Real-Time Craters Generation On Dynamic Terrains

Alfredo Cossetin Neto[1], **Flávio Franzin**[1],**Cesar Tadeu Pozzer**[1], **Natan Berwaldt**[1], **Gabriel Di Domenico**[1], **Gustavo De Freitas**[1]

[1]Centro De Tecnologia – Universidade Federal De Santa Maria (UFSM)
Santa Maria – RS – Brazil

{acneto,ffranzin,pozzer,gddomenico,gmfreitas,nlberwaldt}@inf.ufsm.br

Abstract. *This paper presents an innovative technique for simulating crater deformations caused by explosions in large height map-based vir- tual terrains. Unlike traditional methods, our proposed approach does not directly deform the landscape. Instead, we discretize the crater information into simple variables and store it in a compact GPU-based hash table. The actual deformation is then calculated later using compute shaders each time a block of the height map is loaded to the GPU. This approach allows for the deformation of terrain at very far distances without the need to load large heigh map textures, thereby saving memory, as demonstrated by our comparison with another method. Moreover, this paper explores techniques to enhance the visual quality of craters by incorporating noise algorithms and appropriate coloring, thus increasing the overall realism of the areas affected by explosions.*
Keywords *height map, hash table, virtual terrain, real-time deformation, explosion, crater*

1. Introduction

Action, war, and shooting games are known for their vast and intricate terrains, which often serve as the backdrop for high-stakes battles and explosive combat. In many of these games, the presence of missiles, grenades, and other explosive elements necessitates the implementation of terrain deformation methods that dynamically respond to these events. However, most of the implemented methods are proprietary and lack the ability to reproduce the results in other game developments. Therefore, the capability to create a real-time deformable terrain that reacts to what occurs during gameplay is crucial to increasing the level of immersion and overall gaming experience.

Dynamic terrains have been widely utilized in video games to optimize performance since they allow changing the number of vertices of the scenario in real-time according to the need for detail. Among the various algorithms used in the creation of dynamic terrains, Geometric Clipmap [Losasso e Hoppe 2004] and Geometric Mipmap [de Boer 2000] stand out. These two methods represent the terrain as a height map, where each pixel value represents a vertex height. Thus, any flat mesh with any level of resolution can be loaded into the GPU, and its configuration changed to the desired shape in real-time through the use of the height map.

Numerous techniques for terrain deformation rely on altering height maps in real-time to simulate changes in the ground caused by various factors. For example, [Crause et al. 2011] utilized a combination of simple and detailed height maps that could be dynamically updated in GPU to create a variety of terrain deformations. Specifically, the coarse-maps were implemented using Geometric Clipmap to generate less precise deformations, while the detailed-maps were used with tessellation techniques to produce more intricate and complex changes in the terrain. However, it is important to note that their approach uses large textures, which can be very memory-consuming. Applying this technique to large areas across an entire terrain could lead to challenges related to memory usage and loading speed, as many blocks of height map texture would need to be used at the same time in case of multiple deformations happening simultaneously or at great distances.

In this paper, a new approach for simulating real-time terrain deformations caused by explosions is presented. Unlike previous methods [Crause et al. 2011] [Aquilio et al. 2006], this approach stores the deformation information not in height maps but in an external and more compact GPU-based hash table structure, allowing many deformations across large terrains at a minimum memory cost. This structure can be used to quickly shape the explosion craters in the terrain's vertices or color it in the fragment shader whenever an area affected by explosions is loaded. This capability enables the game to simulate events that occur at great distances from the player's current location while still being able to interact with the ground.

2. Related Work

With the advancement of current GPUs, modern games strive to create an interactive world where almost all kinds of explosives can interact with the terrain. Proprietary methods are implemented in these games, so any assertion about the techniques used is mere speculation. However, they are likely based on tessellation and parallax methods.

[Crause et al. 2011] employed a combination of coarse and detailed height maps to represent the terrain, which could be quickly redrawn in the GPU whenever a deformation is required to be displayed. Specifically, the coarse maps were implemented using Geometric Clipmap [Losasso e Hoppe 2004] to generate simpler deformations, while the detailed maps used tessellation techniques to produce smaller changes in the terrain. Due to high memory consumption caused by the texture used, a caching system was implemented to keep only the most likely-to-be-used maps in memory. However, this approach restricted deformations to areas close to the player's position, limiting how far away deformations could occur.

[Aquilio et al. 2006] developed a technique capable of calculating and displacing terrain deformations using height maps entirely on the GPU, providing highly efficient deformation handling. However, their method is limited to the areas of the terrain that are currently loaded and can be immediately rendered by the camera limiting the use of level of detail techniques.

In modern terrain deformation techniques, representing the world as a simplified 2D surface provides improved performance and at the same time restricts the ability to simulate 3D deformations, such as overlapping areas accurately. However, in simulation scenarios where accuracy takes precedence over performance, alternative methods are preferred. For instance, [Wan e Tang 2002] utilized a CPU-based algorithm to simulate physically-based explosion deformations on the terrain. Their approach involved employing the ROAM [Duchaineau et al. 1997] algorithm to dynamically reconstruct the mesh of the whole terrain every time a new explosion happened, allowing any kind of possible deformation.

[Shao et al. 2020] devised a physical modeling approach to simulate vertical deformations resulting from explosions on virtual terrains. They employed mathematical geometries to approximate the shape of craters, considering the properties of the soils and the explosive forces that induce the deformations. Our method uses an adapted version of this technique to compute deformations in the height map due to its simplicity and efficiency.

3. Proposed Method

Our proposed method is a novel approach to handling craters caused by explosions in dynamic terrain-based applications that use height maps. The method involves storing simple information about each explosion that occurs on the terrain using a hash table data structure. A compute shader can access this information in real-time and use it to calculate the appropriated deformation(if there be any) and store it in the terrain's height map as it is being loaded. For the blocks that are already loaded, the computer shader is also called when an update occurs in the hash table.

In order to carry out the appropriate coloring of a crater, the fragment shader also utilizes the hash table. However, unlike the vertex displacement stored in height maps, color calculations would require a very high amount of memory to be stored in texture, which is not reasonable. As a result, during each frame, the fragment shader needs to iterate over the hash table to determine the color of a crater, which can be quite expensive as shown in Section 4.

Our implementation does not rely on tessellation shaders, which limits how detailed the deformation can be due to the terrain's resolution. Without tessellation, there is a limit below which craters become too small and compromise visual quality. The smallest acceptable size depends on the desired quality and the terrain used. An example of this effect is shown in Figure 1.

