Marching Cubes-based Terrain Generation and Surface Editor

Qisong Zeng zengq3@rpi.edu

Tianshi Zhou zhout5@rpi.edu

Abstract

In this paper, we introduce an efficient terrain generation system using Marching Cubes and Perlin Noise, which creates realistic landscapes. Furthermore, we present a real-time terrain surface editor based on the Marching Cubes algorithm, developed within the Unity engine. Our proposed system enables users to interactively modify terrain surfaces with a mouse, facilitating the creation of customized landscapes through protrusions and indentations. The paper discusses the implementation, results, limitations, and future directions of this innovative approach to terrain editing and generation.

Introduction

Real-time terrain editing and generation play a crucial role in the fields of computer graphics and game development, as they enable the creation of dynamic and engaging environments. These techniques allow designers to make on-the-fly adjustments to landscapes and generate diverse terrains, resulting in more efficient workflows and the possibility of creating unique, player-driven experiences. Considering the importance of real-time terrain editing and generation, this paper introduces our approach to implementing a real-time terrain surface editor and generation system using the Marching Cubes algorithm and Perlin Noise.

Our method focuses on delivering an interactive terrain editing and generation system, allowing users to customize and create landscapes. The Marching Cubes algorithm, a well-established technique for generating polygonal representations of 3D scalar fields, serves as the foundation for our approach. In combination with Perlin Noise, we provide a

robust solution for generating realistic terrains. By integrating these algorithms into the Unity engine, we offer a user-friendly solution for modifying and generating terrain surfaces with ease and precision.

Throughout this paper, we will discuss the implementation of our Marching Cubes-based terrain surface editor and generation system, highlighting its core features and advantages.

Related Work

Our real-time terrain surface editor based on the Marching Cubes algorithm builds upon a rich body of previous research in the field of computer graphics and terrain generation. The Marching Cubes algorithm, first introduced by Lorensen and Cline [1], has become a widely adopted technique for constructing high-resolution 3D surfaces from scalar fields. Their work laid the foundation for our approach, enabling the generation and editing of terrain surfaces.

Bourke's work [2] on polygonizing a scalar field further elaborated on the practical implementation of the Marching Cubes algorithm, providing valuable insights into the process of generating complex 3D models from scalar data. A key contribution of Bourke's work that proved instrumental in our project was the introduction of a simple lookup method using bit masks and a table. This approach allowed us to efficiently identify the 256 distinct cases required for the Marching Cubes algorithm. By incorporating Bourke's table and bit masks into our implementation, we have significantly streamlined the terrain generation and editing process within the Unity engine.

Moreover, Anderson's implementation of the Marching Cubes algorithm [3] serves as a crucial reference for our project. Anderson's work offers a comprehensive and accessible explanation of the algorithm's mechanics, as well as detailed guidance on its implementation.

An essential component of our terrain generation process is the use of Perlin noise, as introduced by Perlin in his seminal paper [5]. Perlin noise has become a popular method for generating natural-looking textures and terrain in computer graphics applications. By incorporating Perlin noise into our terrain generation process, we have been able to create more realistic and visually appealing landscapes within our terrain surface editor.

By building upon the foundations established by these pioneering works, our real-time terrain surface editor expands the utility and applicability of the Marching Cubes algorithm within the realm of game development and computer graphics applications. These pioneering works have not only provided us with the theoretical foundations and practical insights necessary for implementing the Marching Cubes algorithm, but also served as invaluable references for optimizing our approach and expanding the applicability of our terrain surface editor in the realms of game development and computer graphics applications.

Implementation

1. Marching Square Generation

In our implementation of the Marching Squares algorithm, we utilize a 4-bit bitmask to represent the four vertices of each square cell. Each bit within the bitmask corresponds to a vertex, with a 0 signifying that the value at the vertex is below the isosurface value, and a 1 indicating that the value at the vertex is above the isosurface value.

