Planetary Marching Cubes: A Marching Cubes Algorithm for Spherical Space

Zackary P. T. Sin
The Hong Kong Polytechnic University
Hunghom
Hong Kong
+852 3400 8421
csptsin@comp.polyu.edu.hk

Peter H. F. Ng
The Hong Kong Polytechnic University
Hunghom
Hong Kong
+852 2766 7248
cshfng@comp.polyu.edu.hk

ABSTRACT
There is a growing interest in digital games with user-generated content. Games with user-generated content usually involve terrain editing and marching cubes is a popular algorithm that permits a dynamic terrain. On the other hand, there is also growing interest in games with a planetary theme. Hence, a question is asked on whether can marching cubes be used to generate a planetary terrain. This study investigates how to adopt the marching cubes algorithm in a spherical space, specifically, for generating a planetary terrain. The result is the proposed planetary marching cubes, which compared to previous methods, could generate more complex terrain features while retaining smooth surfaces.

CCS Concepts
• Computing methodologies ~ Mesh models.

Keywords
Mesh Generation, Marching Cubes, Game Engineering, User-generated Content, 3D Terrain Model

1. INTRODUCTION
Classically in digital games, game content such as characters, levels and gameplay itself is served to the player in a waterfall manner. Artists create the game content, deploy to the game and the game content will be static as it is shipped to the players. Although not a novel idea, recently there is a growing interest in game content creation that involves the players. It is referred to as user-generated content as the players could use the tools provided by the game to create their own content. It is said that this is not a novel idea as, as early as 2008, the species evolving simulation game Spore by Maxis allow players to edit characters, buildings and vehicles by exchanging, scaling, moving parts and more. It is argued that it is the sandbox game Minecraft by Mojang which caused an increased interest in user content generation as many digital games involving user content eneration comes after. Terraria by Re-Logic, Project: Spark by SkyBox Labs and Dragon Quest Builder by Square Enix to name a few. One of the key user-generated content elements in these games is that it allows the player to edit the game world's terrain.

ICVIP 2018, December 29–31, 2018, Hong Kong, Hong Kong

ACM ISBN 978-1-4503-6613-7/18/12…$15.00
https://doi.org/10.1145/3301506.3301522

Since the terrain could be modified by the players, the terrain model could not be prepared as a static model by the artists. Instead, the model needs to be supported by some algorithm to anticipate change. To generate a smooth 3D terrain that is modifiable, a popular algorithm or perhaps the algorithm to use is the marching cubes algorithm by Loresnsen [1]. In short, the idea of the algorithm is to use a 3D grid as the volume to bond the terrain. Each edge of a cube in the 3D grid consists of a parameter, or isovalue. By tuning the isovalue, the terrain mesh could be dynamically generated by computing the isosurface based on the isovalues. The marching cubes' 3D grid exists in the canonical x, y and z spatial space.

On the other hand, there also seems to be a growing interest in games based on planets. Games like Super Mario Galaxy by Nintendo, Universim by Crytivo Games and ECO by Strange Loop Games feature planets where the players could interact with. Of course, this would require a 3D model of the planet to be deployed into the game.

Here, we ask the question, how could we use marching cubes to generate a planetary terrain? This is a question worth investigating as originally marching cubes is for a canonical 3D space while a planet is best described in a spherical space. This is likely why we do not use x, y and z to describe a location on earth and instead use a geographic coordinate system to do so. Although a marching cubes algorithm could still be used to directly generate a planetary terrain, there exists a few problems. Mainly, there is an issue regarding the incompatibility of using 3D-grid-bound parameters to approximate a planetary surface. It could be imagined that the same terrain feature at different position on the planet would requires different parameters which is not optimal. Ideally, the same terrain feature regardless of positions would require similar parameters such that the terrain feature's meshes are as similar as possible. Another problem is that if we are to generate a specific planetary model and a planet is best described in spherical space, how could we, with respect to the planet, infer the suitable parameters to generate its mesh? As can be seen, directly using a typical marching cubes algorithm for generating a planetary model could cause some cumbersome problems.

