Advanced Graphics and real time rendering: report

Brandon Stafferton - 20010806

Table of Contents

[Literature Review 2](#_Toc133944449)

[Static Terrain Generation 2](#_Toc133944450)

[Grid 2](#_Toc133944451)

[Height Map 3](#_Toc133944452)

[Tessellation 4](#_Toc133944453)

[Basic Tessellation 4](#_Toc133944454)

[LOD 5](#_Toc133944455)

[Procedural Terrain Generation 7](#_Toc133944456)

[Diamond Square Algorithm 7](#_Toc133944457)

[Fault Line Algorithm 9](#_Toc133944458)

[Perlin Noise 11](#_Toc133944459)

[Skeletal Animation 13](#_Toc133944460)

[Mesh 13](#_Toc133944461)

[Model 14](#_Toc133944462)

[Bone 17](#_Toc133944463)

[Animation 19](#_Toc133944464)

[Animator 20](#_Toc133944465)

[Skinning 22](#_Toc133944466)

[Blending 23](#_Toc133944467)

[Graphic APIs 24](#_Toc133944468)

[Bibliography 25](#_Toc133944469)

# Literature Review

## Static Terrain Generation

### Grid

To start terrain, a grid of quads, which were made of two triangles, was made which its height and width can be changed. A few variables must be calculated first such as the rows, columns, total cells, total faces, total vertices and texture coordinates as shown below:

Text

Description automatically generated

A vector of a struct Vertex is then made to gold the total amount of vertices, the struct hold vertex information of position, normals and texture coordinates. After that, a double for loop is made looping over the number of rows and columns and creating and x, y, z position of the triangle’s vertices and is also done for the indices and a vertex and index buffer is also created as shown:

Text

Description automatically generated

The results are shown below:

A picture containing outdoor, fish, ray

Description automatically generated

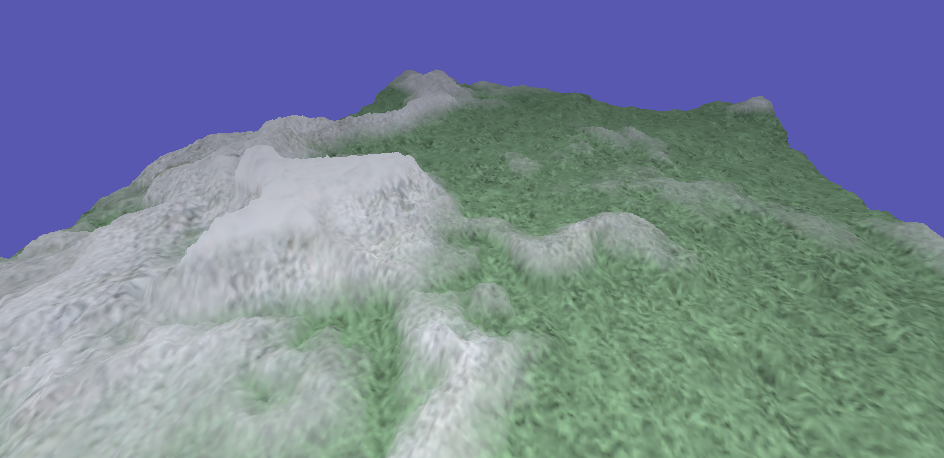
### Height Map

A height map is an image that store values for height for each pixel within the image. The height map was created by passing in a height map image to the shader through the domain shader (or the pixel shader without a domain shader) and sampling it with the texture coordinates from the vertex shader. The terrain height is then multiplied by the height map height values and added onto the world position. This is shown below:

Text

Description automatically generated

The result of this is also shown below:



## Tessellation

### Basic Tessellation

Basic tessellation was done by creating a hull shader and a domain shader. The first step was to create a hull shader and start with the “Control Point Phase” which created the control points as well as the tessellation factor. A function called “HSMAIN” was made to take in an input from the vertex shader into the hull shader where each output to equalled to an input patch which loops through the control points to create triangles as shown below:

Text

Description automatically generated

From the control point phase, it is then moved to the “Patch Constant Phase” which takes in a tessellation factor and setting the edges and insides of each triangle to that tessellation factor which then return an output as shown below:

Text

Description automatically generated

After the hull shader comes the domain shader, this is where the new vertex manipulation is done such as calculating the world position, normals, world, view and projection space using barycentric coordinates. The barycentric coordinates are multiplied with each triangle patch point for the world position, texture coordinates and normals to be interpolated as shown below:

Text

Description automatically generated

The result of tessellation is shown below:

A picture containing purple, nature

Description automatically generated

### LOD

LOD is done in the “Patch Constant Phase” and is done by inputting two positions from the input patch. It is first done by calculating the edge mid-point as shown below:



Then a radius is created around the control points by calculating distance as shown below:



The view space is then calculated by taking the view space and multiplying it by the mid-point.



Then we take this view space and project it into clip space:



After we project into clip space, we then get the normalized device coordinates:

Text

Description automatically generated

Lastly, the normalized device coordinates are converted to screen space and clamped between the lowest tessellation factor and the maximum tessellation factor:

Text

Description automatically generated with medium confidence

The result is shown below where the further away the terrain is from the camera the less tessellation factor is taken and there for less detail is given as a result. However, LOD has a problem where cracks appear between the quads that are drawn, therefore, BiNs must be used to merge the control points from the triangles together to fix this issue.

A picture containing invertebrate, coral, coelenterate

Description automatically generated

## Procedural Terrain Generation

### Diamond Square Algorithm

The diamond square algorithm is the first procedural algorithm done. This was done in two steps: the diamond step and the square step. The diamond step looped through the terrain height and width divided by the side length and creates a centre for the terrain by using the 4 corners of a diamond shape midpoint displacement which this is done by filling a 2D array as shown below:

Text

Description automatically generated

This is then followed by the square step after all available diamonds have been founded which also loops over the terrain height and width. This then does the same like the diamond step but finds 4 corners of a squad and finds the midpoint displacement as shown below:

Text

Description automatically generated

Text

Description automatically generated

This is the result of the diamond square algorithm:

A picture containing outdoor, nature

Description automatically generated

### Fault Line Algorithm

The fault line algorithm is an algorithm that increases or decreases vertices depending on a side of a vector. It’s started by creating a for loop of several inputted iterations. Within this for loop, it calculates a random row and a random column on the grid. It then finds a random point for x, y and z of the point. A random normal vector then is generated and is tested against vertices on the left and right side of this line by checking for the dot product creating angle. If its more than 90 degrees, then it raises the left side and if its less than 90 degree it lowers the right side which manipulates the y valve of each vertex on that side as shown below:

Text

Description automatically generated

The result is shown below:

A picture containing icon

Description automatically generated

### Perlin Noise

The last procedural terrain generation that was added to the application was Perlin Noise which used the lib noise library to generate noise maps. Lib noise generator generated noise by adding modules to create a variety of different noise to combine to together which included a Perlin Noise, Billow and Ridged Noise modules. The modules were made as shown below, each module created noise by setting frequency, scale and bias and then combined by using a selector module. This then was finally followed by a turbulence module to add more variety to the noise.

Text

Description automatically generated

These modules then created a height map which then can be passed over to the shader and used the same way as the previous height map as shown below:

Text

Description automatically generated

The result of the Perlin Noise modules resulted in this:

A picture containing grass

Description automatically generated

## Skeletal Animation

### Mesh

To start skeletal animation, the application needs to load in a model with its indices, vertices texture coordinates and bones. A mesh class was created to store this data when the model will be loaded in the next step. A “Setup Mesh” function was made to setup the vertex and index buffers as well as the sampler states as shown below:

Text, chat or text message

Description automatically generated

The mesh class also contained a function called “Draw” to draw the indices and vertices as well as to set the vertex and index buffer of each model that will be loaded. This function also passes the sampler state to the pixel shader as well as setting the primitive topology to a triangle list. This is shown below:

Text

Description automatically generated

### Model

The Assimp model loading library was used to load any type of model in any format to which the data from this model was placed into its own variables into a model class as well as using the mesh class to draw the vertex and index data. Using the function below reads the file using an Assimp scene which leads to the “ProcessNode” function which loads the model data into its own variables.