To store the vertices of the marching square, we employ pairs composed of two elements. The first element encapsulates the coordinate information of the vertex, while the second element contains a value generated by 2D Perlin Noise, based on the vertex's position. In practical applications, we compare this Perlin Noise-generated value with a user-defined threshold. Should the Perlin Noise-generated value exceed the threshold, we regard the vertex as being above the isosurface value and adjust the corresponding bit in the bitmask to 1.

The bitmask values facilitate the identification of one of the 16 possible cases into which the marching square falls. Upon determining the specific case, we reference a lookup table to establish which edges will be connected. The final shape is subsequently formed by linking the midpoints of these edges, as illustrated in the left image of Figure 1.

However, directly connecting the midpoints of the edges frequently results in jagged shapes. To achieve smoother shapes, we apply interpolation to the points connecting the edges. This interpolation process is dependent on the Perlin Noise-generated values stored in the two vertices of an edge, as well as the user-defined threshold. Consequently, we obtain a smoother image, as depicted by the right image of Figure 1. The interpolated image displays significantly fewer jagged edges and appears more refined compared to the unprocessed shape on the left.

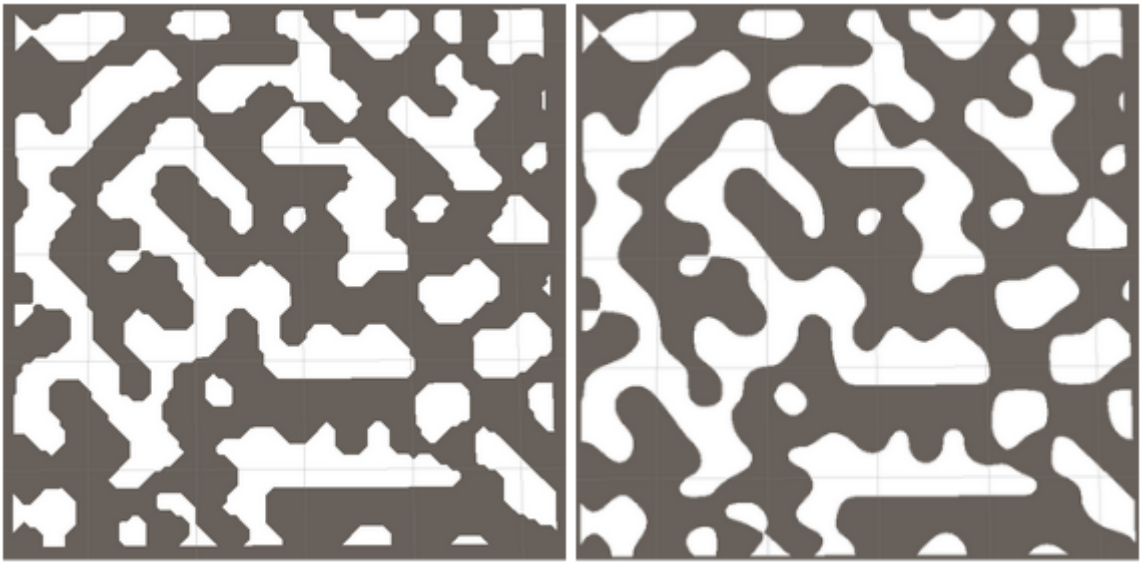

Figure 1: *Terrain generated by Marching square and Perlin Noise. Left image shows the non-interpolated version, while the right image shows the interpolated version.*

2. Marching Cube Generation

Expanding upon the principles established in our Marching Squares implementation, we have adapted the approach to generate three-dimensional shapes using the Marching Cubes algorithm. While Marching Squares operates in a two-dimensional space and considers only four vertices, Marching Cubes functions in a three-dimensional space and accounts for eight vertices.

Analogous to the Marching Squares implementation, we employ an 8-bit bitmask to represent the eight vertices of each cube cell. A 0 within the bitmask signifies that the value at the vertex is below the isosurface value, while a 1 indicates that the value at the vertex is above the isosurface value.

