Scene Overview

# Objects and Techniques

The scene for my coursework application is set in a World War One battlefield, with each object contributing to this theme. The centrepiece of the scene is the terrain object, utilizing the most shader stages and demonstrating the most techniques of all objects in the scene. The terrain starts as a plane, dynamically tessellated with vertex manipulation applied to take on the appearance of a hilly battlefield. This result is then passed through the geometry shader to have blades of grass generated at every tessellated point. Next is the explosion object, starting as a sphere it also has vertex manipulation applied to it, however this time all three channels are used in combination with a time factor resulting in the explosion animating over its lifetime. Each vertex is then passed to the fragment shader and textured to be a different colour based on its progression through the explosion. Zeppelin and biplane models are also used in the scene to help demonstrate the full lighting and shadow capabilities of the scene, one spot light is attached to the zeppelin pointing down towards the terrain, and a point light is created in the explosion class animating along with the explosion mesh. Both these lights have full shadowing capabilities. Once the first pass of the scene is complete, a depth of field post process effect is then applied – focusing the camera’s perspective on whatever it is looking at by blurring out everything else.

# Coursework Brief Response

The scene includes at least one example of all five techniques listed in the coursework specification, with some techniques having more than one implementation. The main challenge in creating this coursework was applying all techniques to work together, mostly all stages in the terrain object.

* Vertex Manipulation
  + A displacement map is used to determine the height of the terrain’s vertices
  + The explosion object uses a multi-channel displacement map to animate over time
  + Both are correctly lit and textured – the explosion mesh, being a light source, has no lighting applied to it but has its output colour at full intensity
* Post processing
  + A depth of field effect is attained by rendering the whole scene to a render texture on its first pass, blurring this texture, and then using both render textures to determine the output texture in the depth of field pass.
* Lighting and Shadows
  + The spot light of the zeppelin and the point light of the explosion are both used to light the scene, with full shadow maps used for both. I chose to implement a point light in this way as it appeared to be the most challenging type of light to implement.
* Tessellation
  + The terrain object is constructed from a predefined number of patches, each patch is individually tessellated based on its distance to the camera. Once tessellation is applied, the terrain then has its normal recalculated to allow for more accurate lighting.
* Geometry shader
  + The tessellated surface of the terrain is then passed to the geometry shader, which reconstructs the terrain faces, and generates three blades of grass for each tessellated point. These blades of grass are then animated to give a swaying effect. All geometry in this stage has correct lighting applied to it and is then textured differently in the fragment shader depending on if the geometry is a blade of grass or section of terrain.

# Controls

* Pause explosion
* Move zeppelin
* Jump to explosion
* Max tessellation value
* Shadow Demonstration[multiple]

Extra:

* Explosion made its own class, so each explosion object could handle its own light, view matrices, and shadow maps etc
* Also made so that individual explosions can be recycled as required, reducing number of lights in scene

Algorithms and Data Structures

* An in-depth explanation and justification (based on complexity and/or hardware architecture) of the algorithms and data structures used in the scene
* Important calculations used, data passed and shader stages
* This should focus on the hlsl/shaders written
* Providing diagrams, code snippets and supporting screenshots as required

# Explosion

A separate class was created for explosions so all relevant information for the lighting and mesh itself could be kept in one place, as well as some helper functions to keep main App1.cpp file tidier.

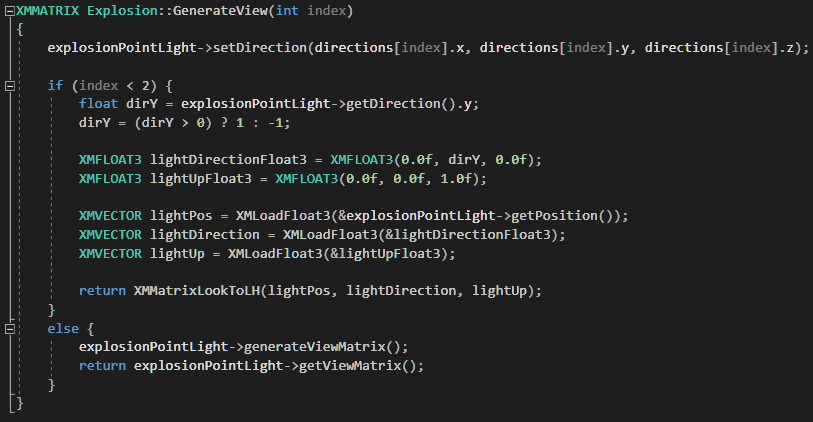
One of the most important functions in the explosion class is the *GenerateView* function. This is responsible for returning a view matrix based on one of the six possible directions of the point light. Most of the time this is as simple as setting the direction of the light appropriately and then calling the point light’s own *generateViewMatrix* function, however calling that function with a view direction that is either exactly up or down will cause the program to crash. My initial solution to this was to pass in a very small number into either the X or Z parameter of the light direction which fixed the crashing issue, however this also resulted in the light having ‘gaps’ between the vertical direction and three of the four horizontal directions. To negate this the *GenerateView* function instead takes in absolute vertical directions and manually calculates the view matrix instead of using the light’s inbuilt function, setting the up vector to be the positive Z direction.