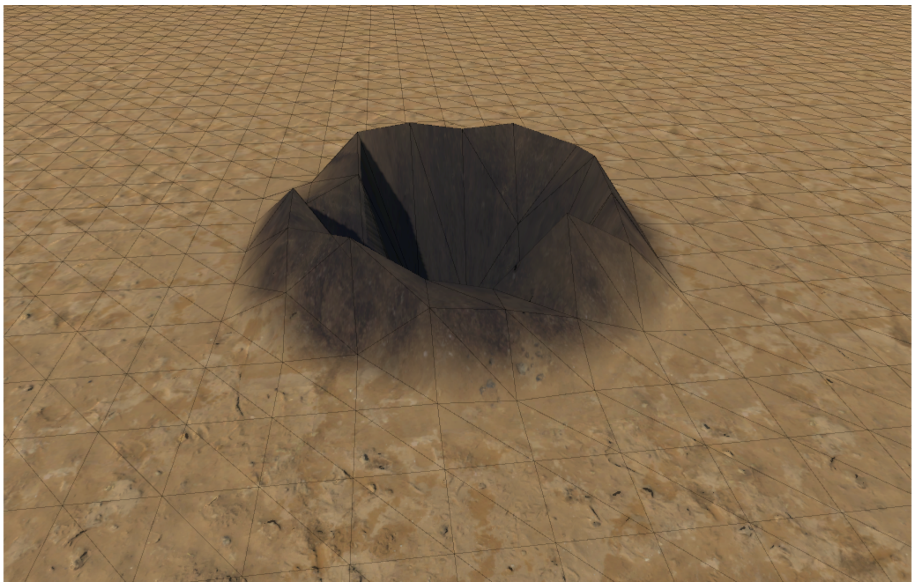
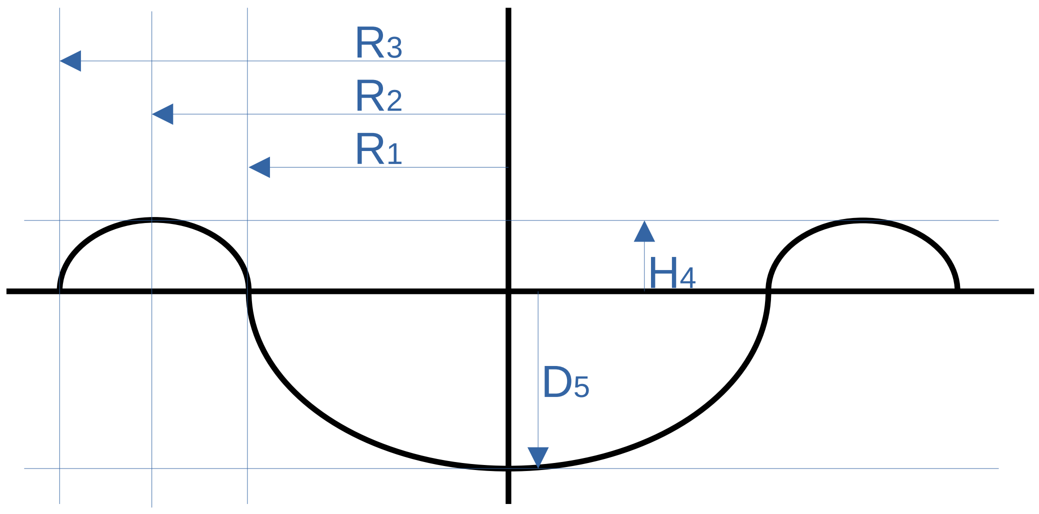
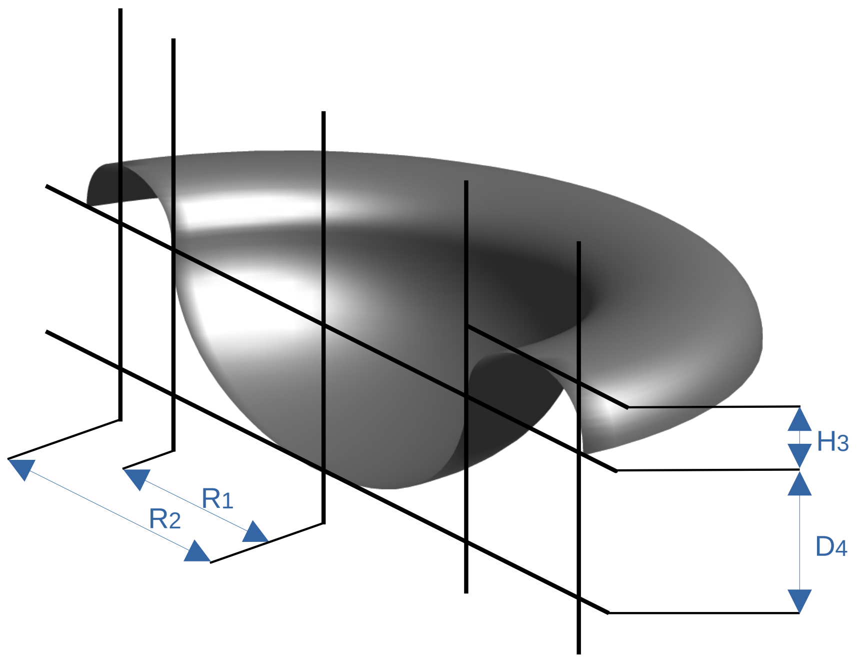


Figure 1. A very small crater in very low terrain resolution

3.1. Calculating Deformation



(a) Crater profile



(b) Half crater in perspective

Figure 2. Base geometry of a crate
R1 - Ellipsoid and Toroid Radius; R2 - Toroid Major Radius; R3 - Crater Radius; H4 - Toroid Height; H5 - Ellipsoid Depth(height)

The fundamental principle underlying our method is to dynamically recalculate the terrain's deformation on the GPU whenever necessary by computing the appropriated displacement of each vertex of the terrain inside the range of a crater. For that, we employ a straightforward geometric definition proposed by [Shao et al. 2020] that can be expressed using simple surface formulas. According to this definition, the shape of a crater can be divided into three distinct regions: the cavity, the internal deformation zone, and the external deformation zone. The cavity corresponds to a half-ellipsoid, while the external and internal deformation zones are described as a half-ellipse that undergoes rotation around the y-axis, resulting in an externally enclosed surface equal to a half-toroid. Figure 2 provides a visual representation of this geometry.

Referring to Figure 2 as a visual guide, we can provide a simplified description for each crater shape. The ellipsoid can be approximated as perfectly circular, allowing us to define it using just two parameters, namely R1 (the radius) and D5 (the depth). On the other hand, the toroid shape can be described using R1 (minor radius which is the same as the ellipsoid's radius), along with R2 (the major radius of the toroid) and H4 (the height of the toroid). R3 can be considered equal to R2 plus its difference from R1, so it is not necessary to be stored and can be discarded. In summary, apart from its world position, the crater can be characterized using only four variables: R1, R2, H4, and D5.

Given a crater specified by these four variables, we need a way to calculate its elevation at any arbitrary point inside its range. While it is possible to use the ellipsoid and toroid surface formulas for this calculation, it is not ideal when dealing with height map terrains. As discussed in Section 2, height maps cannot accurately represent overlapping areas, which is caused by these formulas at the boundaries of the ellipsoid and toroid surfaces. To address this issue, we approximated the cavity and internal deformation zone using a smooth step function that received the distance from the center of the crater as input. For the external deformation zone, we used the circle formula followed by another smooth step function to refine the final output. These calculations can be seen in clear steps in Algorithm 1.

Finally, to get the final displacement of any vertex, we need to consider if multiple craters are intersecting each other. In this case, for each vertex of the intersection, simply adding the elevation of each crater does not give a good visual, instead, it was chosen to add negative displacement values and limit the positive ones to the maximum height of the highest crater. An example of the intersection result can be seen in Figure 8.