In this study, planetary marching cubes (PMC), a variant of the marching cubes algorithm is proposed to generate planetary terrain instead (Figure 1). In more general terms, the proposed algorithm is a marching cubes variant that works in spherical space. To tackle

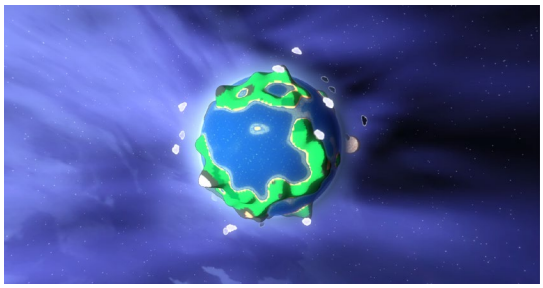

Figure 1. A planet 3D model generated with planetary marching cubes, a marching cubes algorithm adopted for generating mesh in a spherical space.

the problems aforementioned, the proposed algorithm uses a grid in spherical space instead of the canonical 3D spatial space as shown in Figure 2. So, the same terrain feature will use similar parameters to generate while it is also trivial to infer suitable parameters to generate a specific planetary terrain. In addition, it is likely desirable to generate a navigational data structure for navigation of agents played by the computer. However, since the terrain could be edited by the players, a typical navigational mesh baking method which take noticeable amount of time is likely not suitable. Hence, how a navigational grid could be conveniently generated by PMC is also proposed. Aside from generating terrains for planetary-themed games, it is expected that the proposed model could also be useful for generating 3D models for real celestial entities such as Mars. Which, could be useful for scientific education and perhaps even practical simulation.

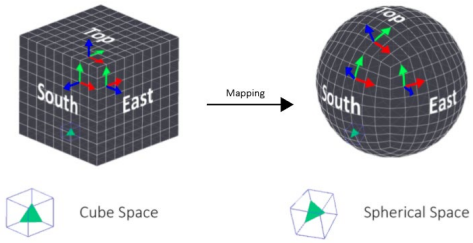

Figure 2. Projection from a unit cube to a unit sphere. The key idea of PMC is to use a spherical space instead of a canonical space for the marching cubes. Original image from [9] and modified.

2. LITERATURE REVIEW

As a computer graphics algorithm developed by Lorensen [1], marching cubes is used for generating a polygonal mesh of an isosurface by using a three-dimensional scalar field. Outside the game industry, marching cubes is usually used for medical visualization. In games, on the other hand, it is mostly used for terrain generation that could support terrain editing. There are many terrain engines that are built upon the marching cubes algorithm and the terrain of Project: Spark is likely to use marching cubes, or an adaptation, for its terrain as well.

The principal idea of the algorithm is to use a 3D grid as the volume to bond the terrain. Since it is a 3D grid, there are cubes. Each corner of a cube is what referred to as a voxel. The voxel is basically a bit that could be on (filled) or off (unfilled). Depending on the on and off combination, a cube would generate a localized mesh as shown in Figure 3. Each edge of the cube consists of an isovalue which could be tuned. By tuning the isovalues, where the mesh (isosurface) meets the edges will be changed.

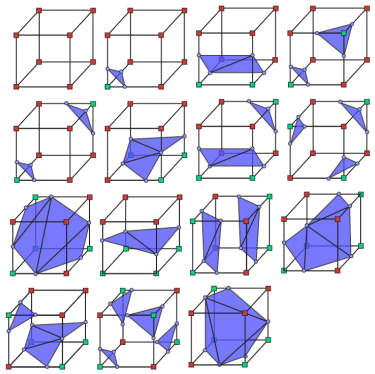

Figure 3. The 15 marching cube combinations. A small colored square represents a voxel state while a small circle represents an isovalue. Each combination will result in a different localized mesh. The blue triangles are the generated polygon.

The vanilla marching cubes algorithm or its extended algorithms could be used for generating the mesh of the planet as well, however, as mentioned, the data structure and subsequently the mesh will not be completely compatible. There will be artifacts at the eight "corners" of the planet as the volume the marching cubes covered is basically a cube with infinite size. For example, when the player is trying to create a hill feature, the hill will look significantly different at the cube's "face" and "corner". This phenomenon is basically due to the issue of unfair sampling. The axes do not align with the planet's surface and hence similar surface model will display different behavior due to different marching cube's parametrization for mesh generation. This is undesirable as at different position on the surface, the same surface model cannot be converted into the same mesh.