Figure 1 - Explosion Generate View Function

The explosion shader itself uses a vertex shader and pixel shader. The vertex shader is passed a multi-channel compose texture and displacement information calculated with a sin functions over time for animation. The combination of both results in the individual extrusion of the vertices from their original position. A compose texture is used in place of sending three separate displacement maps giving the advantage of keeping code tidier but more importantly it cuts the amount of data sent to the GPU to a third, without losing any information and therefore reducing one of the main bottlenecks to performance.

Figure 2 - Explosion Compose Texture

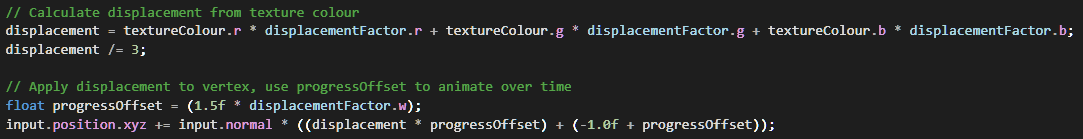


Figure 3 - Vertex Displacement Calculation

These vertices are then passed to the pixel shader, in which their colour is calculated from a ramp texture based on their displacement from their original position – the further from the sphere mesh a vertex is, the further along the ramp texture is sampled, starting with a bright almost white explosive colour, then shifting to a darker smoke colour. Vertices may also be clipped at this stage, with the range of clipped vertices expanding over time – this gives the impression of the explosion dissipating, starting with the furthest out vertices and then reaching its way quickly to the centre of the sphere.

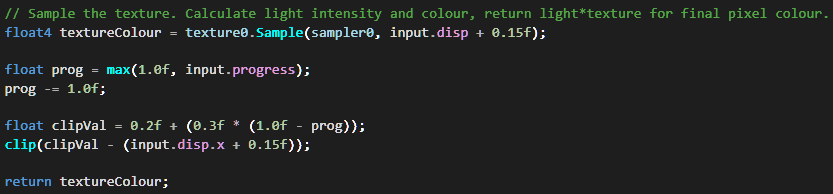


Figure 4 - Pixel Shader Main Function

# Terrain

The terrain shader is by far the largest of all shaders in the program, using every stage of the graphics pipeline covered in this module – though the vertex shader is only used for passing information forward. As mentioned before, the main challenge creating this shader was getting all stages to work together however each of the four main stages also had their own challenges and unique purpose in creating the end terrain object, as detailed below.

Terrain Hull Shader

The main purpose of this stage, as with most hull shaders, is to prepare the patches in the plane for tessellation. Since the level of detail is dynamically controlled, the factor the patch is tessellated by was calculated based on distance. Initially my approach was to simply divide the maximum tessellation factor by the distance calculated, however this resulted in a far too steep decrease in detail, resulting in one patch of highly detailed terrain next to a low detail patch not far from the camera’s view. To solve this, the factor is instead calculated logarithmically, using the current distance as a fraction of the maximum distance of the plane as input. This results in a much more gradual drop off in detail, and at high maximum tessellation values the shift in detail close to the camera is almost unnoticeable.