3.2. Hash Table Structure

To perform the deformation of a vertex or the coloring of a fragment of the terrain, it is necessary to know which craters are close to that vertex or fragment. Although iterating through all the craters and checking the distances is possible, it would be highly inefficient. To improve performance, our method uses a spatial GPU-hash table [Pozzer et al. 2014], to identify which craters are most likely to be in contact with a given point on the terrain and avoid unnecessary ones.

The hash table is created by segmenting the terrain into a rectangular grid composed of square cells of equal size. Each cell contains a vector of structures that hold the necessary data, as described in Section 3.1, to represent a crater. The dimensions of each cell are defined in world coordinates and correspond to a specific square section of the terrain in the world. If a cell contains craters, they exist in the corresponding position on the actual terrain.

Algorithm 1 Calculate Deformation Displacement

```
Input: crater, vertex
Output: deformation displacement
$dist \leftarrow \texttt{distance}(\texttt{crater.worldPosition}, \texttt{vertex.position})$
                                        ▷ EDZ = External Deformation Zone
if $dist > \texttt{crater.EDZ}$ then
    return $0$
end if
                                        ▷ IDZ = Internal Deformation Zone
if $dist \geq \texttt{crater.IDZ}$ then
    $x \leftarrow (dist - \texttt{crater.IDZ})/(\texttt{crater.EDZ} - \texttt{crater.IDZ})$
    $y \leftarrow sqrt(1 - x * x)$
    $y \leftarrow \texttt{smoothstep}(0, 1, y)$
    return $y * \texttt{crater.height}$
end if

$x \leftarrow dist$
$y \leftarrow \texttt{smoothstep}(0, \texttt{crater.IDZ}, x)$
return $y * (\texttt{crater.height} + \texttt{crater.depth}) - \texttt{crater.depth}$
```

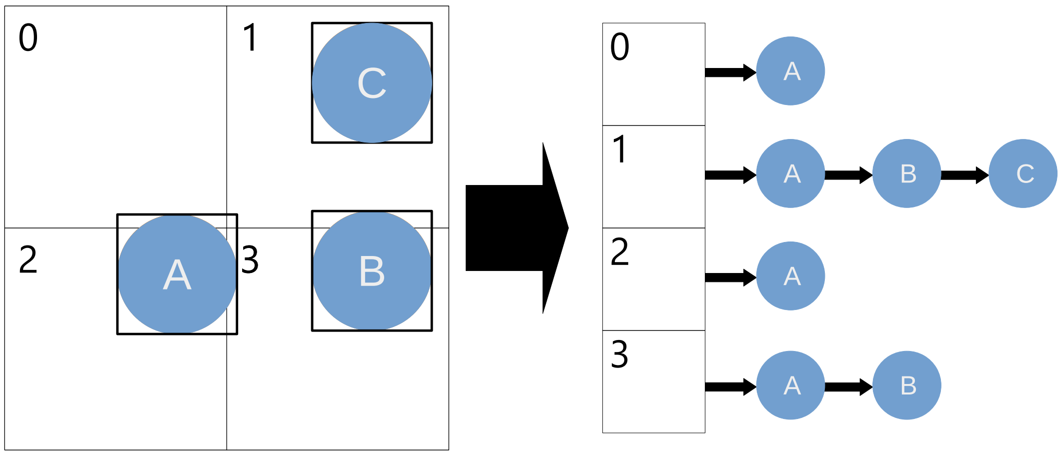


Figure 3. Example of craters approximated as squares being mapped to the hash table cells

When an explosion or equivalent event occurs, the relevant variables are computed in a serializable structure, and a new entry for it is added to a permanent list stored in the RAM. In real-time, this list is verified in regular intervals and if any new entry has been made, the current hash table in the GPU is released, and a new one is created on the CPU and sent to the GPU. This process also triggers the compute shader to deform the blocks of the height map that are already loaded.

During the construction of the hash table, craters are approximated as squares with side lengths equal to the craters' diameters and centered at the craters' centers. Each cell that intersects with these squares receives the relevant information about the craters to store. As a result, multiple cells may point to the same crater as can be seen in Figure 3.

When it needs to know if a given point of the terrain (e.g., vertex or fragment) belongs to a crater, the world position of this point is used to determine the corresponding cell in the hash table grid. Since each cell can contain multiple craters, an iteration process is required to examine each crater inside the cell. The distance between the desired position and the crater's center position is calculated, and if it is found to be smaller than the radius, that crater is considered in the deformation calculus.

3.3. Rendering the Craters

Various graphic techniques have been implemented and combined in order to represent realistic deformations caused by explosions using the simple crater structures proposed.

3.3.1. Vertex Displacement

The vertices of the craters are the same as those that compose the terrain. Being a height map-based terrain, modifying the height map by incorporating the displacement values corresponding to all the created craters, will accurately simulate the desired deformation on the terrain.

When a block of the height map is loaded, a compute shader is called. The shader examines each pixel in the texture by accessing the hash table to check if the pixel's equivalent world position is within range of a crater. If one or more craters are found within range, the displacement function is called with the craters' information and the pixel's world position as parameters. Using these parameters, the new height of the pixel is calculated and added to the height map's value. It is important to note that this modification occurs only on the GPU and does not affect the original height map texture.

