered methods, specially when the number of triangles being drawn is relatively small. Additionally, the results would have been dependent on the application.

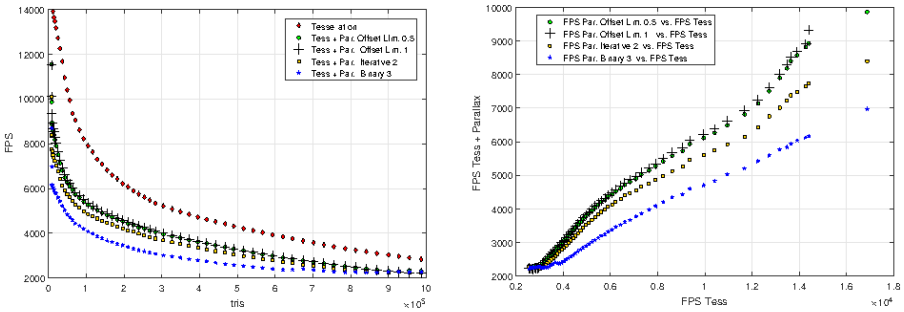

Fig. 12 Results using NVidia GTX 970 GPU: a) FPS yielded by each method b) Comparative frame rates

Figure 12 a) shows the frame rates yielded by each of the considered method. This Figure shows that the binary search method yields the lowest frame rate, followed again by the iterative method and the offset method. Figure 12 b) shows a comparative measurement of the frame rate decreasing in every proposed technique, in regard to the performance obtained by tessellation. Analyzing the numeric values in detail, we have seen that in the case of parallax mapping with offset limiting (regardless the limit is set to 0.5 or 1) the frame rate decreases around 80% when the number of triangles is small (high FPS) and around 25% when it is large (low FPS), in regard to using only tessellation. In the case of iterative parallax mapping, the frame rate decreases around 125% when the number of triangles is small and around 40% when it is large. Finally, for the case of binary search parallax mapping the FPS decreases around 200% for a small number of triangles, and around a 50% for a large one. If we compare these results with the ones shown in Figure 11 a), we can see a great improvement in the qualitative results (MSE) obtained by each of the implemented methods and a limited worsening of the frame rate yielded by the GPU. Since the displayed frame rate is usually limited by the CPU time, the practical effects of the worsening are negligible. Comparing the results for the considered parallax mapping methods among them, and taking into account that current graphics cards can render a large number of triangles, we can conclude that the binary search method is currently (and specially in the future) the most recommended method to be used in real-time terrain rendering.

Finally, Figure 13 shows the visual effects of the proposed methods. The image on the downright corner shows a screenshot of the terrain visualization application at its maximum resolution, using around 130000 triangles. The image on the upleft corner shows the same image at a much lower resolution, using around 2000 triangles (a factor of reduction of 64x) when exclusively tessellation is used. The image on the upright corner shows an image at the same low resolution when using parallax mapping with offset limiting. Finally the image on the downleft corner shows the image at the same low resolution when using the binary search method with 3 iterations. These images show how parallax mapping (particularly the binary search method) improve the visual results in regard to the exclusive use of tessellation, yielding visual results very similar to the full resolution mesh (see for example the road in the images).