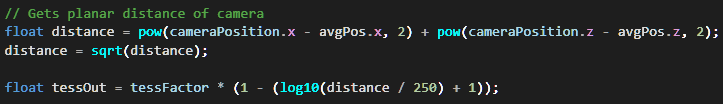


Figure 5 - Tessellation Factor Calculation

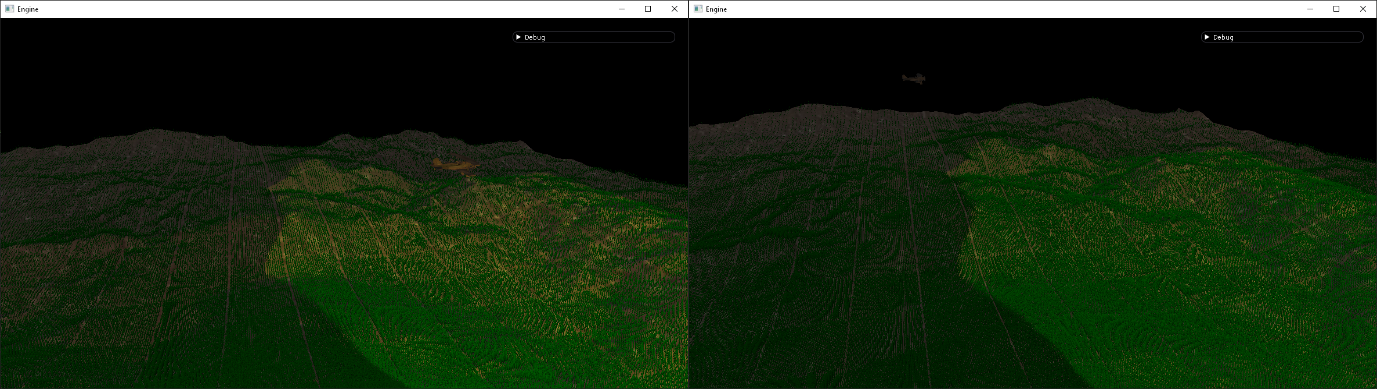


Figure 6 - Division by Distance(L) vs Logarithmic(R)

Terrain Domain Shader

The domain shader has two main purposes; calculating the new vertices’ positions from the original patch positions then modifying these positions based on the displacement map used and recalculating the normals of these vertices to remain correctly lit after the displacement map has been applied. The vertex manipulation is a similar, simpler version of the same method used in the explosion vertex shader – one channel is used instead of three, vertex positions are only moved on one axis, and no animation is involved. The main challenge comes when performing the previously mentioned normal recalculation. My initial approach to this was to calculate face normals in the geometry shader, then apply this normal to each of the vertices of the face. This meant losing a lot of detail when it came to applying lighting to the terrain, and so I instead opted to calculate normals as part of the domain shader based on an estimation of the surrounding vertices.

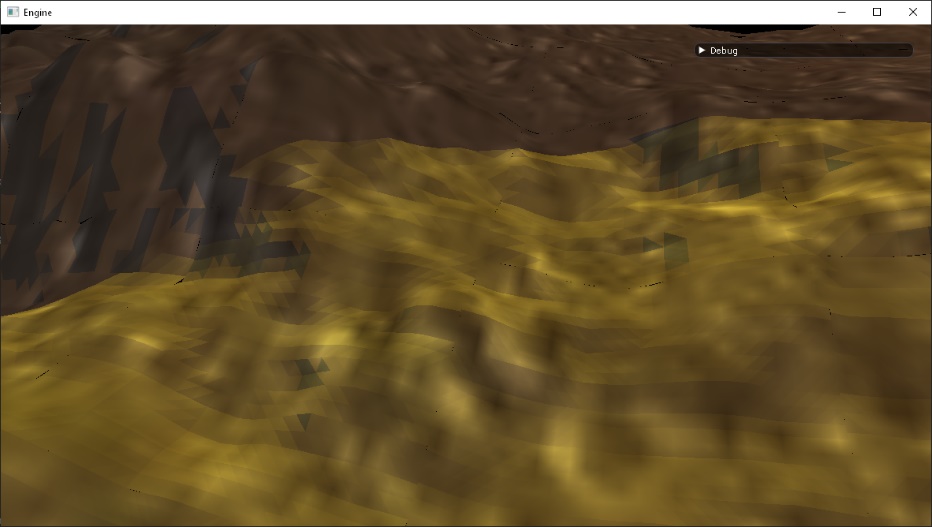


Figure 7 - Per Face Lighting

When implementing this I attempted two different methods; calculating the face normals of the four surrounding faces to the vertex, then combining and averaging these to give the vertex normal required and calculating the cross product between the two vectors that passed through the terrain vertex from adjacent vertices in the X and Z direction.