3.3.2. Adding noise

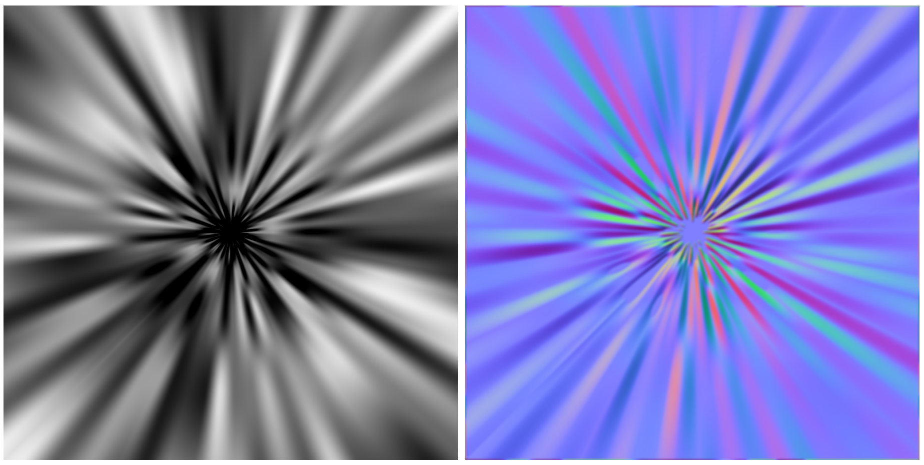
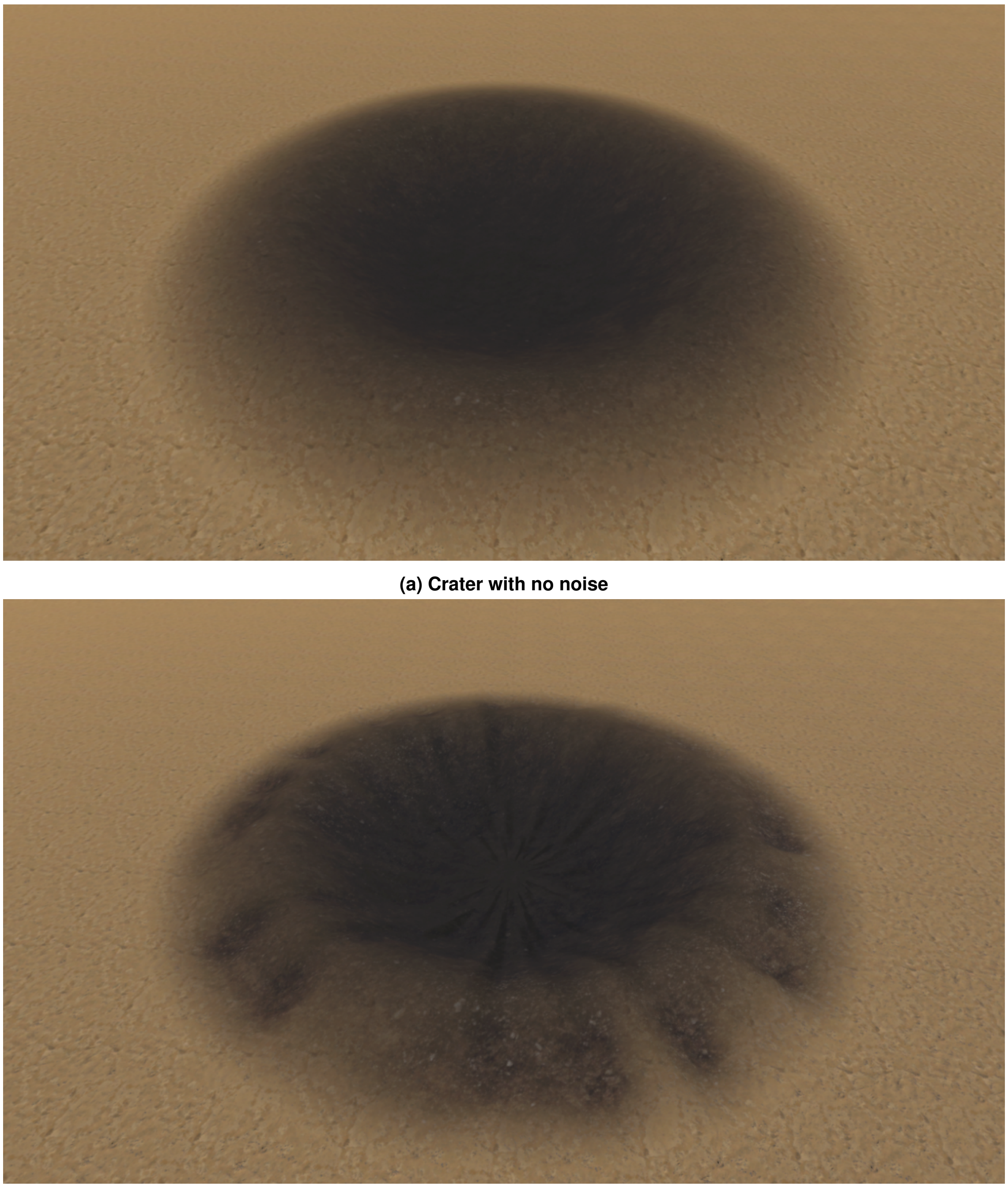


Figure 4. An example of a radial Perlin Noise texture and its normal map

As explained by [Shao et al. 2020], using geometric shapes to represent craters can result in slim and smooth craters that do not appear realistic. In order to create more natural-looking craters, some form of noise must be added to them. [Shao et al. 2020] employed a simple spatial Perlin Noise [Perlin 1985] applied to the entire area of the crater, resulting in homogeneously spread irregularities. To improve upon this approach, we utilized the same spatial Perlin Noise but applied it in polar coordinates with the center of the crater as its origin, which we refer to as ”radial Perlin Noise.”


(a) Crater with no noise

(b) Crater with noise

Figure 5. Crater with and without noise comparison

Using the crater center as the origin of a polar coordinate system it is possible to calculate radial noise values for each crater in real-time. However, it's also a good idea to pre-compute these values with random seeds and store them in gray-scale noise textures similar to the one in Figure 4. Utilizing textures is generally computationally lighter than performing calculations in real-time, but the main advantage is to use them to pre-compute normal maps, as illustrated in Figure 4.

During the coloring process, the normal map can be blended with the terrain's normals, enhancing the visual coherence of the crater. The gray-scale texture serves multiple purposes: it can contribute to the deformation calculation, adding depth irregularities to the terrain, and can also be used in the coloring as explained in the next section.

3.3.3. Coloring the craters

Naturally, it is expected that an explosion strong enough to cause deformation in the terrain would also affect its appearance. To achieve a realistic result, various textures and colors were utilized. A high-resolution dirty texture was used as the base, and on top of that, a dark color was added based on the distance from the center of the crater to create the impression of burned soil.

The transitions between the colors of the crater and the surrounding terrain are controlled by the distance of each pixel from the crater's center, ensuring the gradual blending of the crater's borders with the terrain. To enhance the transition effect and prevent a too-smooth blending, the distance from the center is combined with values from the noise texture detailed in Subsection 3.3.2. This combination results in a more nuanced and realistic transition between colors, as can be seen in Figure 5.

If multiple craters intersect, blending their colors can be accomplished relatively easily. However, if the number of intersecting craters is significant, this process can become computationally intensive. To address this issue, the decision was made to only take into account the two most recently added craters when determining the color of the region at the intersection.

4. Optimization

In the GPU, determining the presence of craters at a specific location requires accessing the corresponding cell in the hash table. However, if that cell points to many craters, it results in a high number of iterations during access. This issue arises when the cell size is not suitable for the application, such as when deformations are concentrated in areas smaller than the hash table cell size.

A high concentration of craters per cell is particularly problematic for coloring because, as explained in Section 3.3.3, the color calculation must access the hash table every frame, directly affecting the FPS. To address this issue, it is necessary to choose an appropriate cell size. If many craters are likely to be created close to each other, smaller cell sizes should be preferred.