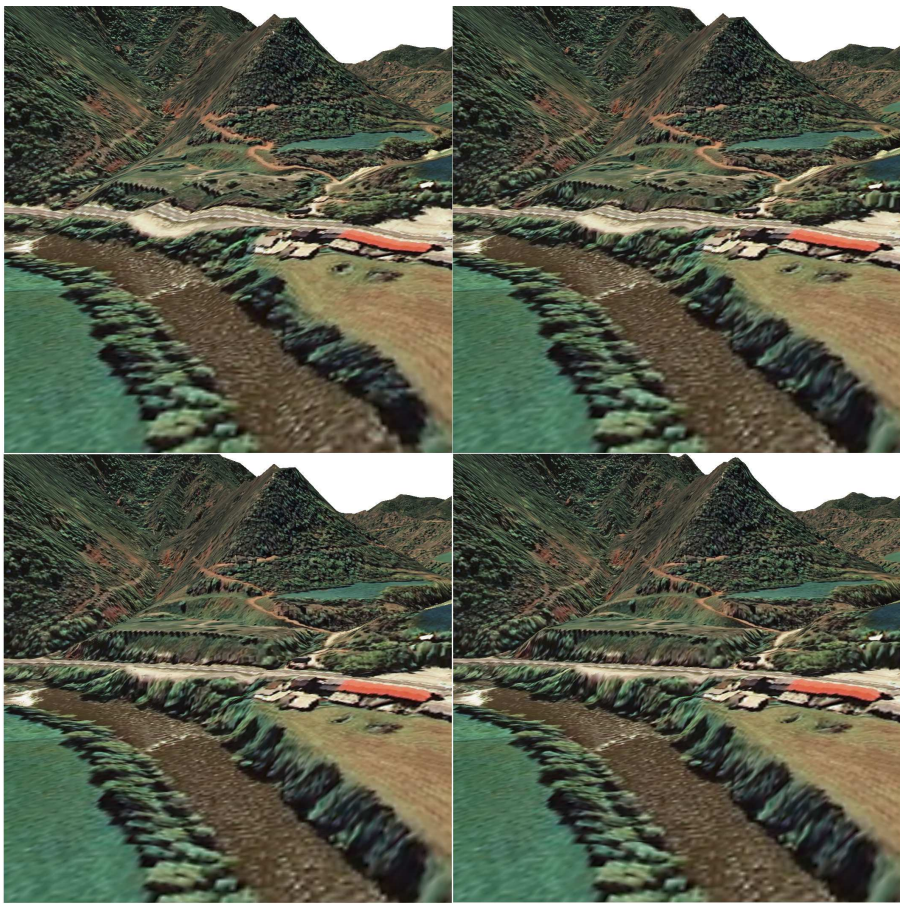

Fig. 13 Visual effects of the proposed methods

4 Conclusions

In this paper, we have proposed a comparative study of different real-time terrain rendering methods which efficiently combines hardware tessellation and parallax mapping, making parallax mapping compatible with hardware tessellation and terrain rendering. The performance evaluation results show that any of the parallax mapping method implemented are well-suited to terrain rendering applications, improving the visual results at the same frame rate with respect to the exclusive use of tessellation. Nevertheless, the binary search method seems the most appropriate method for current graphic cards, since it yields the best relationship between visual quality and frame rate.

References

1. Cantlay, I.: Directx 11 terrain tessellation. Nvidia whitepaper, 8 (2011)
2. Dick, C., Krüger, J., Westermann, R.: GPU Ray-Casting for Scalable Terrain Rendering. In: Proceedings of Eurographics 2009 (2009)
3. González, C., Pérez, M., Orduña, J.M.: A hybrid gpu technique for real-time terrain visualization. In: Proceedings of Computational and Mathematical Methods in Science and Engineering (2016)
4. Kaneko, T., Takahei, T., Inami, M., Kawakami, N., Yanagida, Y., Maeda, T., Tachi, S.: Detailed Shape Representation with Parallax Mapping. In: Proceedings of ICAT 2001, pp. 205–208 (2001)
5. Livny, Y., Kogan, Z., El-Sana, J.: Seamless patches for gpu-based terrain rendering. The Visual Computer **25**(3), 197–208 (2009). DOI 10.1007/s00371-008-0214-3
6. Livny, Y., Sokolovsky, N., Grinshpoun, T., El-Sana, J.: A gpu persistent grid mapping for terrain rendering. The Visual Computer **24**(2), 139–153 (2008)
7. Microsoft, I.: Flight simulator home page. Retrieved April 27, 2016 from http://www.microsoft.com/games/fsinsider (2016)
8. Okuyan, E., Güdükbay, U.: Direct volume rendering of unstructured tetrahedral meshes using cuda and openmp. The Journal of Supercomputing **67**(2), 324–344 (2014). DOI 10.1007/s11227-013-1004-x
9. Olanda, R., Pérez, M., Orduña, J.M., Rueda, S.: Terrain data compression using wavelet-tiled pyramids for online 3d terrain visualization. International Journal of Geographical Information Science **28**(2), 407–425 (2014). DOI 10.1080/13658816.2013.829920
10. Olanda, R., Pérez, M., Orduña, J.M., Rueda, S.: Improving hybrid distributed architectures for interactive terrain visualization. The Journal of Supercomputing pp. 1–12 (2015). DOI 10.1007/s11227-015-1593-7
11. Pajarola, R., Gobbetti, E.: Survey of semi-regular multiresolution models for interactive terrain rendering. The Visual Computer **23**(8), 583–605 (2007)
12. Schneider, J., Westermann, R.: Gpu-friendly high-quality terrain rendering. Journal of WSCG **14**(1-3), 49–56 (2006)
13. Square_Enix, C.: Final fantasy XIV home page. Retrieved May 1, 2015 from http://www.finalfantasyxiv.com/ (2015)
14. Szirmay-Kalos, L., Umenhoffer, T.: Displacement mapping on the GPU - state of the art. Comput. Graph. Forum **27**(6), 1567–1592 (2008)
15. Tran, V.T., Lee, J., Kim, D., Jeong, Y.S.: Easy-to-use virtual brick manipulation techniques using hand gestures. The Journal of Supercomputing pp. 1–15 (2015). DOI 10.1007/s11227-015-1588-4
16. Welsh, T.: Parallax mapping with offset limiting: A perpixel approximation of uneven surfaces. Retrieved April 22, 2016 from http://pds1.egloos.com/pds/1/200603/10/62/parallax_mapping.pdf, Jan 2004. (2004)
17. Zink, J., Pettineo, M., Hoxley, J.: Practical rendering and computation with Direct3D 11. CRC Press, Boca Raton (2011). URL http://opac.inria.fr/record=b1134711. An A.K. Peters book.