Dynamic creation of an underwater terrain according to AUV Data

Kristof Kipp
University of Bremen
Bremen, Germany
kkipp@informatik.uni-bremen.de

Nils Leusmann
University of Bremen
Bremen, Germany
leusmann@informatik.uni-bremen.de

Phillip Schneider
University of Bremen
Bremen, Germany
phisch@informatik.uni-bremen.de

***Abstract*—This document addresses the basic ideas for the deepsea landscape creation for the Travis project of the University of Bremen.**
The aim was to create a solution, that dynamicaly creates a landscape with a given heightmap.
Due to some restrictions in the Unreal Engines source code, we needed to implement a routine that uses procedural mesh components *(PMC)* to create a landscape at runtime.

I. Introduction

The Unreal Engine Editor provides the possibility to import a heightmap to create a landscape. One of the requirements on TRAVIS was, to load different mission related data on runtime. The Unreal Engine offers this specific solution only during the runtime of the editor. This means that every engine code which implements the import of a heightmap is only initialized during editor runtime. Using this code would also implay a copyright violation, because using editor code in a packeged software is not allowed by epic [1]. All in all it is not yet intended by epic to provide a heightmap import routine outside the editor. A good solution for our problem seem to be PMC. Hereinafter should be described how PMCs function and how we used them to implement our deep sea landscape.

II. How Procedural Meshes Work

A Procedural Mesh Componets in simple is a mesh of triangles which can represent any geometric object and manipulate its shape during runtime. One advantage of PMCs is, that they also can be generated during runtime. All instances of a PMC can have multiple different mesh sections. Every single section can have its own structure, texture, can be updated independently [2]. An object of this type consits of the following elements:

- a list of points (more accurate: vertices)
- a list of triangles
- two lists of normals and tangents to control lightning and visability
- a few optional lists for texture coding

The Unreal Engine takes this information and then drwas the specific geometric object: Three vertices build one triangle. By using certin vertices for multiple triangles a mesh is shaped.

In detail the constructor contains the following parameters *(all of them are arrays)*:

- SectionIndex Index of the section to create or replace.
- Vertices Vertex buffer of all vertex positions to use for this mesh section.

[1] https://forums.unrealengine.com/unreal-engine/feedback-for-epic/26495-terrain-editing-in-runtime/page11

[2] for further reading: cf. [1] , [2] and [3].

- Triangles Index buffer indicating which vertices make up each triangle. Length must be a multiple of 3.
- Normals Optional array of normal vectors for each vertex. If supplied, must be same length as Vertices array.
- UV0 Optional array of texture co-ordinates for each vertex. If supplied, must be same length as Vertices array.
- VertexColors Optional array of colors for each vertex. If supplied, must be same length as Vertices array.
- Tangents Optional array of tangent vector for each vertex. If supplied, must be same length as Vertices array.
- bCreateCollision Indicates whether collision should be created for this section. This adds significant cost.

III. How procedural mesh component is used in TRAVIS

A. Necessary steps to generate the deepsea landscape

The generation of the procedural mesh component is split in different parts. *First part* is reading and convertig the mission data. *Second part* is to put the necessary parameters into arrays[3]. In the end the unreal engine draws the mesh object. All of this individual parts are descriped in detail below.

B. First draft of the algorithem building our landscape mesh

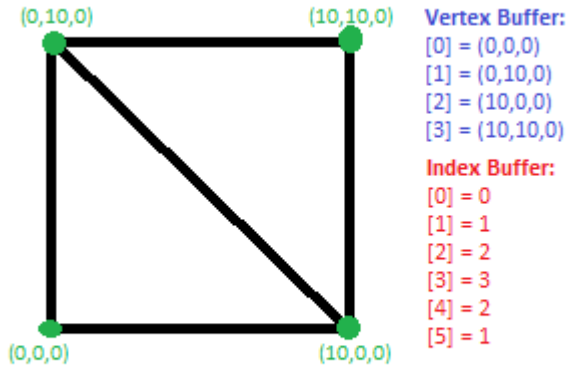
The first version of the allgorithem was very simple and is inspired by [2].

```
for (int k = 0; k < width; k++)
{
    // adds two starting points for every row
    vertices.Add(FVector(size, k*size, 0));
    vertices.Add(FVector(size, (k+1)*size, 0));

    for (int j = 1; j <= length; j+=2)
    {
        // add two additional points to generate a
 rectangle
        vertices.Add(FVector(size*j, size*k, 0));
        vertices.Add(FVector(size*j, size*(k+1), 0)
);
        // fist triangle
        Triangles.Add((j - 1)+(k*length));
        Triangles.Add(j+(k*length));
        Triangles.Add((j + 1)+(k*length));
        // second triangle
        Triangles.Add((j + 2)+(k*length));
        Triangles.Add((j + 1)+(k*length));
        Triangles.Add(j+(k*length));
    }
}
```

It just adds triangles together to form a rectangle. This way only four vertices are necessary and used by two triangles. The rectangles are then connected together to form a two-dimensionale mesh. In a nested for-loop the algorithem creates two starting points for each new row in the outer-loop. Each inner-loop then generates two new points. This four given points are then used to create two vertecis which form a rectangle.