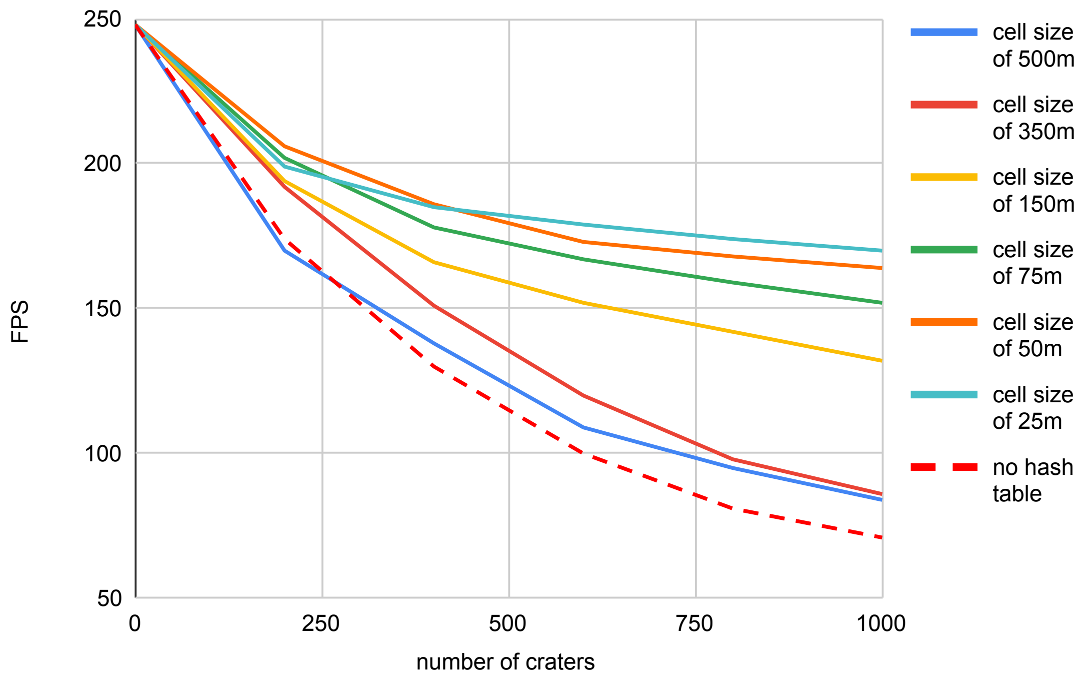


Figure 6. Performance measurement of coloring craters in a 500-meter diameter area

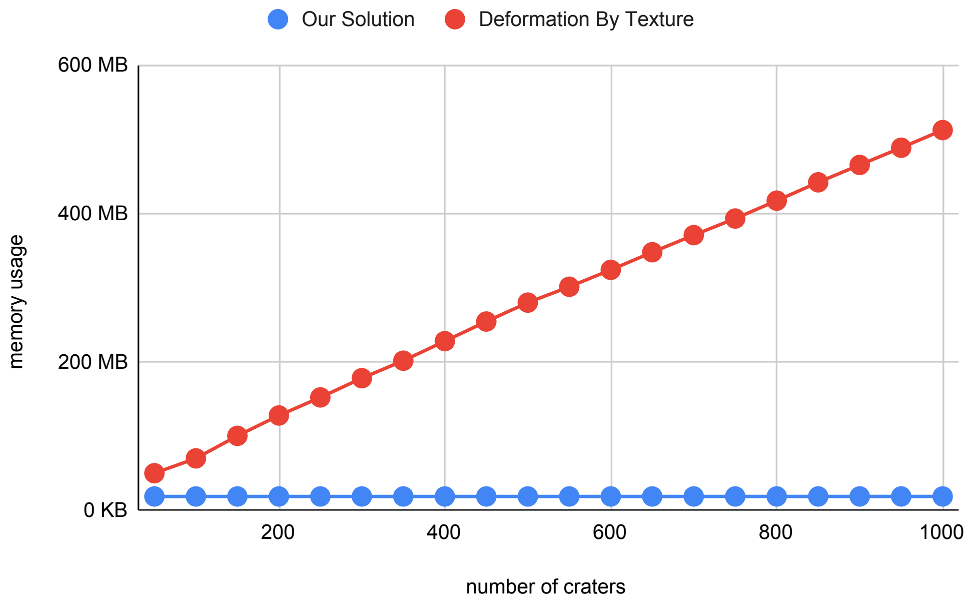
To validate these hypotheses, we conducted tests to evaluate the performance of our method in such scenarios. We utilized a terrain built with a Geometric Clipmap and a hash table with 1024 by 1024 cells. We recreated the terrain six times, varying each time the side length of the hash table cells from 500 meters to 25 meters. In all terrains, we gradually added new craters over time, starting from zero and going up to 1000, while measuring the FPS. To maximize the overlap of craters, all were randomly added within a circular 500-meter diameter area, which matches the largest cell size we tested. The craters' size varied from 5 meters to 20 meters in diameter.

Since the goal of this test is to see how the hash table behaves in different configurations, it was made an extra experiment using no hash table at all. Without it, any attempt to colorize a fragment of the terrain should require iterating over all the craters, even the ones that are further from the fragment's world position.

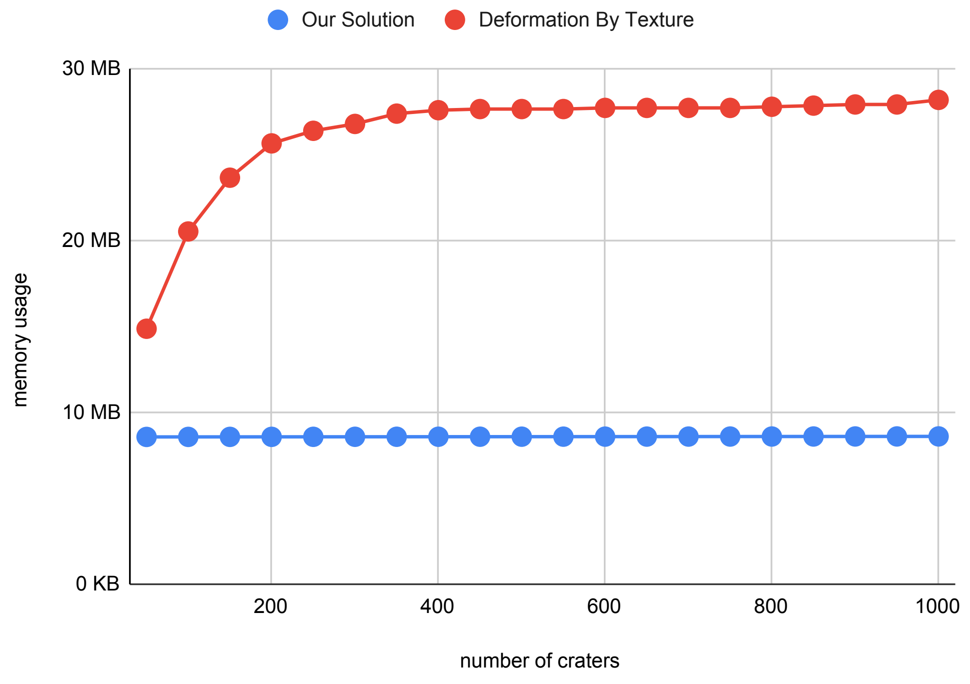
The results of the experiment can be seen in Figure 6. These tests were conducted on a computer equipped with an RTX 3060, i5 12400F, and 32GB of RAM while rendering a large-sized terrain in full HD resolution. The chosen terrain for testing employs Geometric Mipmap and it was developed inside the Unity game engine which uses meters as its default wolds coordinate unit system.

As can be seen in Figure 6, the presence of a significant number of craters concentrated in a few hash table cells has a notable impact on performance, leading to an almost 50% decrease in FPS during one of the tests. It is evident that the hash table is essential to the viability of this technique and smaller cell sizes can help in multiple overlapped craters scenarios. The figure also indicates that a cell size of approximately 75 meters is optimal. Beyond this value, there is no significant improvement in performance, likely because the cell size approaches the average crater diameter used during the test.

