COMP477
Fall 2019 Project

Mark-Zackary Rodrigues-Gillich
Concordia University
mark_rod@encs.concordia.ca

Erik Smith
Concordia University
e_ith@encs.concordia.ca

Cynthia Cherrier
Concordia University
c_herrie@encs.concordia.ca

Abstract

This report details the documentation and design of a graphics and animation project which attempted to simulate a piece of cloth material being burned away by a fire particle system. The following summarizes the different components of the project, such as technologies used, the development process, methodologies, as well as the final visual result rendered in our application.

Contents

1 Introduction

1.1 Motivation

Cloth is often present in video games and other digital media when we need to model clothing for humanoid characters or other objects made of fabric. Our team was interested in finding how other forces from its environment can interact with a simulated cloth. The focus of the simulation was to examine the possible effects caused by fire. This included, how the cloth could change color, how it could move or stretch due to forces applied, and lastly, how it could be destroyed. This project demonstrates the effect of fire, modelled as a particle system, on a small rectangle piece of cloth.

The idea of our project could have many different applications. In a video game where a character is dressed in cloth, it would enhance the realism if the different systems (the cloth and the fire) interact with each other. Clothes that are made of destructible material could help illustrate to a player how much damage the player has taken from its surroundings. The use of a spreading fire or any other type of particle system could be useful in not only video games, but any application which involves a particle system moving its source from one point to another or instantiating multiple particle systems over time.

1.2 Background

Since the focus of our project was not the implementation of the cloth system itself, we began development opting to not build one from scratch. To understand such a system, we studied the work of Dayeol Lee [4], who implemented a cloth simulation using compute shader (OpenGL), SSBO, Mass-Spring model, Verlet Integration, Phong shading, normal mapping and the libraries GLUT and GLEW.

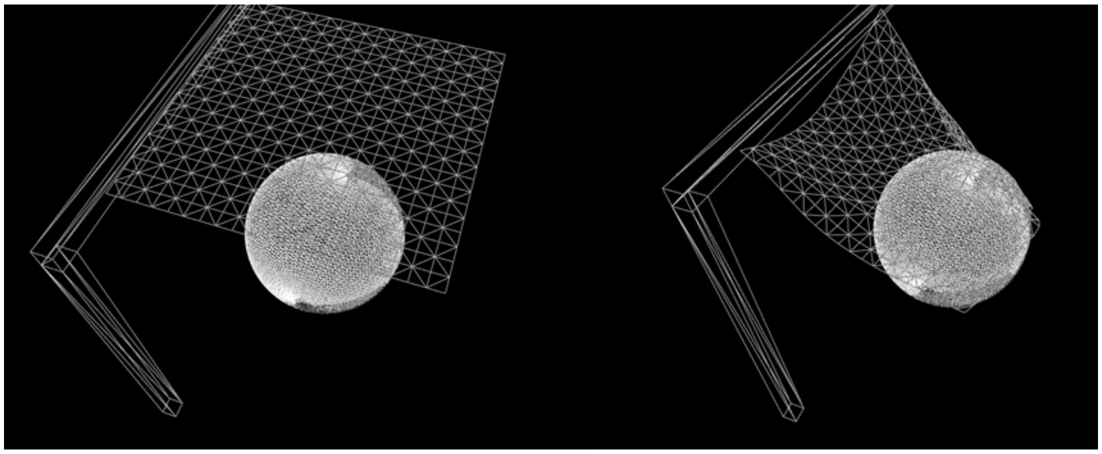
Figure 1: The cloth simulation from Dayeol Lee

The scene rendered is a cloth modelled as a triangle mesh with n particles, hanging by 2 of its corners on an arch. Initially the cloth is not animated, it lies flat horizontally with no forces acting on it. After a key is pressed to activate the forces, the gravity will pull the cloth toward the ground. With the way the cloth is implemented, the vertices have a mass and are connected to form a spring-mass system composed of a damping force and spring force. With this implementation, the cloth will bend and deform to mold the shape of the sphere object that is placed under it.

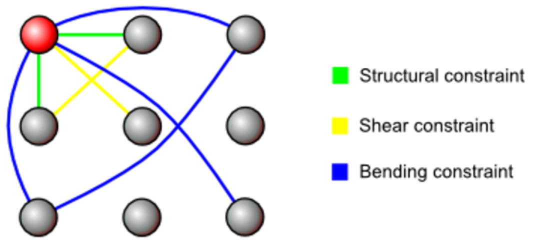

Figure 2: The representation of the cloth [4]

However as it will be further explained in the Development section, our project encountered some difficulties. Ultimately the focus of our programming efforts shifted to the particle system, rather than trying to merge the two applications into one. The final result was a further polished particle system onto a flat colored 3D mesh which would disintegrate as the fire spread and destroyed the vertices that made up the mesh.