Adaptive methods, either adaptive marching cubes or tetrahedrizations could be used to ease the problem of unfair sampling [2, 3]. However, they could not resolve the issue of sampling as they only increase the resolution at the "corners" to attempt to reduce the difference between "corners" and "non-corners". There is also research regarding how to produce more accurate isovalues [4]. But similarly, it could only reduce the difference. It still remains true that the axes do not align with the planet's surface at the corners and as such similar model may be represented by different parameters, resulting in inhomogeneity.

Most of the previous studies on marching cubes have focused on solving ambiguity, improving performance, and extending the basic approach [5]. For example, trying to solve the face ambiguity problem that may occur in some patterns of the 256 possible marking scenarios for a cube when there are multiple facetization that could be done for the same pattern.

There exist a few marching cubes research works that are related to non-cube spaces, but they are not related to solving the sampling issue, a main focus of this study. Ref. [6] has proposed a method about reducing triangular mesh complexity (e.g. reducing the number of vertices) by reparametrizing the mesh representation in marching cubes to one represented by nearest neighbor coordinates with parameter domain in a unit sphere. Ref. [7] has proposed a

method of creating a marching cubes mesh from data collected with coordinates in cylindrical and spherical coordinates.

3. PREVIOUS METHODS

In this section, previous common methods in generating a planetary mesh are discussed. As these are methods used by the game development community, they should be easily found on the Internet.

Generally, it seems that two approaches are most popular. The first one is to use a height map on a spherical grid to generate the terrain of the planet (Figure 4). A height map is simply a 2D grid with heights representing as values. When generating the terrain, each height value will be used to infer how high the vertex should be. The planet will be divided into six faces, and each face will be represented by a heightmap. Although using height maps could create a smooth surface, it also means that more complex terrain features such as a cave or overhangs could not be represented. The game Universim seems to use this method as its terrain mesh does not exhibit complex features such as caves or overhangs.

Figure 4. A planet generated using a heightmap by Romain (https://github.com/Roldak/PlanetGeneration).

The second method is to use a voxel engine on a sphere to generate the terrain of the planet (Figure 5). Voxel engine refers to an idea of using voxels to approximate a 3D model. Not to be confused with voxel from marching cubes, each voxel in a voxel engine is simply a 3D block mesh that either exists or not. The engine will choose the relevant voxels to exist to approximate the 3D model. Games like Minecraft uses a voxel engine to generate their terrains. Similar to using heightmaps and PMC, the planet will be firstly divided into six faces. Then, each face will have its own set of voxels for generating the terrain. The result is understandably "blockish" as a voxel engine could only generate terrain mesh by choosing the correct set of voxels.

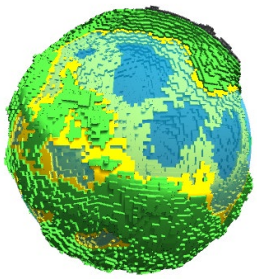

Figure 5. A planet generated using a spherical voxel engine by Frankson (https://forum.unity.com/threads/spherical-voxel-engine-cc-by.259121/#post-1721678).

4. METHODOLOGY

Here, the technical details of PMC are presented. Several problems need to be addressed. These includes how to create of a spherical space, how to convert between the coordinate system between the faces, how to organize the object hierarchy and how to resolve the inconsistency at the border of faces. In addition, how to generate a navigational grid and how to procedurally generate the terrain will also be presented.

4.1 Six Faces Voxel System

To define the spherical space of PMC, a common cube-sphere mapping equation is used:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x\sqrt{1-\frac{y^2}{2}-\frac{z^2}{2}+\frac{y^2z^2}{3}} \\ y\sqrt{1-\frac{z^2}{2}-\frac{x^2}{2}+\frac{z^2x^2}{3}} \\ z\sqrt{1-\frac{x^2}{2}-\frac{y^2}{2}+\frac{x^2y^2}{3}} \end{bmatrix} \quad (1)$$

Eq. (1) will convert a unit cube coordinate (x, y, z) to a unit sphere coordinate (x', y', z'). So, (1, 1, 1) will be mapped to $(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2})$.