Similar to the Marching Squares implementation, we also employ pairs consisting of two elements for the vertices of the marching cube. The first element encapsulates the coordinate

information of the vertex, while the second element contains a value generated by 3D Perlin Noise, based on the vertex's position. In practical applications, we compare this Perlin Noise-generated value with a user-defined threshold. If the Perlin Noise-generated value surpasses the threshold, we consider the vertex as being above the isosurface value and set the corresponding bit in the bitmask to 1.

Utilizing the bitmask values, we can determine which of the 256 possible cases the marching cube falls into. Once the specific case is identified, we consult the lookup table summarized in Bourke's paper [2] to ascertain which edges will be connected. The final shape is then formed by connecting the midpoints of these edges, as demonstrated in the left image of Figure 2.

As in the Marching Squares implementation, we can apply interpolation to the points connecting the edges to generate smoother terrain shapes using 3D Perlin Noise, as demonstrated in the middle image of Figure 2. However, this level of smoothness might not be adequate for certain 3D terrains. To achieve even smoother shapes, we are required to recalculate the normals for each vertex. For every vertex, we compute a weighted average of the normals of all adjacent triangles, factoring in the areas of these triangles. By recalculating the normals for each vertex, we achieve a smoother terrain, as illustrated in the right image of Figure 2. This enhanced smoothness leads to more visually appealing and realistic three-dimensional shapes.

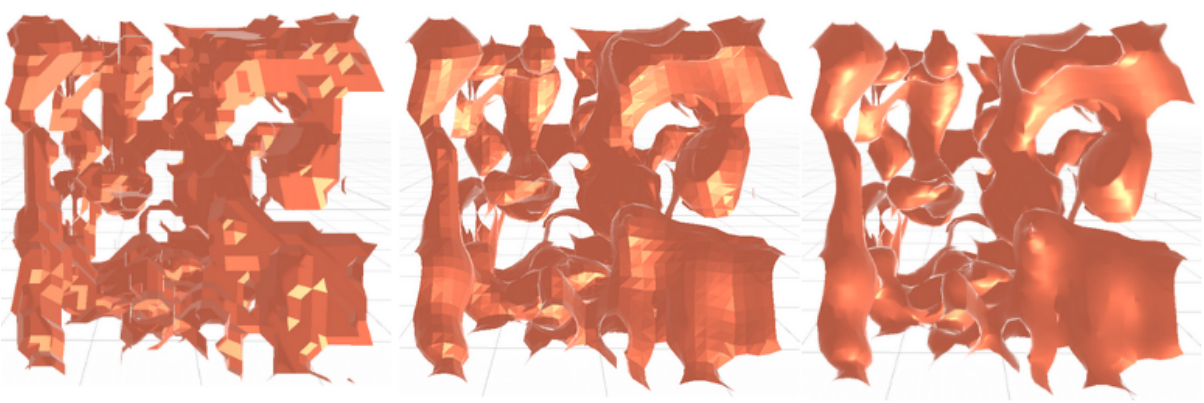

Figure 2: *Terrain generated by Marching cubes and Perlin Noise. The left image shows the non-interpolated version, the middle image shows the interpolated version, and the right image shows the terrain with normals reconstructed for each vertex.*

3. Interactive Terrain Editing with Player Controller

In the terrain editing component of our implementation, we have developed an intuitive and user-friendly controller to enhance user interaction. This controller enables users to navigate within the scene by employing the WASD keys on their keyboard. Terrain editing can be performed by left-clicking the mouse, with two distinct options available for users: indenting the surface and creating protrusions.

The core principle behind the terrain editing feature is the use of a ray emitted from the camera in the direction the user is facing. This ray detects any colliding objects within its path. Upon identifying the colliding object, we employ raycasting to gather relevant data. Subsequently, we perform a spherical examination based on the user-selected brush size. This examination allows us to identify all the vertices of the marching cubes present within the detection range. After locating the vertices, we modify the ones situated on the surface facing the user, tailoring the terrain according to the user's preferences.