2 Technologies

2.1 Overview

This section gives a broad overview of the different software used throughout the development of the project, as well as a general schematic to give a visual representation of how the different classes within the application are related and interact with each other.

2.1.1 Compilation

Development of this application was done purely through Microsoft Visual Studio 2017 on Windows operating system with the Windows 10 SDK, and in the C++ programming language. To build our project, the file “base-code.sln” has to be opened in Visual Studio and the build command should be executed. An executable file will be produced and can be run if the “freeglut.dll” file is placed in the same directory.

2.1.2 Librairies

OpenGL was the primary graphical engine that was used, while additional libraries included “freeglut” [3], “glm” [1], and “glew” [5]. The use of these libraries allowed for free access to open-source functions that streamlined the graphics pipeline. These libraries would facilitate the calculations involved in simulating physics and allow a series of vertices to be animated as a piece of cloth whether it drapes over an object, is pulled by winds, or is disintegrated by fire.

2.1.3 Communication

As development of any application proceeds, the usage of version control becomes an obvious necessity. For this project, the web-based repository Bitbucket [2] was used as the group’s repository. It allowed for the different team members to focus on different aspects of the project simultaneously, and it was accessed remotely through use of Git.

2.2 Class diagram

Figure 3 outlines the numerous classes utilized in our application for the particle system and illustrates the class hierarchy for “GameObject”. Our classes are explained in detail in section Final implementation.

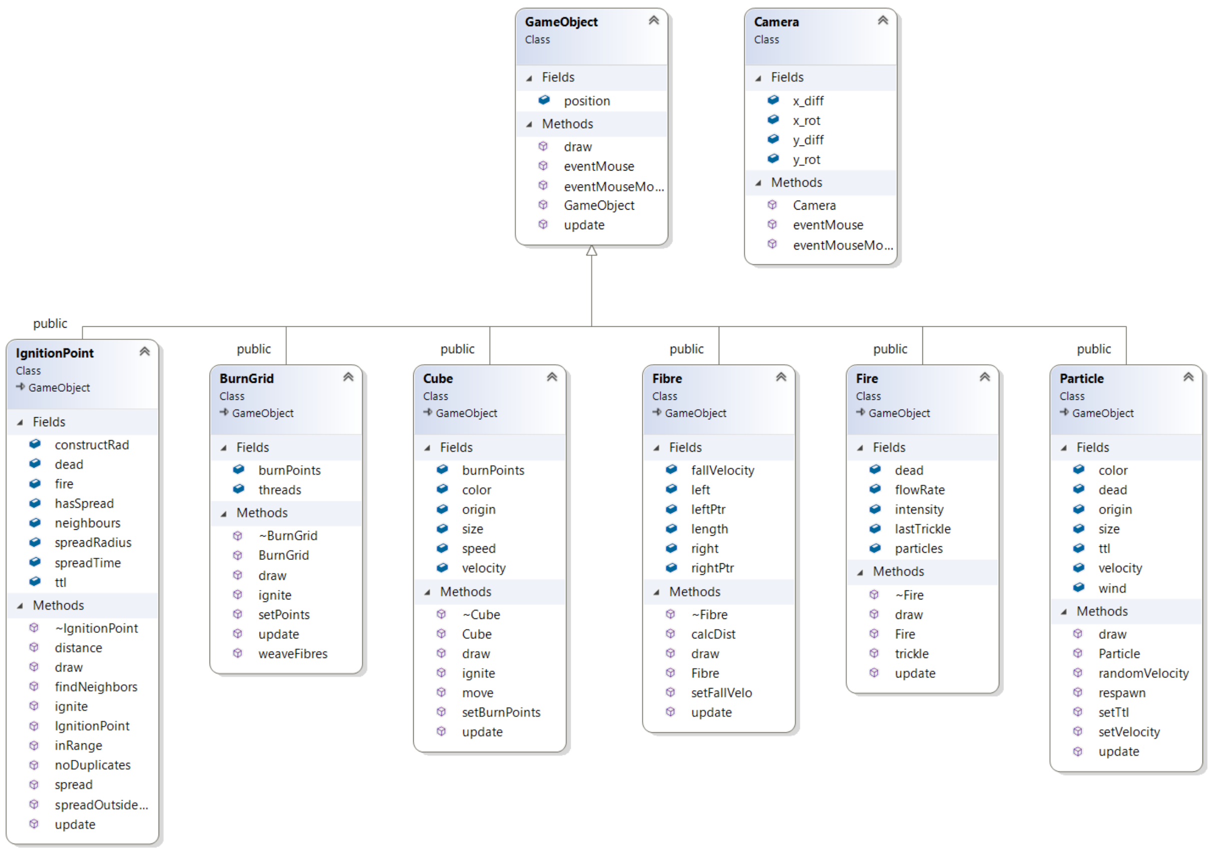

Figure 3: Class diagram of our project

3 Development

3.1 Initial Implementation

3.1.1 Particle System

The particle system for the fire effect was originally built for the second assignment of the course. The first version of our particle system was built as a series of 49 points all evenly spread out along a horizontal axis forming a 7x7 square. The "fire" would start at one of 3 hard-coded positions, and then spread accross all the other points.