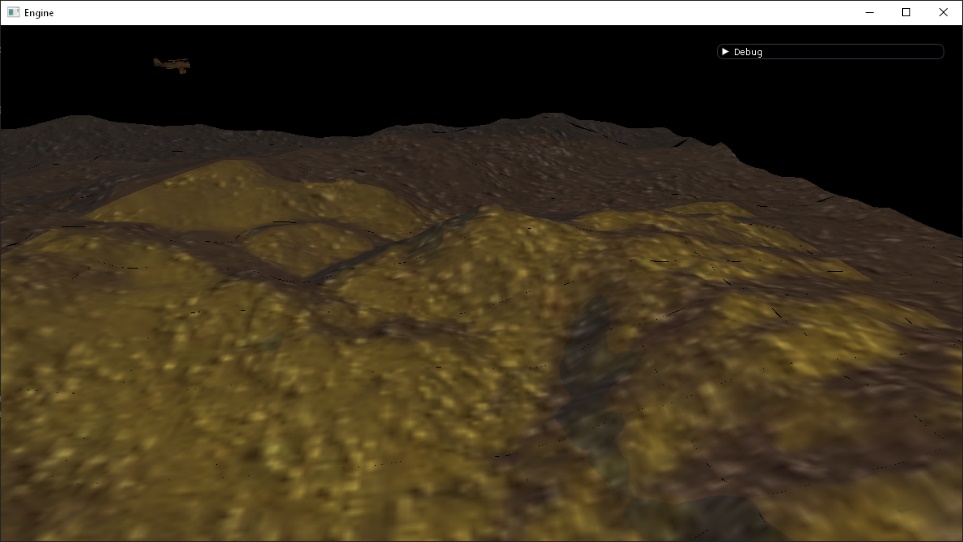
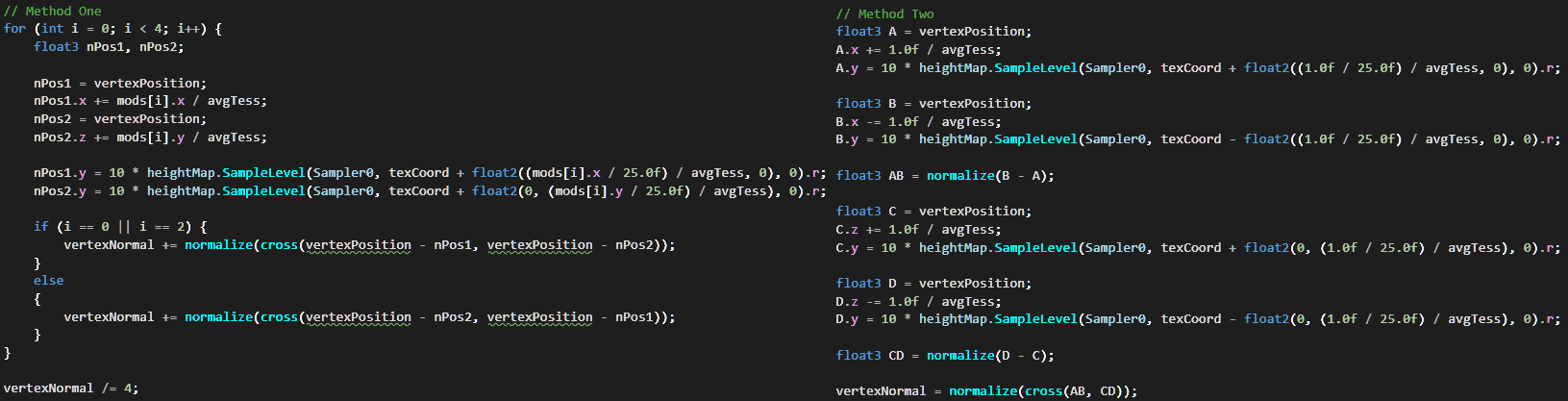
Comparing both methods side by side there was no noticeable difference in each result, so I chose to go with the cross product between two vectors method to avoid the branching used in my implementation of calculating adjacent face normal, hopefully improving the GPU performance here by not interrupting the lock-step execution. The result of calculating normals in this way gave much more accurate compared to the previous method, with no visible triangles or faces with lighting applied.

