Human motion on terrain

# Introduction

The objective of this assignment is to display a humanoid character walking on some rough terrain. In order to achieve this, the problem has to be broken down into two parts. The first part is the terrain itself. Loading a terrain as a very fine mesh is going to occupy most of available memory resources. It is unlikely to store that amount of information regardless if it is going to be processed in an optimal way since all of the processing time is going to be spent just rendering the terrain. Therefore, leaving no space for other necessary processes to be completed under the preferable time limits of ~33.3 and ~16.6 milliseconds.

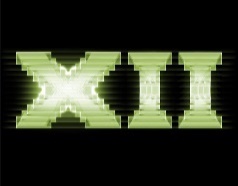
Taking that into account, this problem can be solved by applying techniques that take a subset of points and then interpolate between those points to produce features that correspond to the sort of ones that are expected to be found in terrains of such nature such as a hill or mountain range. That then offsets some of the need to store information that is able to go through techniques that produce those dynamically as they are required. Mathematical algorithms can be used to simply generate terrain as needed thus meaning that large amount of data does not necessarily have to be stored as mainly the routines and the production rules that enable the generation of terrain only need to be stored.

By observing an image of a real mountain, different materials including snow lines can be distinguished thus proving that after a given altitude the snow appears. Hence, setting certain thresholds will produce a more realistic look depending on the type of terrain. Additionally, In the foreground, the resolution is very apparent, so the techniques used to generate and render those features are going to be more costly and require more detail than those in the distance. For this reason, LOD (level of detail) methods have been developed to divide the number of triangles generated on the mesh depending on the camera distance.

The second part includes a model of a biped and its walking animation on terrain. First and foremost, the 3D model needs to be properly loaded in and rendered. In order to produce an animation, there are a number of techniques each with their own advantages and disadvantages. There is blending animation, 3D transformation animation and skeletal animation which is going the chosen method. The main disadvantage of blending animation is that it is very expensive especially on memory. For instance, if an animation is 1 megabyte, then 20 animations will be over 20 megabytes making it very memory costly. Furthermore, there is no solution for level of detail whereas skeletal animations blend quite well with level of detail. For example, if there was a character in the distance then there would be a need to exist multiple copies of the mesh at lower level of detail and blending between higher level and lower level would produce a problem of finding which vertices were meant to be paired together. Also, blending between multiple animations will not be easily achieved. It will require a lot of effort from the animator and the programmer to accomplish satisfactory results. Therefore, blending animations may be possible but are limited and not preferred to develop a whole game from it. The next option would be relevant to animating rigid bodies, 3D transformation animation can be used to detach the mesh from the actual animation. The animation would be stored in a series of matrices or quaternions to represent the position of each component of the model. Effectively storing the components in some sort of hierarchy. The advantage is that the memory of an animation is in the range of bytes. Additionally, as the mesh is not attached to the animation, it is possible to have lower levels of detail of the mesh thus having a low polygon version. If a model is in the distance, the animation will not change, it will just play a different version of the mesh. However, computer games are not restricted to mechanical arms, so skeletal animation is the solution to produce complex but lightweight animations. Skeletal animations use bones, which are what the animation represents. An animation, which is a series of positions for each bone and lastly skinning. Skinning is where the bone manipulates the actual mesh. Nevertheless, In order to implement the right animation method, there will need to graphics engine to develop the application and render the scene.

# choosing an application programmable interface

With the recent release of several new APIs, picking the right API to start building a 2D or 3D graphics engine is not always so apparent predominantly when some of the APIs are especially designed for expert users. Learning a single API requires a serious undertaking as it is more explicit, bringing the programmer closer to the hardware. Modern APIs better match the way new graphics hardware works resulting in more effective and less error-prone drivers. Nevertheless, they do not make engine programming trouble-free as more responsibilities are assigned to engine developers for handling essential resources such as memory allocation. In an attempt to find the right API that satisfies the criteria of the application that is going to be developed, a description of the chosen and alternate API has been written below:

 Microsoft’s proprietary graphics API is DirectX with the latest instalment being DirectX12. DirectX12 is an expert’s API. It was designed for programmers that had already become skilled with developing complex engines with DirectX11 and now require even more low-level control of the graphics pipeline to correspond to increased performance demands. As the API has removed many of the built-in functions that DirectX11 had before to take advantage of the parallelisation of an efficient graphics workflow. Nonetheless, it also means that basic operations will require large chunks of code while DirectX11 only need one. For this reason, DirectX12 was not chosen.