This equation is quite known in the game development community. The previous methods in section 3 very likely also used this equation to create a spherical space. By converting necessary unit cube coordinate to a unit sphere coordinate, naturally, the sphere will be divided into six faces just like the cube. The base squares of the six faces will be subsequently projected on the related spherical space as well. Fig. 2 shows the result of the projection. For the height map and voxel engine approach mentioned previously, the necessary coordinates are created similarly in this step. Here for planetary marching cubes, the six faces on the unit sphere are also used to create six 3D grids. Each grid G will have its own set of voxels v.

It is proposed that each of the faces of the sphere should act as its own localized space. As shown in Figure 6, each local face should have its own coordinate system. In the Figure, only the two axes are shown. There should be one more axis, which is called the "depth" axis, that point outwards from the sphere center, hence, a localized three-axis system is formed.

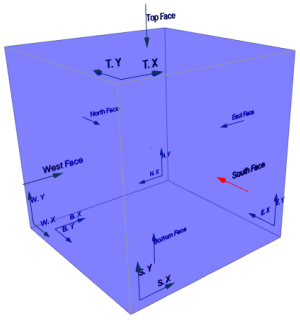

Figure 6. The six local face coordinate systems.

It is suggested that the six faces should be referred with the following names F_{Top}, F_{Bottom}, F_{North}, F_{South}, F_{East}, and F_{West}. The convention suggested for a sphere is that there is no rotation and the camera is looking the sphere from the "back" of the world space. F_{Top} and F_{Bottom} are the faces at the top and bottom of the "screen" respectively; F_{North} and F_{South} are the faces further and

closer respectively; and F_{East} and F_{West} are the faces at the left and right respectively.

4.2 Face Coordinate Conversion

There will be many instances, related to data structure or game logic, where the conversion between the six localized face coordinates become necessary. For example, if a character has move far enough and reached the boundary of a face. Although seemingly a basic operation, the conversion is not entirely trivial as neighboring faces will have a varying degree of differences. In Figure 7, we can observe that the F_{Top} and F_{South} have directly compatible face coordinates. So, the conversion will indeed be a trivial addition operation. However, the coordinates between F_{Top} and F_{East} not only do not have their x and y axes aligned, but also have one pair of their counterpart axes in the opposite direction

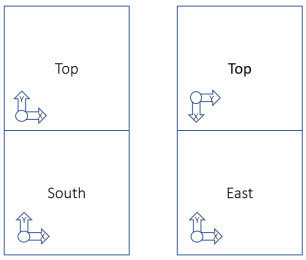

Figure 7. Neighboring local face axis systems have different degree of incompatibility. The left showcases an easy conversion, while the right showcases a problematic one as the two-axis systems are no aligned.

To this end, a coordinate conversion algorithm that could be used to convert coordinate from one face to another using simple addition, multiplication and flipping operations is devised. See **Algorithm 1** for converting the coordinates on one axis to another.

Algorithm 1 Pseudocode of the face coordinates conversion algorithm.

```
Input: Two faces, f_c, the current face, and f_t, the target face, one axis a_c and one 1D position p_c.
Output: One 1D position p_t.
Let a_t as the target axis
if f_t axis a_n with the same name as a_c is orthogonal to a_c
    a_t := OrthogonalAxis(f_t, a_n)
else
    a_t := a_n
end if
var mul := 1
var offset := 0
if IsOppositeDirection(a_t, a_c)
    offset := negate of max value of a_c
    mul := -1
else
    if position of origin of a_t has a difference with that of a_c can be measure by either a_t or a_c
    offset := OriginDifference(a_t, a_c)
    end if
end if
p_t := p_c * mul + offset
```

In **Algorithm 1**, OrthogoalAxis() retrieves the other non-depth axis of the local face axis system; IsOppositeDirection() checks if two axes are pointing different direction; and OriginDifference() retrieves the 1D difference between the origins using the axes (e.g. OriginDifference(y_{top}, y_{south}) is the size of the face while OriginDifference(x_{top}, x_{south}) is 0).

