**Fire Propagation Simulation using Geometric Cubes**

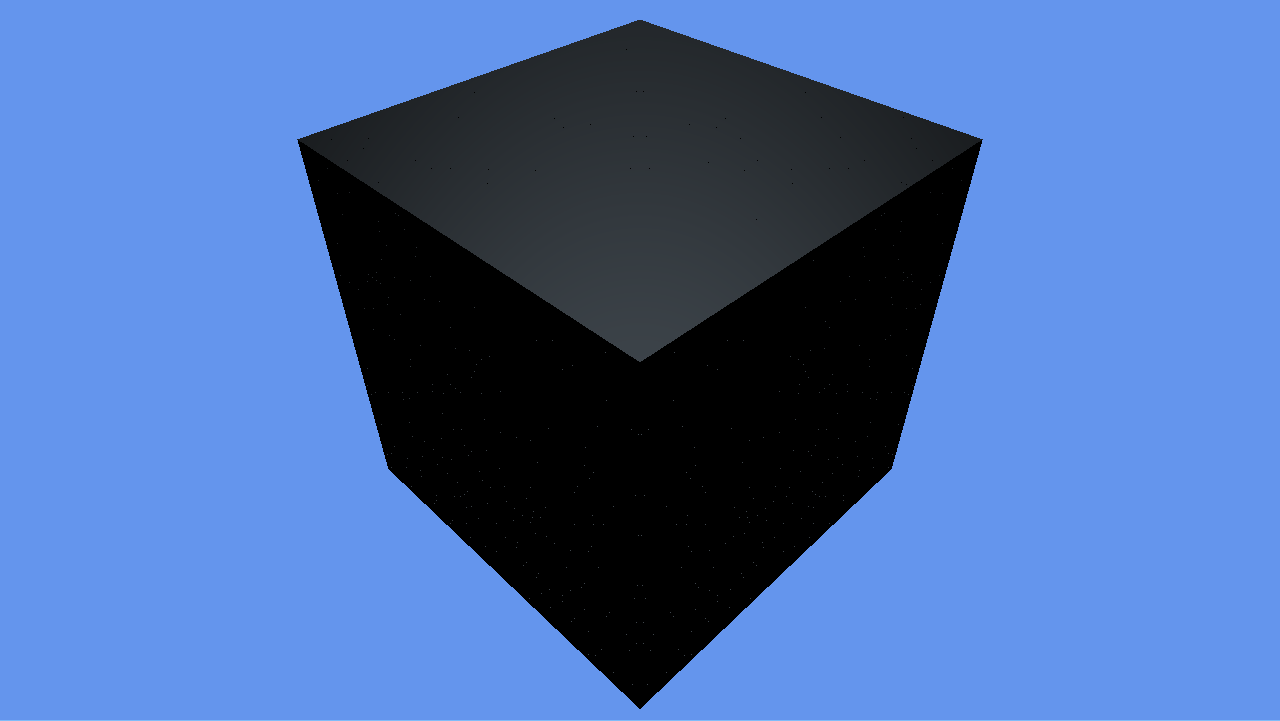
Student name: Gundars Pelns

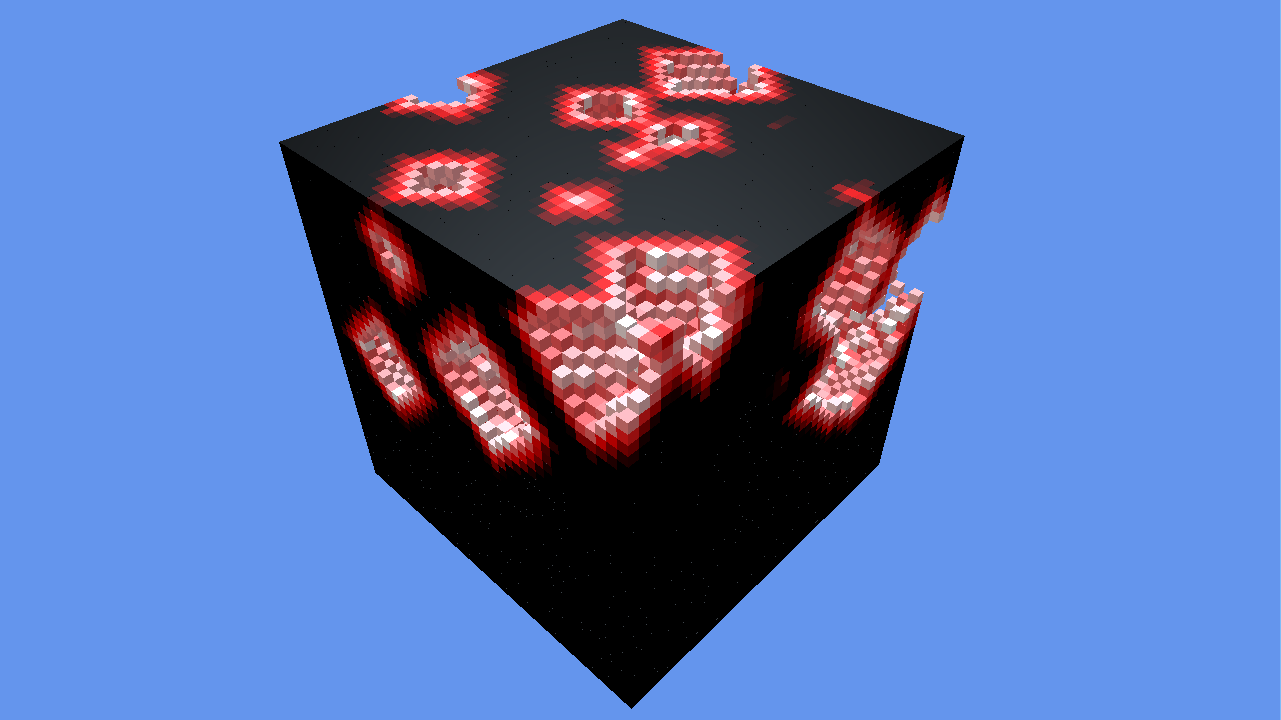
Student ID number: 15014193

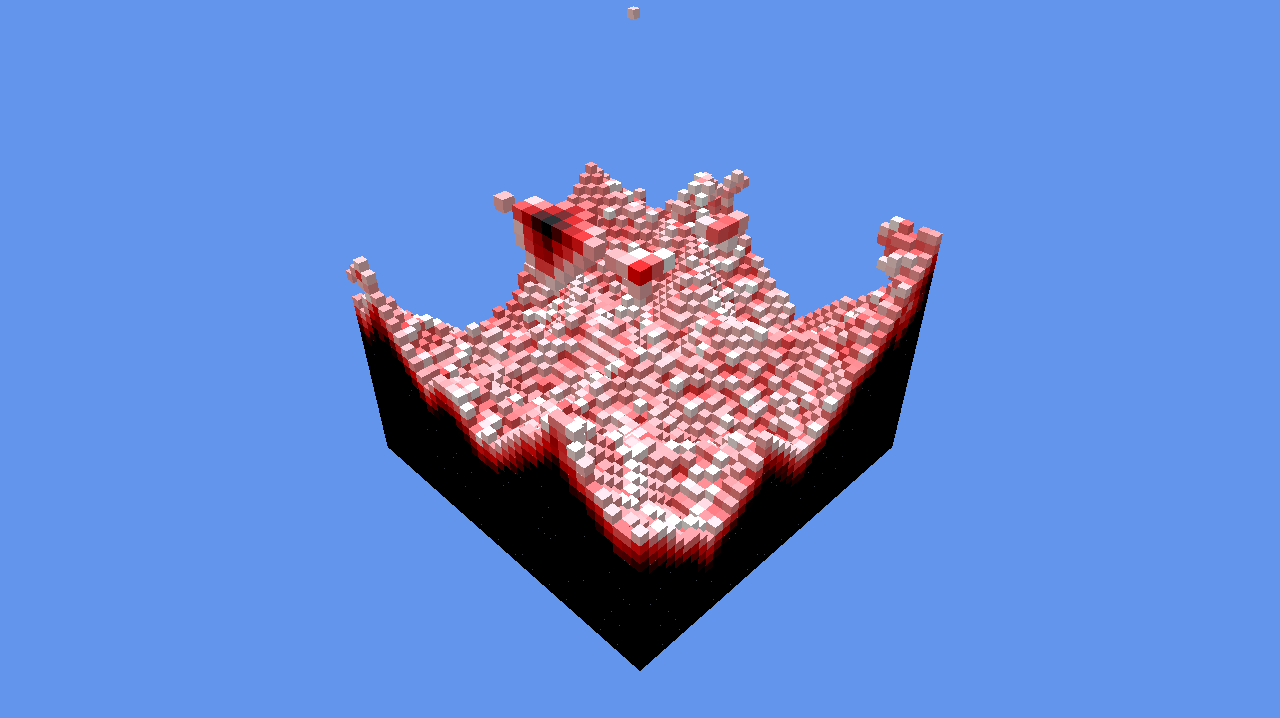
Supervisor name: Tom Bashford-Rogers

Date submitted: 26/04/2018

Module: UFCFS4-30-3 Creative Technologies Project







# **Summary**

A simulation of fire propagation over an instantiated object with a custom size, that burns and deforms as the fire spreads across the object, using voxel cubes (Geometric Cubes) as its mesh. Unlike a normal mesh however, this completely fills the object with burnable cubes, which use fire propagation calculations to spread across the whole object.

# **Biography**

The belief is that this project allowed further development in knowledge about the DirectX 11 engine, as well as the implementation of custom tools such as the DXTK. This project truly thought that logic and code optamazation created more of a intrest than graphical implementation, thus further development will be made within this focus.

# **Links to Project & Video**

**Project:**

**Video:**

**Introduction**

The project was developed to create a fire propagation system which could be applied to any DirectX 11 project that uses the DXTK. This is the general tool made for the DirectX engine to give developers a much more simplistic way of rendering simple objects, such as the ‘Geometric Cube’. This does also include some mathematical functions that can assist the project, however this was not necessary as all of the equations used within this project were custom made. The aim of the project was to allow the user to create a large cube object that is made from smaller geometric cubes. At