[3]Saving and loading this parameters to guarantee userfriendly calculation time is described in the technicalreport *Save and Load Function*



This picture showes the principle which is the underlying in the algorithem above.

Adding the third dimension, some important conditions and of corse improvements form the final algorithem whis is descriped below.

IV. ASC-File

Usually the MARUM uses a specific data type, in which it saves and distribute its bathymetry data. For the purpose of our Simulation we will always receive this .asc-file. It will provide Travis with all the data needed to create an height map.

A. Structure

The simulation will only work with an .asc file and even though there is no real standard for .asc files, the MARUM assured us we will always revive an file with the same structure. For an easy readability the Marum choose to use this ascii coded file format, even though it is slower to read for computer then raw bytes. In addition to this it makes it easier to talk about the values.

The first five lines of the files are always composed of a descriptive String and the corresponding value. This meta information is followed by an huge data matrix, where each cell is value that describe the depth. The values are separated with a white space. Furthermore is it possible to compute the corresponding latitude and longitude values for each cell with the help of the given meta information. The meta values, with example values, can be seen in table I

Human readable String	Example value	Description
ncols	10123	Number of columns of the height matrix
nrows	5097	Number of columns of the height matrix
xllcenter	17.6763952465	Longitude value of the center point
yllcenter	38.2872007428	Latitude value of the center point
cellsize	0.0000100000	Step range between the Cells
nodata_value	-99999.000000	If the AUV could not get any value

TABLE I
THE META DATA OF THE ASC FILE

B. Reading

For the purpose of storing all the data inside the asc file, we created a new class. The *asctype_t* class stores all the meta data and the height matrix. The height matrix points will be stored as GameAnchorPoints which can be roughly described as Poitns inside the Unreal Engine. For more information about GameAnchorPoints see our paper about the Aquarium. After the reading process is finished we want the *asctype_t* to have all the data in a way that we can immediately work with it. Since the asc file only provides us with a latitude and longitude value we need to convert these values into GameAnchorPoints. For this we need to use the aquarium again. In addition to that we only know the latitude and longitude values of the middle point of the matrix. Because we know the stepsize between each cell we can compute the exact position of each value. This will need to be done before we will give the values to the aquarium, because the aquarium will only convert an 3d vector. We know for sure that there will always be an middle point inside the matrix because the numbers of columns and rows will always be odd. While starting to read the file, it is not certain how big the data matrix will become. This can only be said, after the first two lines of the file have been read. Then we know how large the Matrix needs to be. After we obtained this knowledge we will resize the matrix accordingly. The *resize()* function will only work, if we have stored the column and row values of the class.

To increase the performance we will read one line and hand it over to an worker thread. In the worker thread we will do the necessary computations. We will compute the latitude and longitude values and from there use the aquarium to gain GameAnchorPoints. Because of the thread pool library we are using, it is only possible to give a thread one parameter. For the thread to work properly it needs more information. This means we needed to create an new data object which will be passed to each thread. The *ascline* consist of the variables which can be seen in table II.

Human readable String	Description
int idx	The row in which the data will be stored (needs to be -6)
std::string line	The actual data from the line
ascdata_t* _data	A pointer to the dataobject in which the data will be stored
AquariumHelper* _helper	The aquarium in which the points should be converted

TABLE II
THE MEMBERVARIABLES OF THE ASCLINE

Because there is a specific *ascline* for each row we can let each thread write into the real *ascdata_t* object without worrying to overwrite an data.

V. Conclusion

For our usecase the Procedural Mesh Component

VI. Further Reading and Sources

https://wiki.unrealengine.com/Procedural_Mesh_Component_in_C%2B%2B:Getting_Started https://forums.unrealengine.com/unreal-engine/feedback-for-epic/26495-terrain-editing-in-runtime/page11

VII. Appendix

A. Cutted first lines of the .asc File

```
ncols 10123
nrows 5097
xllcenter 17.6763952465
yllcenter 38.2872007428
cellsize 0.0000100000
nodata_value -99999.000000
-99999.00 -99999.00 -99999.00 -99999.00 -99999.00
    [...]
-99999.00 -99999.00 -99999.00 -99999.00 -99999.00
    [...]
[...]
```


Acknowledgment

Thanks to M.Sc. Christoph Schrder, Dr. Ren Weller and M.Sc. Philipp Dittmann for their support ,patience and endurance.

References

[1] "Procedural Mesh Component in C++:Getting Started," https://wiki.unrealengine.com/Procedural_Mesh_Component_in_C%2B%2B:Getting_Started

[2] Chris Conway, "Mesh Basics," https://github.com/Koderz/RuntimeMeshComponent/wiki/Mesh-Basics

[3] Chris Conway, "Basic Concepts of the RMC," https://github.com/Koderz/RuntimeMeshComponent/wiki/Basic-Concepts-of-the-RMC