Text

Description automatically generated

The “ProcessNode” function grabs all the meshes within the scene that Assimp has loaded and places them into a vector to be ready for extracting the data from the model.

Text

Description automatically generated

The “ProcessMesh” function is a function that returns a Mesh, using the mesh class created earlier. This function takes the information from the mesh loaded from Assimp into its own data structures of a Vertex and Texture which is then made into a vector. As you can see below, the meshes vertices, indices, normals and textures coordinates are loaded into XMFLOATs and pushed back into a vector which is then return as a Mesh.

Text

Description automatically generated



The Model class also extracts the bone data into its own variables. This is done by using Assimp to loop through the number of bones within the model to get the bones names as well as the bones information. The bones information is stored within a structure which contains a bone ID as well as a 4x4 matrix for the offset. A map variable is map from the standard library to find each bone name and then set a new bone information structure to that bone. This means when a bone is found it is given an ID as well as an offset by using Assimp to get each bones offset matrix, and this is shown below:

Text

Description automatically generated

One important part of this to note is that DirectX 11 does its matrices in row major and therefore a function called “ConvertMatrixDirectXFormat” was made to convert a matrix from column major to row major.

Lastly, each bones weights are then extracted from the mesh loaded from Assimp into its own variables which includes the weights and the number of weights. A for loop is then used loop through the number of weights and each weights vertex ID as well as weight is extracted from the model as shown below:

Text

Description automatically generated

### Bone

Now that the bone information has been extracted from the model, it now must be used by creating key frames of positions, rotations and scale ready for animation by being made into a struct as shown below:

Text

Description automatically generated

Each of these positions, rotations and scale needs to be interpolated between itself and a timestamp. Each animation has key frames to which are extracted from an animation by extracting the node of each key frame, these key frames are extracted into its own variables using Assimp. Using the key frames structures and key frame nodes, they can be interpolated and then updated to move each bones position, rotation and scale. It is important to note that interpolating rotation is done by using quaternions and this is stored into a 4x4 matrix as the same with positions and scale. The calculation of each variable is shown below:

Graphical user interface, text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated

The extraction of the data using Assimp is shown below:

Text

Description automatically generated

### Animation

Now that model has been loaded, bone data as well as mesh data has been extracted, it can now be used for animation as well as skinning. An animation class was created to read in an animation file to extract data such as the duration and the number of ticks per second of the animation as shown below:

A screenshot of a computer

Description automatically generated

This is followed by reading in the bone hierarchy data as well as reading in any missing bones by creating the following functions: “ReadHeirarchyData” and “ReadMissingBones”. ReadHeirarchyData reads the hierarchy of the bone data structure. It loops through the number of children within the bone structure extracts data into a structure called “AssimpNodeData” which contains a 4x4 matrix for the transformation, a string for the name of the bone, integer for the children count and a vector of the AssimpNodeData to store the children of bone hierarchy as shown below:

Text

Description automatically generated

Text

Description automatically generated

Lastly, reading in the missing bones for the animation is the last step. A function is made to loop through each key frame of the animation and pushes back a new bone into a vector which is then mapped into a new bone information map to read the bones that are engaged within an animation related to their current keyframe as shown below:

Text

Description automatically generated

### Animator

Animator is the last class to be made to animate a character using the bone data. This class gets the current time of each current animation and updates the bone transformation by using the offset of each bone and the parent transformation of each bone to create an identity matrix and therefore a function called “CalculateBoneTransform” was made for this as shown below:

Text

Description automatically generated

This function starts by getting each bones start and end transformation effectors which is stored in an XMMATRIX called “nodeTransform” which has the position, orientation and scale of each bone as shown.