Once the modifications have been made, we update the vertices of all marching cubes within the specified range. This approach enables users to connect adjacent marching cubes seamlessly, resulting in a smooth and cohesive terrain surface. The terrain editing system allows for a high degree of customization, empowering users to create diverse landscapes that cater to their unique requirements and creative visions.

Figure 3 illustrates the step-by-step process of indenting the terrain surface, showcasing the transformation of the landscape as the user interacts with it. In contrast, Figure 4 demonstrates the procedure for creating protrusions on the surface, highlighting how users can build elevated features to enrich their terrain. With this interactive and dynamic terrain editing system, users can achieve highly customized terrain surfaces.

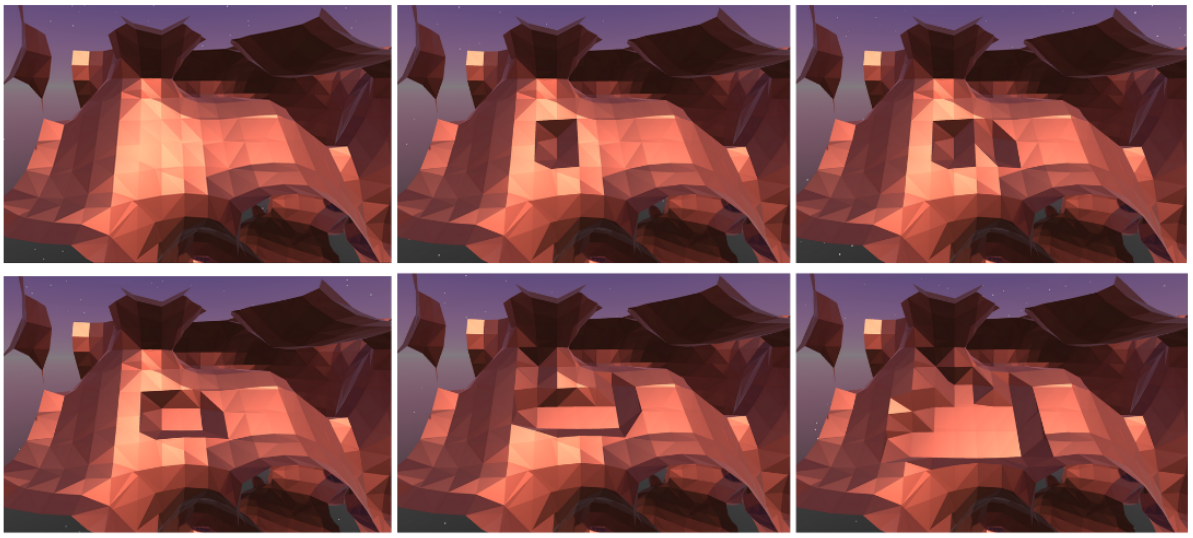

Figure 3: *Step-by-step process of indenting the terrain surface, displayed from the top left to the bottom right.*