Figure 8 - Adjacent Face Normal (L) vs Cross Product Between Two Vectors (R)

Figure 9 - Per Vertex Lighting, with matched tessellation value of previous example

Terrain Geometry Shader

The first step of the geometry shader is to reconnect the vertices of the terrain and output the resulting triangle to the triangle strip. This is also where the final output parameters of each vertex are set to then be passed to the pixel shader. The texture coordinates and normals are just passed on from being calculated in the domain shader, as well as the position which is first converted to clip space. The vertices position from the perspective of the camera are also calculated here for both the spot light and the point light. The next step is to generate three blades of grass for each tessellated point – each vertex of a blade is calculated using the forward, right, and up vector from the tessellated point’s position, with the forward vector being calculated as the vector between the camera’s position and the point’s position. This has a billboarding effect, so the blades of grass are always within view of the camera. Another approach to achieve this would be to set the rasterizer description to not cull the back faces for the blades of grass. All the same information is then calculated and passed on for each blade of grass as for the terrain. This allows for blades of grass to be partially lit if the bottom vertices are obscured by an object while the top is not, or vice versa.

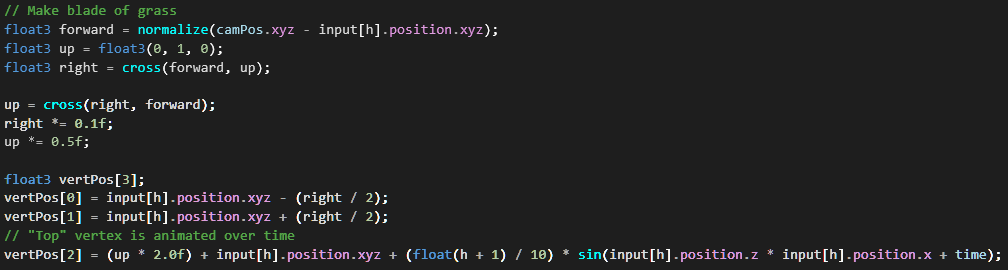


Figure 10 - Grass verex positions calculation

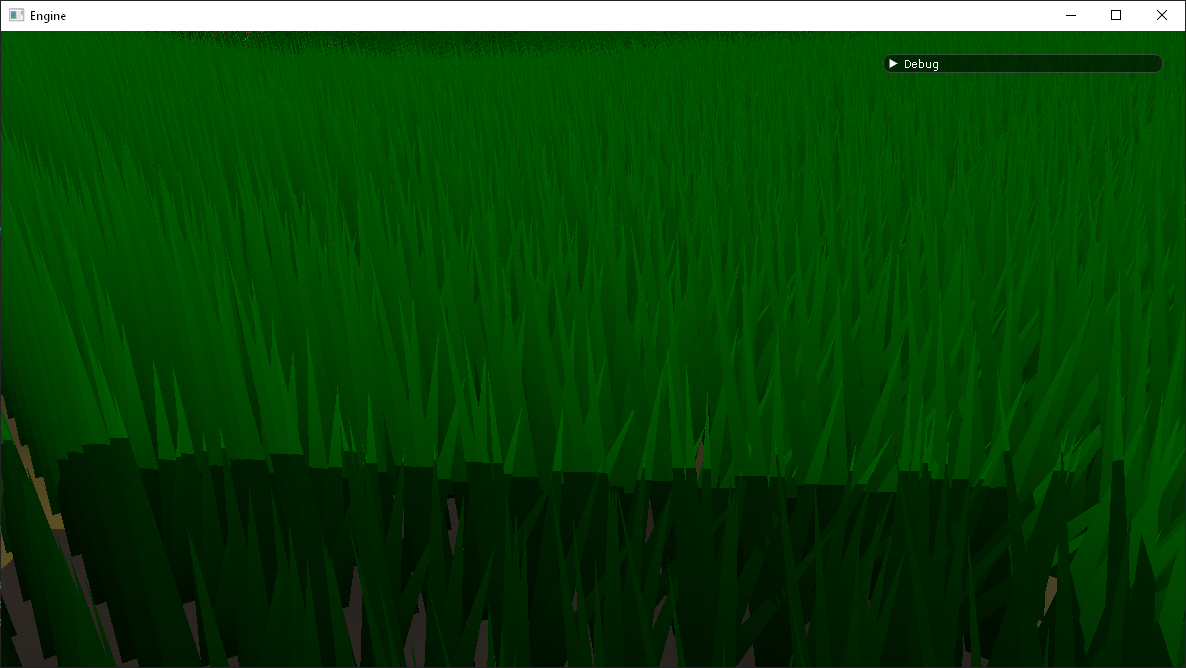


Figure 11 - Partially lit, billboarded grass

Terrain Pixel Shader

Since the pixel shader at this stage is passed two different types of geometry, its first job is to simply decide which texture to sample based on the geometry it is receiving. To inform this decision, I used the last component of the world position output from the geometry shader to indicate if the vertex was created as a section of terrain, or a blade of grass – allowing a different texture for each. After this, the final light colour for the fragment is calculated, for which the light buffer passes in the ambient colour, diffuse colour, direction, and position of each light. All components except the ambient colour is required for every type of light to have full lighting and shadow capabilities however I pass in everything, as well