5. Experiments and Results



(a) 10,000-meter diameter area



(b) 500-meter diameter area

Figure 7. Memory usage comparison

The main purpose of our method is to save memory by not storing the deformation of a crater directly but instead storing the metrics necessary to calculate it during terrain loading. Traditional terrain deformation methods [Crause et al. 2011] [Aquilio et al. 2006] use textures to store deformation data, which works well for small terrains. However, on larger scales, the memory required by these textures becomes prohibitively expensive.

[Crause et al. 2011] addressed this issue by using a caching system that keeps only the most likely-to-be-used height map blocks in memory. However, their approach has a limitation: deformations cannot occur too far from the player's position because it would require loading height map blocks that are not in the cache.

To evaluate the memory-saving performance of our method, we conducted two tests under the same configuration described in Section 4, using a hash table cell side length of 500 meters. The only difference was the test areas: one was a small circular area with a 500-meter diameter, and the other was a larger area with a 10,000-meter diameter. During these tests, we measured memory consumption.

We compared our solution to a classical texture deformation method [Crause et al. 2011], where deformations are stored in height map textures and used to reshape the terrain during rendering. The textures used were 128 by 128 pixels, with each pixel representing 1 meter at the most refined LOD. For accuracy, we measured the total memory usage, including all components, not just the memory used by caching mechanisms or similar techniques.

Figure 7 demonstrates that deformation using height map textures is advantageous for small areas since many of the deformations are stored within the same texture, maintaining consistent memory usage. However, even under these conditions, the use of textures results in significantly higher memory consumption compared to our method.

6. Visual Results

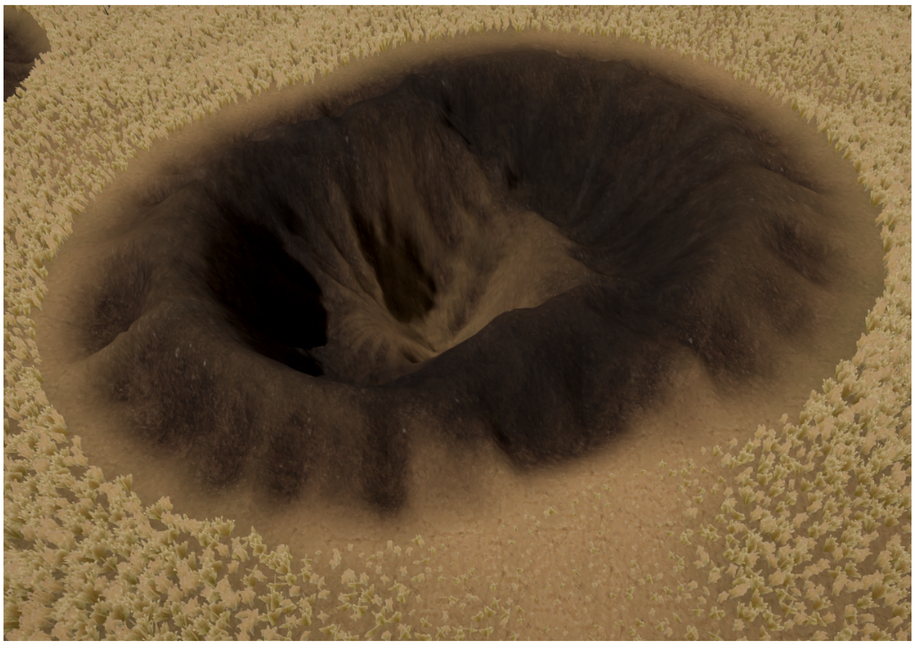


Figure 8. Double crater intersection

As described in Section 3.1, the shape of our craters can vary based on four distinct variables: R1, R2, H4, and D5. In this section, we present illustrative examples of craters generated using our solution, showcasing different soil types and sizes.