DirectX11 is the chosen API as current leading industry standard for graphics APIs. It is supported by a large community of programmers so it will be easier to find websites and forums that explain concepts and help others to solve specific problems that might occur during development. Microsoft released DirectX11.3 alongside DirectX12 acknowledging that the majority of developers would desire the same added technical features but not require the same low-level control that is provided by DirectX12.

# application architecture

The framework from semester 1 was used and expanded to fit the requirements for this semester’s assignment. Hence, only the features implemented for this application will be explained. The concepts and algorithms implemented for this version of the engine are as follows:

## 3D Triangle Grid

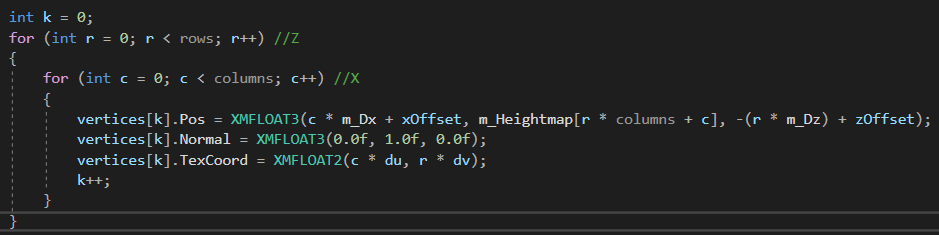


Figure : Implementation of a triangle grid in C++

A terrain can be generated by loading in a heightmap texture and applying the height values to the vertices’ y component. A nested for loop iterates through the 2D array used for generating the 3D triangle grid terrain. For the x and z values, the size of a cell is multiplied with each iteration plus an offset at the end for separation. The Z component requires the multiplication to be negative in order for the indices to render the triangles looking up the Y vector so it can be viewed, otherwise the terrain is rendered upside down. Lastly, the texture coordinates are calculated by simply dividing each element with columns - 1 and rows - 1 respectively resulting to an accurately mapped texture as demonstrated bellow:

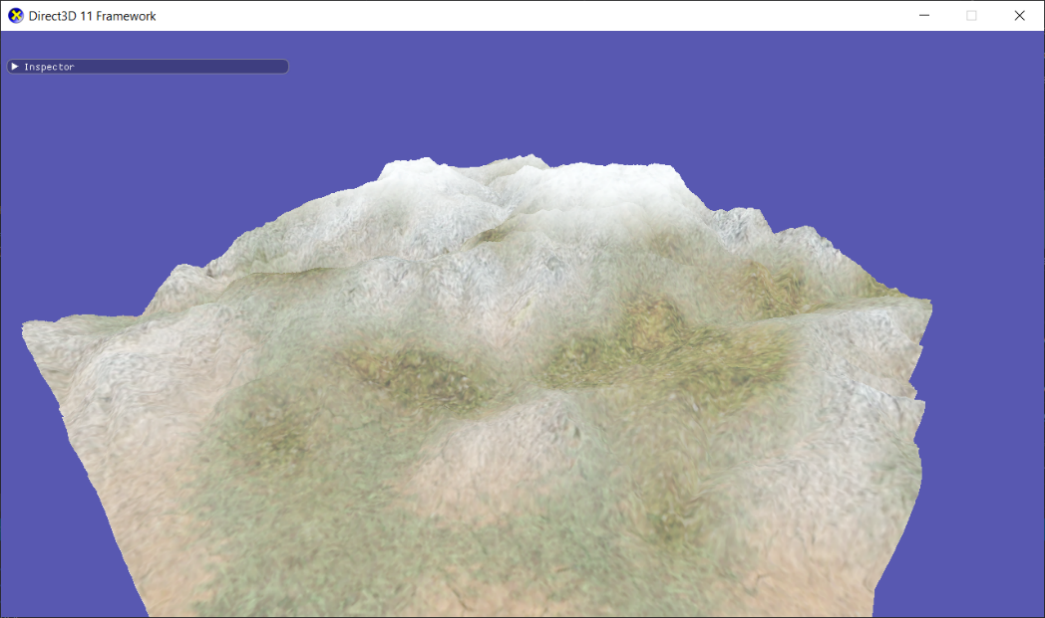
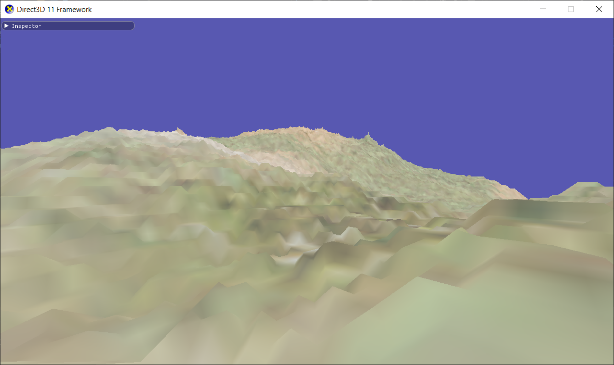
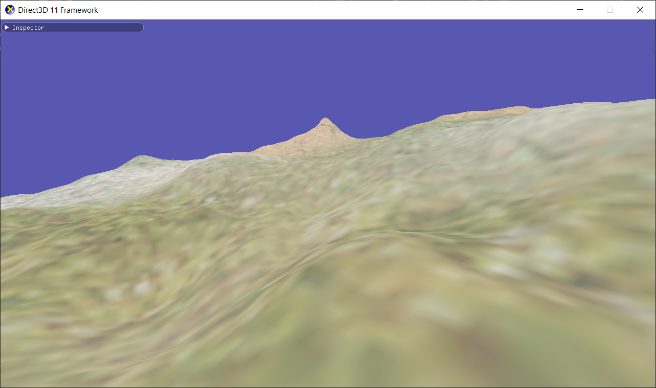