as using a float4 for every component, to ensure correct buffer alignment and to make expanding the amount of lights handled much easier. For the point light of the explosions, six passes are done for each direction of the light to calculate the lighting of a fragment – with a separate shadow map for each pass. This could be optimised to just three passes since a vertex will only ever be in view of three directions of a light at any one time. The spot light only has one pass for its one direction, however, to attain the “cone” effect I wanted for this light, a distance check is done from the centre of the shadow map to the view position of the vertex in place of the usual “bounds” check to make sure the vertex is within view of the full light source.

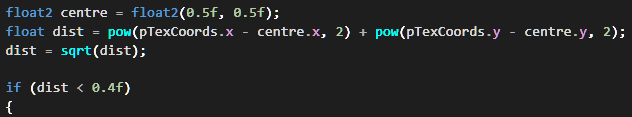


Figure 12 - Spot light distance check

# Depth of Field

Before the depth of field post process effect is applied to the scene, a blurred version of the scene is required as input. For this I expanded on the separable filter gaussian blur example from week eight, firstly by combining both the horizontal and vertical blur pixel shaders into one shader, taking an extra input field to determine which blur is being applied. The second change I made was to automate the weighting value of the neighbouring pixels sampled. To do this, the remainder of subtracting the centre pixel’s weighting from one is halved (to account for both directions from the centre pixel). This remainder is then used when looping through the neighbour weightings, with each neighbour taking half of the remainder as its value which is then subtracted from the remainder itself. Once the loop is complete, the remainder is added onto the centre weighting to ensure the original brightness of the pixel is conserved.

Figure 13 - Blur weighting calculation

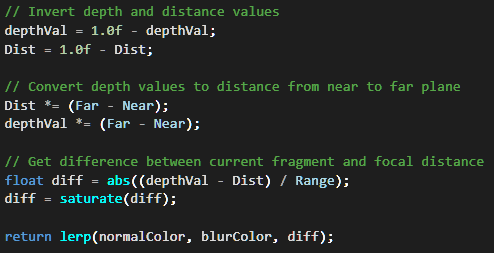
Once the both blur passes are finished, the resulting render texture is then passed into the depth of field shader. Here, the colour for the current fragment is sampled from both the unaltered scene view, and the blurred scene view. The difference between the camera’s depth value for the current fragment and the depth at the centre of the camera depth texture is calculated (and divided by a given range to specify how far around that plane stays in focus), which is then used to calculate the output colour based on a lerp between the two input textures.