a certain point it would then ignite the object, spreading heat across the geometric cubes within the object. This would then continue until the whole object has burnt and nothing is left of it. The amount of cubes on each face is altered through changing the width and height of the desired object.

The inspiration came from a game called ‘Minecraft’, in this game almost everything within its world is made out of geometric cubes. And as you can imagine, some of these cubes are able to burn and ignite other burnable cubes around it. Thus the idea was to recreate this simulation within the DirectX engine. The purpose of the project was to make this system slightly more complicated, where instead of changing the cubes texture it would rather actually slowly burn chunks off the main cube, causing the object to deform until it is completely burnt. The reason for this was to see how it could be implemented within a DirectX atmosphere, as well as to see what difference this could make to the game itself. This project would allow others that use cubic shapes within their projects that they would like to be burnable with the functionality of deforming.

The project would need to allow the user to alter its quality, this is to let the user create a custom burnable object to their desire. As the project will increase in functionality and quality if bigger values are implemented, but lose its performance as this can be very heavy on the RAM. Through development it was discovered that a 60x60x60 cube caused a massive performance issues, but this was because this created 216,000 cubes that had to be rendered each frame. They also have to check what’s happening around them, as they need to trigger if a cube around it has reached the burning point. However this would be an incredibly high quality object, especially within a ‘Minecraft’ environment, as the cubes within it are a single 1x1x1 cubes.

# **Practice**

The initial proposal was to develop a real-time realistic fire simulation, it was then discovered however that shaders in the DirectX engine are quite difficult. As DirectX 11 uses HLSL as its main graphics language, which is truly a great shader language. There has been no previous experience within this field, which did not stop development, instead it caused some changes to the project. These being that the project would more focus on the physical aspect of fire propagation rather than the visual graphical appearance. Throughout further research, a lot of unique functions were discovered, these functions were all to do with fire propagation. However they weren’t simple functions by any means, as they were designed to calculate not only the rate of the fire spreading but all other features that a burning object might have. These included mass, fuel, heat, radius of heat, and possibly difference in material as that could potentially have an effect on the age and mass of the burnable. These values were very useful to keep in mind, as they can form a reasonably realistic fire simulation if used correctly. Therefor it was important to research some techniques or equations others had made that involve these features or similar ones in their simulations. However this appeared to be quite a difficult task as the equations changed depending on how the simulation was set up, this caused some issues for a while as it required some figuring out of the functionality of a fire propagating over a series of objects using the mentioned values.