Figure : Textured terrain constructed with a loaded heightmap

## Diamond-square

A procedural terrain generation algorithm was also implemented to generate a heightmap using the diamond square algorithm. Providing the algorithm with 4 random initial corner values of a 2D array will result in filling the inner values with midpoint displacement. It is worth noting that the algorithm requires with width and height of the 2D array to be 2n+1. For instance, the width and height that was used to test was 2x2 at minimum and 513x513 at maximum i.e. 20+1 and 28+1. This formula is important as the numbers it outputs possess an essential property for calculating their midpoints. The first step of the algorithm is called the diamond because it takes the average of the four corners subsequently giving back the value at the centre. Then, the next step is called the square as the midpoint of each of the edges is calculated by solving the average of its corner points. The function applies recursively the rest of the values to the rest of the points and adds a random displacement to each one for variety.

A function was also added to smooth the spikes that were occurring throughout the terrain. The method used for smoothing out sudden spikes from some vertices is called average or mean filtering and the main purpose is to reduce intensity variation between vertices on the terrain. The way thus the function works is that it creates a copy of the generated heightmap and then checks all its 8 neighbouring vertices and store their height consequently taking the average of all and applying them back to the original heightmap.

# Terrain class

The terrain class contains all the necessary fields and functions for generating a terrain. GenerateMesh() function will produce the necessary vertices, indices and buffers for rendering the terrain. LoadHeightMap() will read the values from a binary heightmap file and then store them in a member variable before applying them to the vertices. Diamond-square algorithm is made using three functions. DiamondSquare() will iterate through the vertices and call SquareStep() and DiamondStep() where accordingly. SmoothHeights() is called after DiamondSquare() as a heightmap needs to be generated first to smooth out the spikes that might have existed in the initial state of the heightmap.

# Critical appraisal and performance of the software

Importing ImGui was one of the best decisions that benefited development. It really helped with debugging and editing the scene at runtime thus making the application look more like an engine now that an editor is in place.

A noticeable performance bottle neck is when standing on one corner of the terrain and looking over the other side of the terrain thus having to render millions of triangles simultaneously. Resulting to substantial performance overhead. Two techniques that would benefit performance greatly are frustum culling and Level of detail. Frustum culling will stop complete models for being sent to the GPU if they are not viewed by the camera. Level of detail techniques would benefit the engine as it would render in detail object that are closer to the camera. In conclusion, combining those two techniques would provide instantly noticeable performance boost.

# references

Medium. 2020. Diamond-Square Algorithm Explanation and C++ Implementation. [ONLINE] Available at: <https://medium.com/@nickobrien/diamond-square-algorithm-explanation-and-c-implementation-5efa891e486f> [Accessed 29 April 2020].

learn.64bitdragon. 2020. The Diamond Square Algorithm. [ONLINE] Available at: <https://learn.64bitdragon.com/articles/computer-science/procedural-generation/the-diamond-square-algorithm>. [Accessed 29 April 2020].

<https://stackoverflow.com/questions/19913596/c-2d-array-to-1d-array>

wccftech. 2020. Microsoft DirectX 11.3 Revealed – The Feature Set of DX12 Minus the CPU Overhead Reduction. [ONLINE] Available at: <https://wccftech.com/directx-11-3-api/>. [Accessed 29 April 2020].

Gamedev. 2020. Box Filtering Height Maps for Smooth Rolling Hills. [ONLINE] Available at: <https://www.gamedev.net/reference/articles/article2164.asp>. [Accessed 30 April 2020].