4.3 Engine Hierarchy and Parameters

In our implementation, a similar architecture to a typical voxel engine is used. As shown in Figure 8, A hierarchy of planet, faces, chunks, voxels and voxels' edges is suggested (with the former being the parent of the latter). As stated, a planet will have six faces, with each face having its own chunks. At the base of a face, there should be n_c^2 chunks as the chunks will fill the square planet face with each side having n_c chunks. For each chunk, it will have n_t^3 tiles with the width, height and depth having n_t tiles. A tile is simply a conceptual construct made up of eight voxels. It is basically the marching cube, but in this instance, the cube is not a strict cube. It has been morphed by the spherical space into a trapezoidal prism that has a smaller bottom and a larger top as shown in Figure 9. Each voxel will have three voxel edge that each expands along one of the three axes of the localized face axis system. Since each voxel edge will connect the nearby voxel, aside from the voxels at the border of a face, each voxel will be connecting to six voxel edges. Voxel will have a state of on and off to determine if it is dense and the voxel edge could be considered as an objectification of marching cubes' isovalue. Unlike the other two types of voxel edge, the type of voxel edge that expands along the depth axis has an undefined length l_d. Additionally, a base depth d_b should be set to define the radius of the unit sphere where the six faces are built. Setting a larger d_b could save memory on needless data at the core of the planet. d_c should be defined for specifying the number of chunks along the depth axis as, like l_d, it is undefined but d_c could be infinite. If d_c is infinite, chunks should be loaded dynamically

$$\eta_c = 6n_c^2 \cdot d_c \tag{2}$$

$$\eta_v = 6 \cdot (n_t \cdot n_c + 1)^2 \cdot (n_t \cdot d_c + 1) \tag{3}$$

$$r = d_b + l_d \cdot (n_t \cdot d_c) \tag{4}$$

In summary, for the planet, the total number of chunks η_c, voxels η_v and its radius r will be (2), (3), and (4) respectively. The parameters could be tuned to change the properties of the planet.

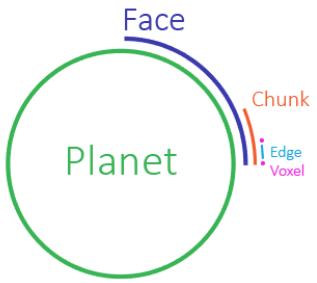

Figure 8. The hierarchy of PMC. A planet is a parent to faces; A face is a parent to chunks; A chunk is a parent to voxels; and a voxel is a parent to edges.

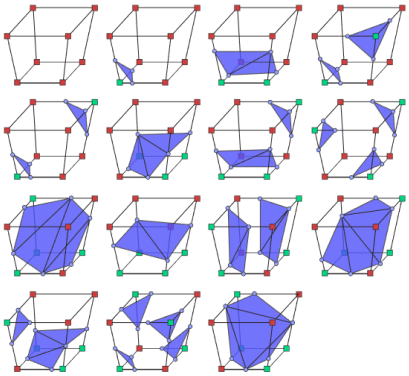

Figure 9. The 15 marching tiles combinations used in PMC.

4.4 Pairing Voxels and Edges

Unlike a simple voxel engine where each block has little connection with its neighbor, marching cubes algorithm has its voxels and subsequently voxels' edges related to their neighbor. A voxel is connected by six edges and inversely, an edge is connected by two voxels. This property will be problematic along the borders of the six faces as each face has its own local space.

For a marching cubes algorithm in canonical space, it is typical that neigboing chunks are connected by adding edges between the chunks. This solution is likely difficult and unintuitive to implement in spherical space since, unlike in a canonical space, when in a spherical space we need to consider connecting chunks internally in a face and externally between faces.

Instead of adding new edges, this paper proposes that neighboring chunks should be connected to one another by overlapping. Two chunks could be connected to each other by overlapping each other's bordering voxels. On the same face, voxels that are repeatedly overlapped should be removed such that chunks will share the bordering voxels and relevant edges. This rule, however, should not be applied for those on the faces' borders. The reasoning is that, although they may overlap, edges might be representing different directions. Ergo, our approach pairs those overlapping voxels and their edges on the border instead. By pairing the voxels and edges at the border of faces, their values could be updated to their counterparts to synchronize each other.

4.5 Navigational Grid Generation

A navigational grid is useful for navigational algorithm such as A*. Here, it is proposed that it could done via checking for terrain features such as cliffs, overhangs, and caves. When the surface is deemed "accessible", the grid could be extended to there (Figure 10).