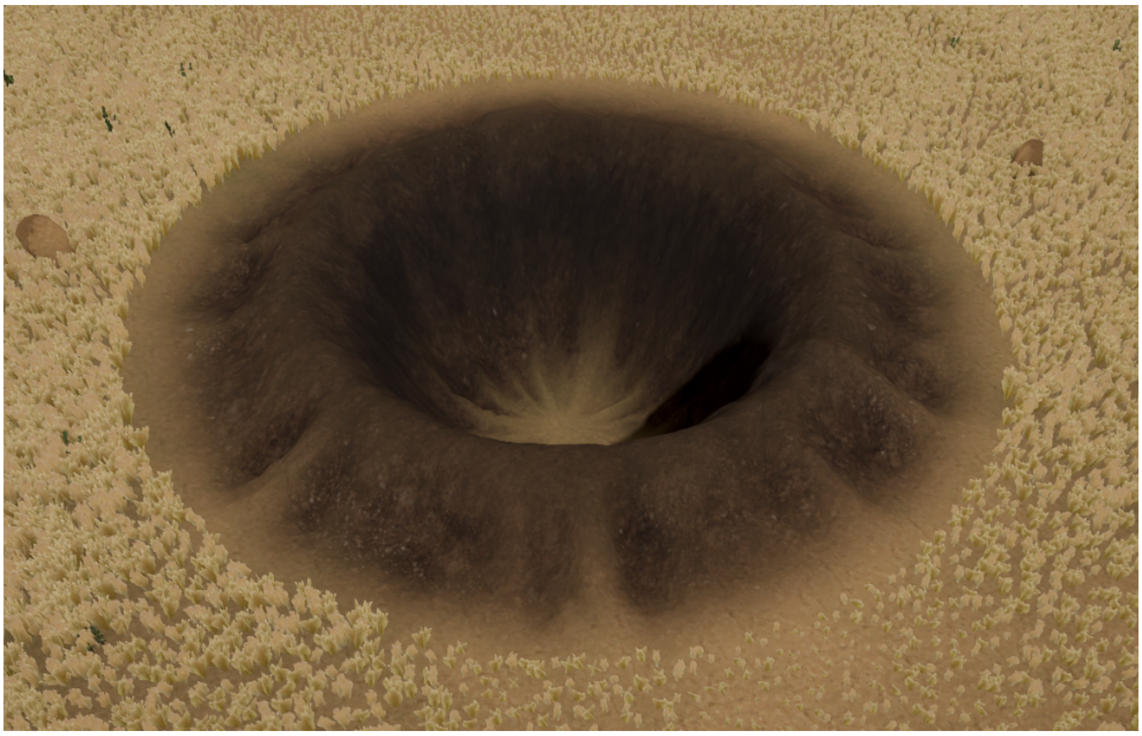


Figure 9. Example of a crater with high values for depth and height variables

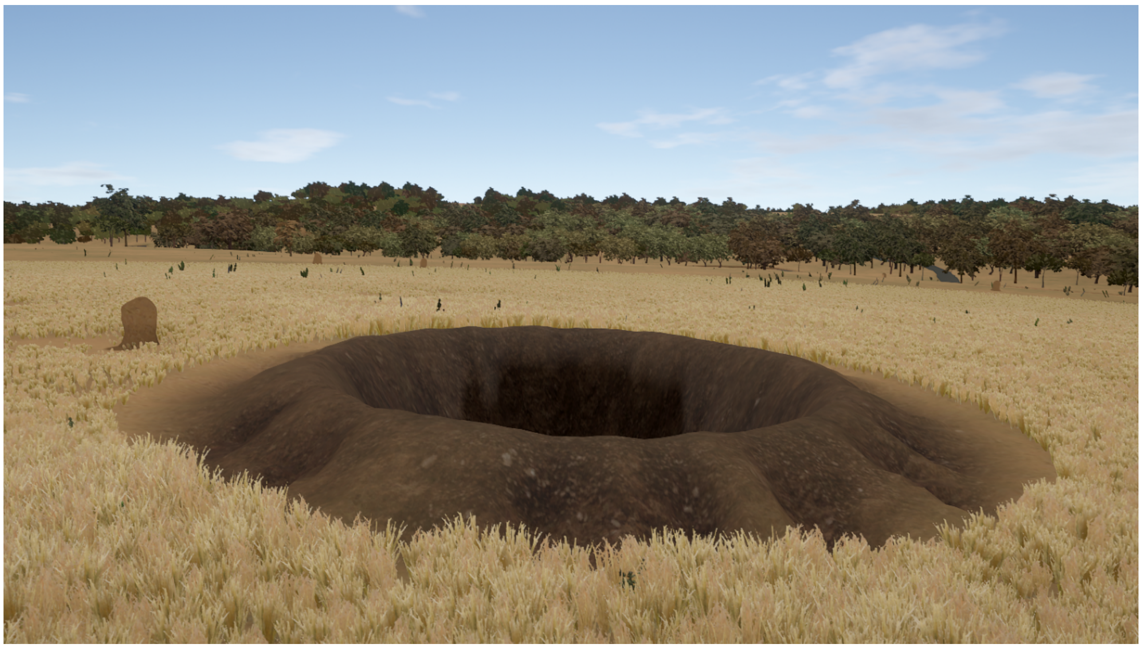
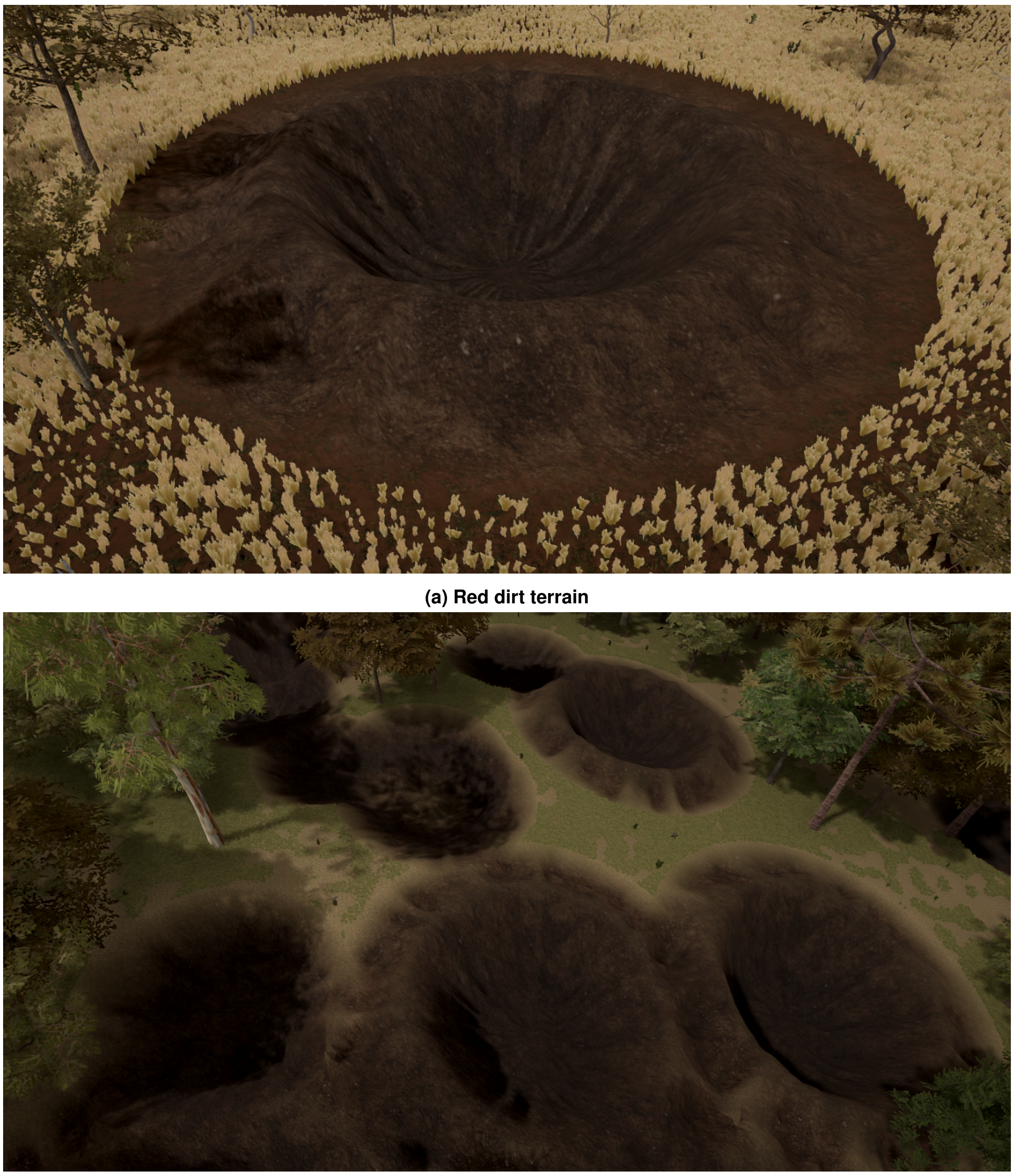


Figure 10. Example of a crater viewed from first person

Figures 12 and 9 illustrate the range of crater shapes and sizes our parameterization can achieve. The larger crater demonstrates the use of high depth and significant height, resulting in pronounced deformation. In contrast, the smaller crater shows the effect of shallow depth and no height, resulting in less prominent deformation. By adjusting these parameters, we can create a diverse array of crater shapes and sizes. Craters from Figure 10 and 13 also showcase the parametrization but when seen from the first person.