Further in development it was discovered that not all of these values are needed, as they were there to make the simulation more complex, which in theory does create a more realistic simulation. However visually this did not change much, so it was decided upon that this simulation would keep its propagation simple for now as there were still some issues with the object itself. As you may know, there isn’t a simple way of where you are able to instance objects the way you can in Unity for example. This meant that research had to be made on instancing within the C++ environment DirectX is set in. Quite a few examples were found, however this one forum on ‘StackOverflow’ explains how you are able to create instances using object classes. This is a regular class which holds the information of an object you would like to instance. The reason for it requiring to be within a class it mostly to make it more OOP which improves the general quality of the code. But the main reason for this is to allow the developer to simply create new instances of a desired object by simply calling its initialize function. In our project this would be a single geometric cube, as they will be the instances that form the burnable object itself. Therefor this information was then used to create a vector that stored pointers to the geometric cube class. This however did not work as planned, as it was only rendering and updating one instance out of the 100 that were initialized.

At first this caused quite a panic, simply because when debugging, it was not showing any errors or even null pointers as they were all storing the correct information. Since there was not much on this available online, a professor from UWE (University of the West of England) was asked for advice and possible help for this issue. The professor Simon Scarle who is an expert in Computational Biology and Chemistry (BSc, PhD), suggested a very interesting point when analyzing the vector of pointers. He suggested that the pointers might be pointing to the same exact class, rather than creating a pointer with a copy of that class. And he was absolutely correct, as the vector was set to use ‘.assign’ this meant that the same pointer was being ‘assigned’ to all of the reserved slots in the vector. So then small research was required to see what other possibilities were available for vectors when using pointers to a class. Thankfully these have been around for years, there for a source was found, which contained all of the possible calls a vector can call. So instead of using the ‘.assign’ call, ‘.push\_back’ was used instead. The difference in this is that every time you ‘push\_back’ a value onto the vector it will create a new and unique pointer to the value or class in this case. So you can see that this is exactly what we required, but not only that, when using ‘push\_back’ you do not reserve a set amount of space on the vector. As it will add an element at the end of the vector each time it is called.

Slight adjustment to the project itself was then necessary, as at this point you were able to see a single side of the cube, and this was due to the way the camera was set up. Once figuring out the functionality of the view matrix for the camera, it was reasonably easy to then alter the position and view of the camera to fit more of the object within its view. Although this was acceptable and usable, due to the fact that you could technically see every part of the object, this did not keep in mind the possibility of changing the size of the object. Which at some points could get too big for the camera to see, thus a decision was made to implement camera movement into its functionality. This was not a part of the proposed project, however it feels like this would allow the user running the simulation to have a bit more control of what they see, even allowing them to move behind or inside the object. At first it seemed to be quite a difficult task as it required changes in matrixes that were quite important to the camera and the project itself. However this proved wrong as most of the variables were already set up to allow the camera to actually render what was on the scene, and due to the fact that DXTK was implemented. Getting keyboard input was as easy as a function call, the only thing now was to adjust the values accordingly. As it was using vectors and matrixes it required to use the DirectX math library to adjust these values, which made things much nicer as it had quite complex equation functions. And all you had to do was send the correct value type to then return your desired outcome.

As you can see by looking at the code snippets to the left, these being the math functions used to implement and adjust our camera. The first two snippets are quite similar, and you can most likely understand what they are doing by simply reading their function names. Where ‘XMLoadFloat4’ returns a vector from loading a float4 value, whereas ‘XMStoreFloat4x4’ takes a matrix and applies it to a float4x4 variable.

As you can see by looking at the code snippets to the left, these being the math functions used to implement and adjust our camera. The first two snippets are quite similar, and you can most likely understand what they are doing by simply reading their function names. Where ‘XMLoadFloat4’ returns a vector from loading a float4 value, whereas ‘XMStoreFloat4x4’ takes a matrix and applies it to a float4x4 variable.













Further on you can see a function call ‘XMVector4Transform’, this was used for the movement of the camera as it allows you to transform a 4D vector by a matrix, thus changing the global position of the camera. ‘XMMatrixLookToLH’ was one of the more difficult functions, it was also one of the final calculation functions, meaning that it required an updated ‘cPos(Position), cTarg(Focal Point), cUp(Up Direction)’ to build a view matrix for a left-handed coordinate system. This would then be passed to the game to adjust the global view matrix. As you’ve noticed we have adjusted the position, however the rotation has still not been calculated. This is when we use the ‘XMMatrixRotationAxis’ to build a matrix that will rotate around an arbitrary axis, which will be the up direction of the camera, we then need to normalize these values by using ‘XMVector3Normalize’.