Before considering those "inaccessible" terrain features, there is a need to check for voxels that represent a surface of the planet. Those voxels that are on the surface of the chunk's mesh are referred to as surface voxels. A surface voxel could be found by checking whether the voxel above it is filled or not. If it is not filled, the voxel is considered as a surface voxel. That is a voxel $v^{(x, y, h)}$ is considered a surface voxel when $v^{(x, y, h)} = 1$ and $v^{(x, y, h+1)} = 0$, where (x, y) denotes the local face spatial index, h denotes the height index, v is 1 when filled and v is 0 when unfilled.

It is observed, however, that the navigational grid is not simply a collection of surface voxels. With the example in Figure 11 as a reference, it can be seen that overhangs and caves could block a path if nearby surface voxels are directly used to determine the surface area. Agents would be seen passing through the graphical mesh. Instead, it is proposed that when a surface voxel is searching for true nearby surface voxels, meaning surface voxels that could be "accessible", the algorithm should also check for those overhangs and caves. In parallel with finding the nearby surface voxels, an overhang could be found by checking if any voxel above is suddenly filled. The checking should be done iteratively, voxel by voxel, until the true surface voxel nearby is found. A cave could similarly be found as a cave is simply an extended overhang.

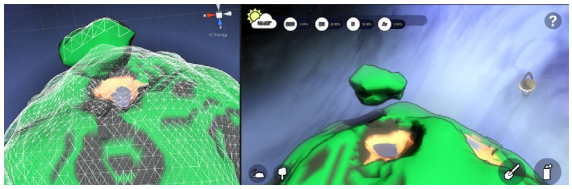

Figure 10. The result of the proposed navigational grid generating method. It could create a navigating grid when the surface is accessible and detach grids when they cannot reach each other.

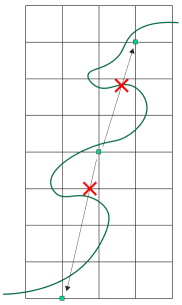

Figure 11. An illustration of an overhang, or cave blocking the path to neigbouring surface voxels at the red crosses. They are considered "inaccessible". The green line represents the terrain surface while the grid is the collection of tiles in PMC.

4.6 Terrain Procedural Generation

For a marching cubes algorithm implemented with canonical 3D axes, without the PMC algorithm, applying Perlin noise [8] to generate a planetary terrain would not be intuitive.

On the other hand, with PMC, procedural terrain generation is possible by using Perlin noise. The first intuitive method of generating the terrain could be using a 2D Perlin noise to generate a terrain for each face. Hence, there will be six different 2D Perlin noise sampling. But this requires tedious smoothing between the faces. This paper instead proposes to use fractal 3D or higher dimensions Perlin noise to generate the terrain (Fig. 12). The most simple and intuitive example is to generate the height map for the sphere, and hence the planet, using 3D Perlin noise.

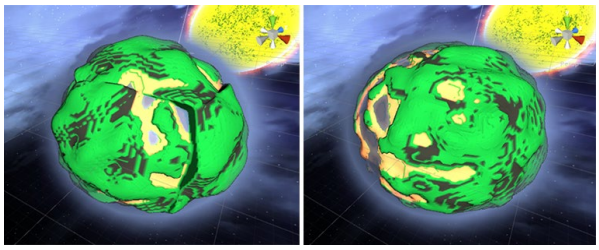

Figure 12. The difference between using 2D and 3D Perlin noise. Using 2D Perlin noise for generating terrain requires additional smoothing operation (Left). Using 3D Perlin noise to generate could bypass this need (Right).

It is worth noting that the scalar values of edges could be used for other algorithms to extract useful information for their own purposes. For example, the depth of the sea could be extracted by evaluating the scalar values of edges and the states of the voxel. This could be used for shading the shore of a sea.

5. RESULT AND DISCUSSION

Previous methods either uses heightmaps or voxels engines to generate planetary terrain. The disadvantages of the two methods are that the former could not express complex terrain features such as caves or overhangs while the latter could only generate the terrain in blocks. With PMC, it is possible to generate a planetary terrain that could express those complex terrain features while maintaining smooth surfaces as shown in Figure 13. Although the terrain allows dynamic editing, a navigational grid could also be dynamically generated in parallel (Figure 14). Lastly, 3D Perlin noise is proposed to procedurally generate the terrain. The usefulness of PMC in STEM education is shown in a previous study [9]. Therefore, it is expected that PMC could be useful for practical application such as scientific education and simulation aside from digital games.