(a) Red dirt terrain

(b) Dense vegetation terrain

Figure 11. Craters generated using different scenarios

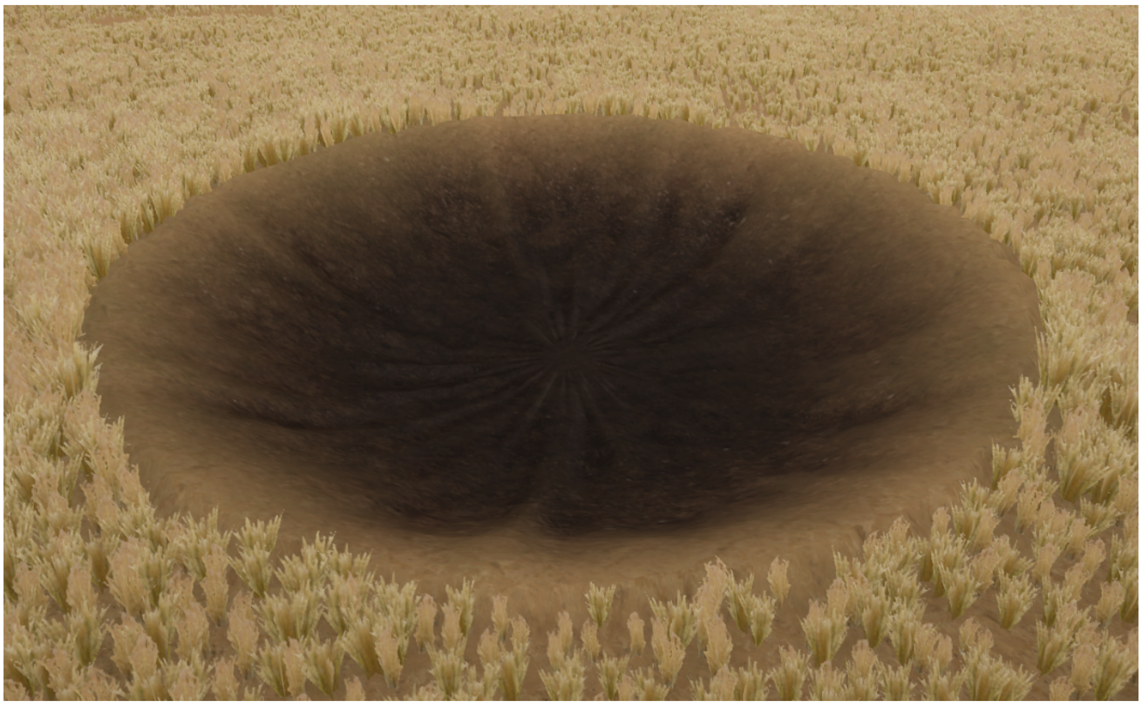


Figure 12. Example of a crater with no height. Internal and external deformation zones values are equal to zero

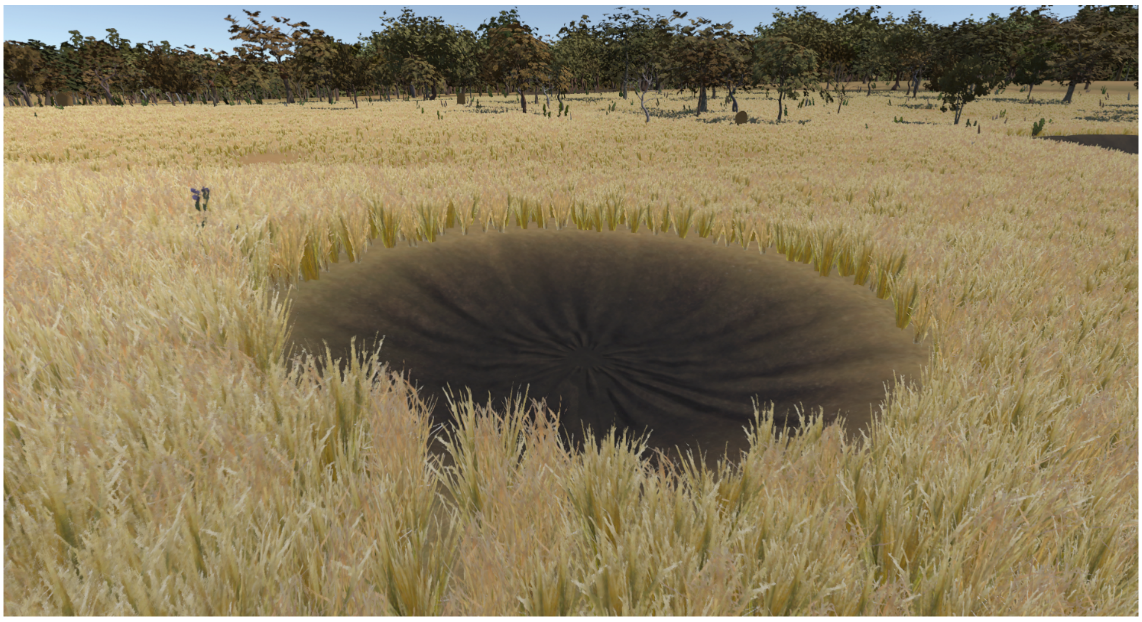


Figure 13. Another example of a crater viewed from first person

Figure 8 provides visual illustrations of the interaction between multiple craters within our solution. In these examples, we showcase the accurate vertex displacement calculation, as described in Section 3.1 which enables us to recreate the deformations caused by the presence of multiple craters. In Figure 11, it can be seen how the craters can be applied to different types of vegetation and colors.

7. Conclusion and Future Work

In this paper, we presented a new approach for simulating real-time terrain deformations caused by explosions in video games. Our method uses a hash table structure to store deformation information, allowing us to create deformations even in areas that are at great distances from the player without compromising memory. As a result, the game can simulate deformation events that happen simultaneously over large terrains.

There are several directions in which this work can be extended. One possible avenue is to explore the use of different data structures to store deformation information, such as octrees or other spatial partitioning structures. Another direction is to investigate how this approach can be applied to 3D-based terrain systems, where the problem of overlaps between deformations can be more challenging. Additionally, the integration of physically-based simulations for the terrain deformation caused by explosions could increase the realism of the simulation. Finally, our approach can be extended to handle other types of interactions with the terrain, such as impacts caused by bullets, vehicles, or other objects.

Acknowledgments

We thank the Brazilian Army and its Army Strategic Program ASTROS for the financial support through the SIS-ASTROS GMF project (TED 20-EME-003-00).

References

Aquilio, A. S., Brooks, J. C., Zhu, Y., e Owen, G. S. (2006). Real-time GPU-based simulation of dynamic terrain. In Bebis, G., Boyle, R., Parvin, B., Koracin, D., Remagnino, P., Nefian, A., Meenakshisundaram, G., Pascucci, V., Zara, J., Molineros, J., Theisel, H., e Malzbender, T., editors, *Advances in Visual Computing*, volume 4291, pages 891–900. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

Crause, J., Flower, A., e Marais, P. (2011). A system for real-time deformable terrain. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists Conference on Knowledge, Innovation and Leadership in a Diverse, Multidisciplinary Environment*, pages 77–86. ACM.

de Boer, W. H. (2000). Fast terrain rendering using geometrical MipMapping.

Duchaineau, M., Wolinsky, M., Sigeti, D., Miller, M., Aldrich, C., e Mineev-Weinstein, M. (1997). ROAMing terrain: Real-time optimally adapting meshes. In *Proceedings. Visualization '97 (Cat. No. 97CB36155)*, pages 81–88. IEEE.

Losasso, F. e Hoppe, H. (2004). Geometry clipmaps: terrain rendering using nested regular grids. 23(1):769–776.

Perlin, K. (1985). An image synthesizer. 19:287–296.

Pozzer, C. T., de Lara Pahins, C. A., e Heldal, I. (2014). A hash table construction algorithm for spatial hashing based on linear memory. In *Proceedings of the 11th Conference on Advances in Computer Entertainment Technology*, pages 1–4. ACM.

Shao, X., Xu, B., Niu, S., Jin, T., Wang, S., e Guo, C. (2020). Explosion force and shape fitting based modelling of dynamic virtual crater deformation. 502:012006.

Wan, T. R. e Tang, W. (2002). Explosive impact on real-time. In Vince, J. e Earnshaw, R., editors, *Advances in Modelling, Animation and Rendering*, pages 247–265. Springer London.