# **Outcomes**

The project allows a user to create a custom, box shaped, object. That then slowly burns its vortex cubes, which refer to the smaller cubes within the larger cube. The main question is, could this project then be of any use to other developers or even other projects? Although the project was initially aimed to be more of a graphics simulation, the mechanics of it seemed more intriguing and could still apply of use to others. The way it could do so, could potentially be a Minecraft type game, which utilizes these large chunks of burnable blocks. As each of these blocks have unique properties, you could simply create new IDs for specific parts of the created burnable object. For example you could say that after a certain layer, or even each layer as the object goes down. The cubes gain more mass (health), this would cause the lower layers to burn a lot more slowly than the rest. Or simply having unique IDs such as wood or ice, as they both have properties of deforming (when burnt or melted) but would have different functionality when heat is applied.

Another feature that this project has, but cannot be seen in release, is being able to create a wireframe version of the burnable object. This makes the object appear see through, allowing the developer to see exactly which cube is set alight. This is not only useful for seeing inside the object however, remember it was mentioned that each of these vortex cubes have their own unique variables including their position. This means that you could even use this vortex cube as an environmental feature, for example it could be created into a 100x100x1 (Width, Length, Height) plane which is located on top of a town, or even a forest. You could then use the information stored within a single vortex cube to alter functionality of the objects within this cube. So a simple use for this could be environmental fire, this would currently be quite simplistic and would require additional features and behaviors. Nevertheless it would still currently work, as it can take the heat level of a certain area and apply it to the environment. Possibly causing plants to die or as the heat grows more fire is spawned within that area.

As stated before, the project had quite a big change occur through the research process. This being the big change from the graphical side of a fire, to more of a technical project. This was not only due to the fact that not much was discovered about DirectX and its shaders, but mostly due to sheer interest about physics and functionality as such. However a big factor of this decision was because of the lack of information throughout the web, it wasn’t empty keep in mind. Rather it had very broad information (barely any on some papers) about it, with the information involving A-Level physics with fluid mechanics and such. Therefor instead of having to panic about shader code not doing anything, it was decided to create a deformable mesh using geometric primitives.

# **Conclusion**

The project began quite nicely, as I decided to keep contact with my professor throughout the start of the research period, it was possible to keep track on required research. However further in researching about shaders and how HLSL code works, I decided to keep my focus on logic, thus creating a fire propagation simulation to a degradable vortex cube. At first there were quite a lot of issues, one of the major ones being the direction of the fire spreading. The aim of the project here was to allow a user to simply implement a width and a height (length is calculated within the code), and give them the opportunity to then set this object on fire. Since this meant that values were constantly changing, we had to create more robust functions, which could work from those two values and not cause fire to spread across faces or cause any out of scope issues.

This project could progress in a few ways, however the most intriguing would be the optimization of this simulation. As there is a slight issue of FPS loss when a cuboid that has more than 250k cubes is spawned, this also causes the fire to spread slower however the more you wait the faster it will get ass all those cubes burn off. Or even not just taking the project itself further, but the logic of it. As logic can be used globally when developing games, an example of this in a different engine could be a room fire simulation. Where the room is filled with these vortex cubes, all connected together, with random objects inside them. These vortex cubes would obviously be only visible during debugging, using the wireframe to see the outlines, whereas in game they would work in the background. In this simulation these cubes would hold that specific areas heat level causing other behaviors to occur, such as causing a fire particle system to spawn around that area, possibly gaining size as that areas heat level rises.

# **References**

<https://www.3dgep.com/understanding-the-view-matrix/>

<https://www.braynzarsoft.net/viewtutorial/q16390-transformations-and-world-view-projection-space-matrices>

<https://gamedev.stackexchange.com/questions/152654/moving-camera-in-3d-directx12>

<https://github.com/Microsoft/DirectXTK/wiki/GeometricPrimitive>

<https://stackoverflow.com/questions/22206044/difference-between-object-and-instance-c?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa>

# **Appendixes**