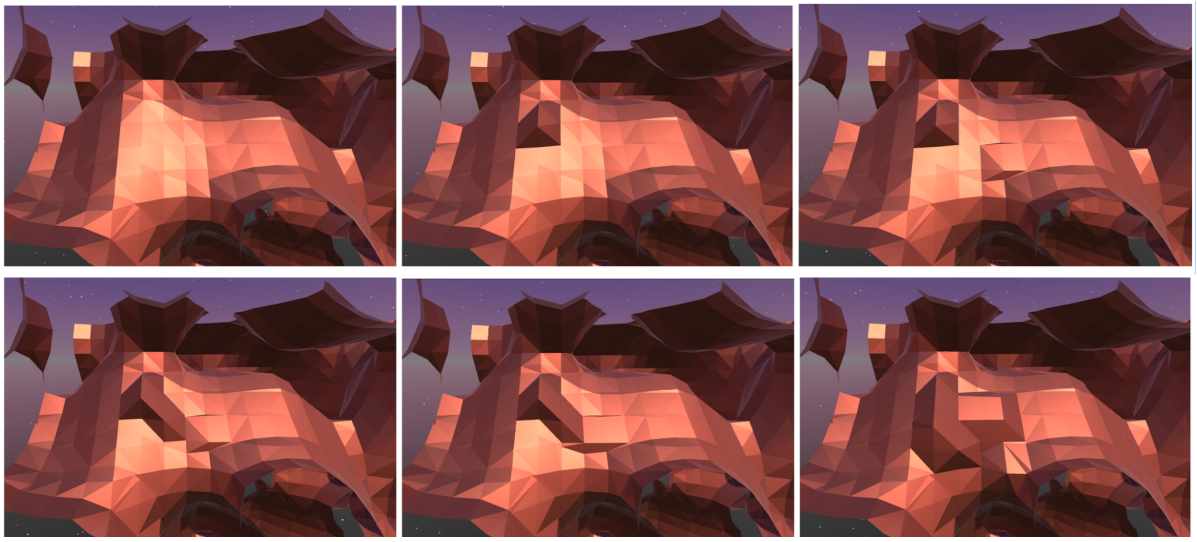

Figure 4: *Step-by-step process of creating protrusions on the terrain surface, displayed from the top left to the bottom right.*

Results

The results of our study demonstrate the effectiveness of our real-time terrain generation and editing system, which incorporates the Marching Squares and Marching Cubes algorithms, as well as Perlin Noise. Our implementation within the Unity engine enables users to interactively create and modify terrain surfaces with ease and precision.

Our Marching Squares implementation generates two-dimensional terrain with a smooth appearance by employing interpolation, resulting in reduced jagged edges and a more refined visual outcome.

Similarly, our Marching Cubes implementation generates three-dimensional shapes with progressive improvements in smoothness and visual appeal. The use of interpolation and vertex normal reconstruction further enhances the terrain's appearance, providing more realistic results.

In addition to terrain generation, our implementation features an interactive terrain editing system, allowing customization of the terrain surface. Our user-friendly controller and intuitive editing options enable users to create indentations and protrusions on the terrain surface, empowering them to achieve diverse and engaging landscapes.

In conclusion, our terrain generation and editing system, utilizing the Marching Squares and Marching Cubes algorithms, provides an effective method for real-time creation and modification of terrain surfaces.

Limitations

Despite the promising results and advantages offered by our terrain surface editor based on the Marching Cubes algorithm, it is essential to recognize and address its limitations. One primary limitation lies in the computational demands of our approach, as all terrain generation and editing calculations are performed on the CPU. This can lead to a computationally expensive process when attempting to generate expansive terrains, potentially resulting in increased processing times and a noticeable reduction in performance. This limitation is particularly significant when dealing with applications that require large-scale landscapes or continuous terrain modifications.

Another constraint stems from the use of the CPU for terrain generation within the Unity engine, which imposes a limitation on the number of vertices allowed in a single mesh for a game object. With a maximum threshold of 65,536 vertices per mesh, this constraint necessitates dividing large objects or terrains into smaller segments for generation, as opposed to creating them in one unified piece. Failure to adhere to this limitation can lead to substantial issues, such as partial mesh generation, where only a fraction of the intended

terrain is rendered, and the remaining portion fails to be generated, ultimately impacting the visual quality and usability of the generated terrain.

In summary, our Marching Cubes-based terrain surface editor has some limitations that need to be acknowledged. These limitations primarily involve computational demands and restrictions on the number of vertices in a single mesh within the Unity engine. In the future work section, we will explore potential improvements and solutions to address these limitations, ensuring the ongoing viability and effectiveness of our approach.

Future Work

Building upon the current limitations of our terrain surface editor, we have identified several areas for improvement and exploration in future work. Firstly, we plan to transition from a CPU-based terrain generation approach to a GPU-based one by employing compute shaders within Unity, as suggested by Geiss [4]. This change would help alleviate the computational burden associated with terrain generation and editing, thereby enhancing performance and enabling the creation of more expansive landscapes. Additionally, employing GPU-based terrain generation will help overcome the 65,536 vertex limit imposed by Unity's mesh generation system, allowing for larger and more detailed terrains.