Figure 14 - Depth of Field Pixel Shader

Critical Reflection

* All areas/shaders could be improved in one way or another (**screenshot all problems**)
  + Go back over week 7 proposal, note what has changed and why
    - Using depth of field rather than edge detection for post processing
      * More relevant since scene can be quite “busy” with the amount of geometry shown (especially grass) and can help player focus on what they’re looking at
* Explosion
  + If viewed closely cracks appear when displacement is applied
  + Would be solved by using an ico sphere, with more even distribution of faces, or by using a compose texture more suited to a cube sphere
* Terrain
  + Cracks appear in terrain when using dynamically controlled tessellation
    - This is due to mismatched edge tessellation values on adjacent patches, where one edge may have more detail than the other ending up with different heights (diagram)
    - Could be solved by one edge “inheriting” the edge value of the other (based on which is closest to the camera/which has a higher LoD to retain detail)
      * Elaborate
    - Reference practical rendering
  + Normal recalculation better than nothing (or taking face normal) but still imperfect
    - Works well most of the time, but some cases where the normal values are slightly off – could be due to the positions used to recalculate normals being wrong, or just a limitation on how accurate a “best guess” technique can be
    - Solution would be to use normal maps instead, either generated once from the maximum resolution of the whole plane or hand drawn by a computer artist (with more ability to do so than me). Programs exist that automatically generate normal maps based on height map but not 100% reliable
  + Grass placement linked exactly to tessellated points is imperfect
    - Ends up with patchy squares of grass where you can clearly see the difference from one patch to another
      * No grass between planes
    - Also has problem of grass moving about as tessellation values change as camera moves
    - Not as much of a problem at high tessellation
      * Another reason to use logarithmic distance since new and old positions of tessellated points blend more seamlessly
    - Finding a solution to this is difficult since direct link to tessellated points ensures correct placement on terrain
      * One solution could be to do grass generation as its own pass, and giving it the origin and dimensions of the plane as well as the height map to manually place grass unlinked from terrain
        + Could end up with floating grass
  + Shadow map bias can be incorrect under certain circumstances
    - A constant value was cherry picked which gives the most consistent results for the typical angle lighting happens
    - Not always perfect since explosion positions are random
      * Could test for a larger range of angles the explosion could happen at
      * Better solution would be to adjust bias based on angle between object and light source, would mean using a different bias for each light source
  + Terrain doesn’t cast shadows on itself
    - Not immediately noticeable since lights cast from fairly high positions, and only vertices
    - Dynamic LoD means rendering depth information of terrain in same way as all other objects would be useless – would just get depth information for the flat plane
    - A number of solutions:
      * At depth pass stage, use exact same terrain shader but flip output type to not use lighting etc in the pixel shader but rather output the depth information
        + Would mean tessellating, calculating normals, generating geometry all over again
        + Essentially doing the most expensive stage all over again (with the only advantage being no lighting information needing calculated)
      * Same solution as above but separate all steps of shader so only tessellation would be done again
        + Would potentially introduce floating grass problem mentioned before when rendering terrain as normal
        + No grass depth information to be used, shadows would just be flat hills
      * Output to multiple render textures when rendering scene as normal
        + First render texture would be normal scene view with lighting etc
        + Second render texture would be depth information
        + Depth information then merged with standard depth pass texture (keeping closest depth information, discarding further depth info) and used in next scene pass
        + Would mean no extra work done tessellating etc all over again but high detail, correct depth information
        + Main draw back is that terrain depth information would be one frame behind everything else

Wouldn’t be as noticeable as drawbacks with other solutions

My personal preference

This could even be solved with deferred rendering?

* Depth of field/Blur
  + DoF effect applied immediately as player switches from looking at one object to another
    - Can be jarring when distance between both objects is large enough
      * Effect can be seen in original uncharted game
    - Solution would be to implement some sort of gradual shift in depth of field over time, with a maximum difference that can be applied each frame (or per second)
  + Range value is hard coded, meaning objects may be partially out of focus if they are big enough
    - Could be solved by dynamically adjusting range based on distance between face camera is looking at and the face behind it on the same direction vector
* Performance
  + Relatively low performance with high tessellation
  + Initial plan in proposal was to adjust amount of down sampling done based on frame rate
    - Soon discovered post process effects would not be the limiting factor in performance, but rather high tessellation and grass generation leading to a large amount of lighting calculations needed
    - Could be solved by limiting max tessellation value – often increasing beyond 32 doesn’t give meaningful enough change to justify drop in performance

References

Zepplin 3D model and texture:

<https://free3d.com/3d-model/zepplin-v2--65873.html>

Biplane 3D model and texture:

<https://free3d.com/3d-model/biplane-58447.html>

Explosion shader tutorial (also source for burn texture):

<https://stevencraeynest.wordpress.com/2013/03/29/easy-volumetric-explosion-in-unity3d/>

Depth of field tutorial:

<https://digitalerr0r.wordpress.com/2009/05/16/xna-shader-programming-tutorial-20-depth-of-field/>

Terrain displacement map:

Wikipedia page’s main image

Tessellated plane & Potential fix for terrain cracks:

Practical Rendering and Computation with Direct 3D 11 – Chapter 9­