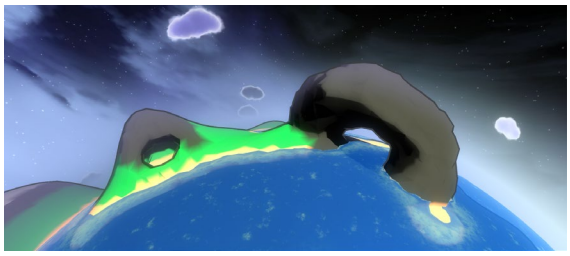

Figure 13. This figure illustrates that PMC permits the generation of complex terrain features such as caves and overhangs.

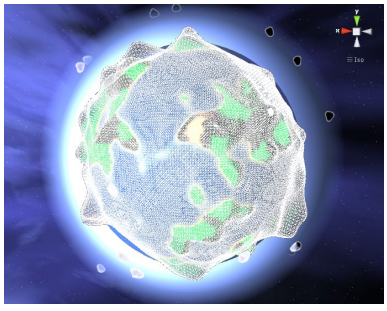

Figure 14. This figure shows the navigational grid of a PMC-generated planet.

A limitation in this research is that the PMC-driven mesh will continue to be coarser as the planet gets further from its core. A possible solution to ease this elevation-related distortion could be done by having a different n_t at different level. The new n_t should be set such that it is divisible by the previous one so that the previous tile could entirely cover the new tiles in integral number. But this solution will not truly resolve this distortion and seems to be unavoidable for spherical parametrization.

6. CONCLUSION

To generate more complex and smooth terrain features not achievable by heightmaps and voxel engines for planets, PMC, a marching cubes algorithm for spherical space is proposed. Some challenges such as the coordinate systems, navigational grid generation and procedural terrain generation have been explored.

7. REFERENCES

[1] W. E. Lorensen and H. E. Cline, "Marching Cubes: A high resolution 3D surface construction algorithm," *ACM Siggraph Computer Graphics,* vol. 21, no. 4, pp. 163-169, 1987. DOI= https://doi.org/10.1145/37402.37422

[2] R. Shu, C. Zhou and M. S. Kankanhalli, "Adaptive Marching Cubes," *The Visual Computer,* vol. 11, no. 4, pp. 202-217, 1995. DOI= https://doi.org/10.1007/BF01901516

[3] H. Muller and M. Wehle, "Visualization of Implicit Surfaces Using Adaptive Tetrahedrizations," in *Scientific Visualization Conference*, 1997.

[4] S. Fuhrmann, M. Kazhdan and M. Goesele, "Accurate Isosurface Interpolation with Hermite Data," in *International Conference on 3D Vision*, 2015. DOI=https://doi.org/10.1109/3DV.2015.36

[5] T. S. Newman and H. Yi, "A Survey of the Marching Cubes Algorithm," *Computer & Graphics,* vol. 30, no. 5, pp. 854-879, 2006. DOI= https://doi.org/10.1016/j.cag.2006.07.021

[6] G. M. Nielson, L.-Y. Zhang, K. Lee and A. Huang, "Spherical Parameterization of Marching Cubes Isosurfaces Based Upon Nearest Neighbor Coordinates," *Journal of Computer Science and Technology,* vol. 24, no. 1, pp. 30-38, 2009. DOI= https://doi.org/10.1007/s11390-009-9201-z

[7] J. Goldsmith and A. S. Jacobson, "Marching Cubes in Cylindrical and Spherical Coordinates," *Journal of Graphics Tools,* vol. 1, no. 1, pp. 21-31, 1996. DOI= https://doi.org/10.1080/10867651.1996.10487453

[8] K. Perlin, "An Image Synthesizer," *ACM Siggraph Computer Graphics,* vol. 19, no. 3, pp. 287-296, 1985. DOI= https://doi.org/10.1145/325334.325247

[9] Z. P. T. Sin, P. H. F. Ng, S. C. K. Shiu and F.-L Chung, "Planetary marching cubes for STEM sandbox game-based learning: Enhancing student interest and performance with simulation realism planet simulating sandbox," in IEEE Global Engineering Education Conference, 2017. DOI= https://doi.org/ 10.1109/EDUCON.2017.7943069