Another area of improvement involves implementing spatial partitioning techniques for terrain generation. This approach would not only allow for more intricate and larger terrains but also facilitate efficient rendering and level-of-detail management. By partitioning the terrain into smaller segments, we can optimize the rendering pipeline, selectively updating and rendering only the segments necessary for the user's current view. Spatial partitioning

will contribute to faster terrain generation, especially for large scenes, ultimately leading to performance and efficiency improvements.

Furthermore, we plan to incorporate occlusion culling techniques in conjunction with procedural noise generation to improve overall game performance. By generating terrain only within the camera's visible range and disregarding areas outside the field of view, we can significantly reduce the computational demands associated with terrain generation, leading to improved performance and a more seamless user experience.

Lastly, our current implementation does not include texture calculations due to the performance limitations of the CPU-based approach. In future iterations, drawing inspiration from Geiss's work on procedural terrains using the GPU [4], we aim to integrate texturing capabilities to enhance the visual appeal of the generated terrain, creating more realistic and visually engaging landscapes.

By addressing these areas of improvement, we believe that our Marching Cubes-based terrain surface editor can be further refined and optimized.

Conclusion

In this paper, we have presented our real-time terrain generation and surface editor based on the Marching Cubes algorithm, offering an interactive solution for creating and editing terrain within the Unity engine. Our approach leverages the power of both the Marching Squares and Marching Cubes algorithms, along with Perlin Noise, to enable users to generate and modify landscapes.

Our terrain generation and surface editor exhibits several advantages and features, including its user-friendly interface and real-time editing capabilities. However, we also acknowledge the limitations of our current implementation, particularly concerning the computational demands associated with CPU-based terrain generation and the vertex constraint imposed by the Unity engine's mesh generation system.

In response to these limitations, we have outlined a series of potential improvements and directions for future work. These include transitioning to a GPU-based terrain generation approach using compute shaders, implementing spatial partitioning techniques for more efficient terrain generation, and incorporating occlusion culling and procedural noise generation to enhance performance. By addressing these areas of improvement, we aim to refine and optimize our Marching Cubes-based terrain generation and surface editor further, ultimately striving to develop more efficient terrain editing solutions.

Acknowledgements

We would like to express our heartfelt gratitude to Professor Barbara Cutler for her invaluable knowledge, guidance, and support throughout the development of our project. We are also immensely grateful to XiaoC. Luo for providing suggestions and advice that helped to refine and improve our project. Additionally, we extend our appreciation to the Unity Asset Store (https://assetstore.unity.com/) for providing the essential art assets and extensions utilized in our project. Special thanks go to the following contributors on the Unity Asset Store for their indispensable offerings: Asset Bag for their Crosshairs Plus package, Denis Rizov for NaughtyAttributes, and BOXOPHOBIC for the Free Skybox Extended Shader. These assets have greatly enriched our project and streamlined the development of our terrain surface editor within the Unity engine. We sincerely acknowledge these contributions for

their significant impact on our work and for enabling the successful realization of our terrain surface editor within the Unity framework.

References

1. Lorensen, W. E., & Cline, H. E. (1987, July). Marching cubes: A High Resolution 3D Surface Construction Algorithm. In *ACM Siggraph Computer Graphics* (Vol. 21, No. 4, pp. 163-169). ACM.
2. Bourke, P. (1994). Polygonising a Scalar Field. Retrieved from http://paulbourke.net/geometry/polygonise/
3. Anderson, B. (n.d.). An Implementation of the Marching Cubes Algorithm. Retrieved from https://www.cs.carleton.edu/cs_comps/0405/shape/marching_cubes.html#1
4. Geiss, R. Generating Complex Procedural Terrains Using the GPU. In GPU Gems 3 (Chapter 1). NVIDIA Corporation. https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-1-generating-complex-procedural-terrains-using-gpu
5. Perlin, K. (1985). An Image Synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3), 287-296.