Next, a check is done in an if statement to see if a bone exists within the current animation structure and proceeds to call the update function and gets the current transform of that bone in the current key frame.

Text

Description automatically generated

After that, matrix called “globalTransformation” is made that using the current node transform and multiplied by the current bones parent transform to create a transform matrix.



After that, a XMMATRIX is then made for the offset of the current bone in the animation key frame which is multiplied with the global transformation matrix to create an identity matrix. This is then done for every child in the bone structure for the animation duration as shown below.

A screenshot of a computer

Description automatically generated with medium confidence

The identity matrices made from each bone in the animation is stored in a vector of matrices called “FinalBoneMatrices” which will be passed through the vertex shader for skinning.

### Skinning

Skinning is done by manipulating the vertices of model to make the skin of model move with the bones when moving the key frames of an animation. A maximum of 4 bone IDs and bone weights are passed through the vertex shader as shown below:

A screenshot of a computer

Description automatically generated with medium confidence

Next, the final bone matrices that was made in the animator are passed to the shader from a constant buffer, the final bone matrices are a vector and has a maximum number of bones it can hold, this can depend on the model as each model has a different number of bones. The more bones the more memory it uses, therefore, a maximum of 100 bones is set to pass through.

Text

Description automatically generated

Now for the manipulation of the weights and using the weights to change the vertices position within the vertex shader. Firstly, a loop is made to loop over the maximum number of weights and within this for loop a float is made called “localPosition” which multiplies the bone matrices by the vertex input position. The bone weights inputted from the vertex shader are then multiplied by the local position and then outputted into the pixel shader. There are two if statements that also exist. The first one checks whether a bone is present or not, if there is then it continues to skin the model. The second if statement check if there are more bones than the maximum amount of bone and if there is it sets it to an identity matrix.

Text

Description automatically generated

### Blending

Blending animations is the last step to creating smooth animations which is done within the animator class and modifies the “CalculateBoneTransform” function as well as a function to update the blended animations called “BlendAnimation”. The new “CalculateBoneTransform” function will now take in two animations to blend between, one starting animation and an animation to be layered on top, the animation current time for the current animation and the layered animation time to blend to as well as a blend factor which must be between 0 and 1. The function will also check for the bones within the layered animation like the original function as shown below:

Text

Description automatically generated

To blend the two animations, a new blending matrix will need to be created which blends the rotation of the two animations together. This is done by getting the quaternion rotation from the two animations and then using slerp to interpolate them together. Lastly, the new blended matrix needs to consider the percentage difference between the blend factor or else the transformation will break. This is shown below:

Text

Description automatically generated

Now the bone transformations are now blended, the animation needs to be updated with speed multipliers to transition from one animation to another by getting the first animations duration and dividing by the layered animations duration and then multiplying by the blend factor to allow a smooth transition. This is shown below:

Text

Description automatically generated

## Graphic APIs

The two main graphic APIs that are used for this is DirectX 11 and OpenGL. The biggest difference between DirectX 11 and OpenGL were the math libraries. DirectX 11 has a big difference when it comes to using quaternions as they are setup differently as you have to use XMVECTORS and use DirectX 11 functions to change it to a quaternion and then into a matrix whereas OpenGL has quaternion variables to use from the start and can be easily changed to a matrix using “quat\_cast” which are all from the glm maths library.

Another difference between these two APIs is that DirectX 11 uses row major maths for matrices which changes how you view the maths behind matrix manipulation, as compared to OpenGL, it uses column major matrices. Therefore, using DirectX 11, the process of changing the matrices to row major was needed and made it a bit more complex as new function was needed to flip the values around.

# Bibliography