The `Particle` class underwent a thorough revision as it was planned for it to be applied to an outside application. Initially, the `Particle` class had six attributes. A starting position or `origin` would be used so the particle could easily respawn at its original location. A `position` and a `velocity` attribute described its current position and velocity. Each particle also had a randomized time to live denoted as `ttl`. Additionally a `burnTime` attribute served as a counter that would determine how many times a particle could respawn but this was better handled when implementing the `Fire` class. The last attribute was simply the particle's `color` which would be stored in a `glm` 4-dimensional vector. This `color` had an initial value of (1.0, .3, 0.0,

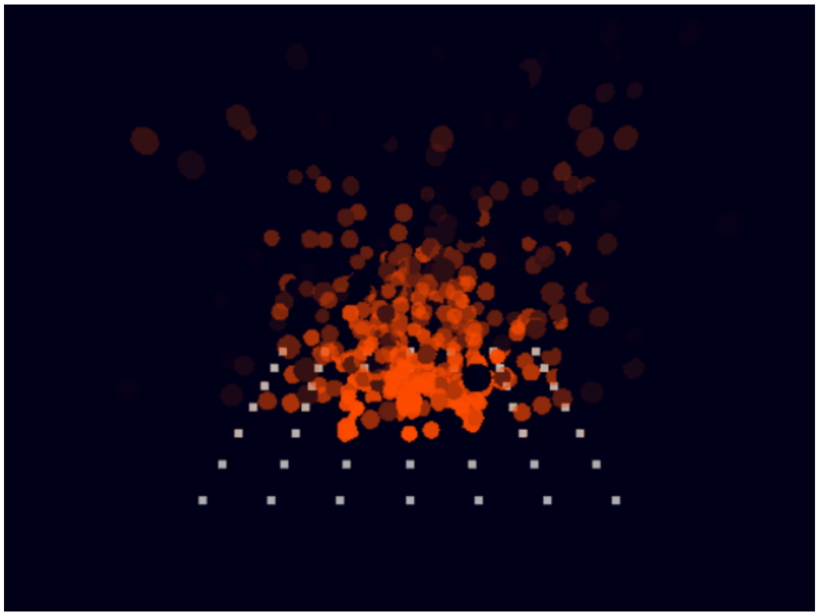
Figure 4: The first implementation of our fire particle system

1) which would generate a bright orange color. However as seen in Figure 4, the `color` fades over time at a random rate, and also becomes partially transparent. This was meant to add some slight realism to the fire.

Finally the `Particle` class is in fact the child of an `Object` class. This `Object` class however was later renamed to `GameObject` in the final project to avoid naming conflicts with the cloth simulation's "Object" class. In early development, the only notable detail about this parent class, was the inclusion of an extra default attribute called `dead`. This attribute was a boolean variable used to indicate when a `Particle` object should stop being drawn. This attribute was first placed in the `GameObject` class to facilitate access as the Particles were originally stored in a vector of `GameObjects`. However the `GameObject` class was later reworked to be a broad abstract class with only an attribute describing `position` was shared among all children. The `dead` boolean was also moved to the more specific objects in the class hierarchy `IgnitionPoint`, `Fire`, `Particle`.

3.1.2 Fire Spreading

Apart from the particle system itself, the other critical component to simulating fire was its ability to spread. Our methodology to accomplish this heavily changed from the initial particle system we created for the second assignment compared to the final project.

The way we originally implemented our fire spreading algorithm was by spawning a sphere that is not rendered by default (a key press reveals the sphere), which would grow in size. This one sphere would expand and eventually its boundary would collide with the points placed on the horizontal plane. This triggers the creation of a new fire emitter at the position of the point, and this emitter would start creating new particles, thus creating an effect of the fire spreading.

Initially it was designed this way because it was simplistic method that required no hard-coding to specify which points to start burning in what order, but our team realized in a cloth simulation this method would be flawed. Having a growing sphere decide which points start to burn essentially risked that the fire could jump to different parts of the cloth when it should not. An easy example of this would be if the cloth is draped over a character. If the fire starts on the character's backside, the growing sphere would spread the fire to the character's front instead of spreading over the character's shoulders, and then to his front. Knowing this, we realized what worked for the assignment needed to be reworked for the project.

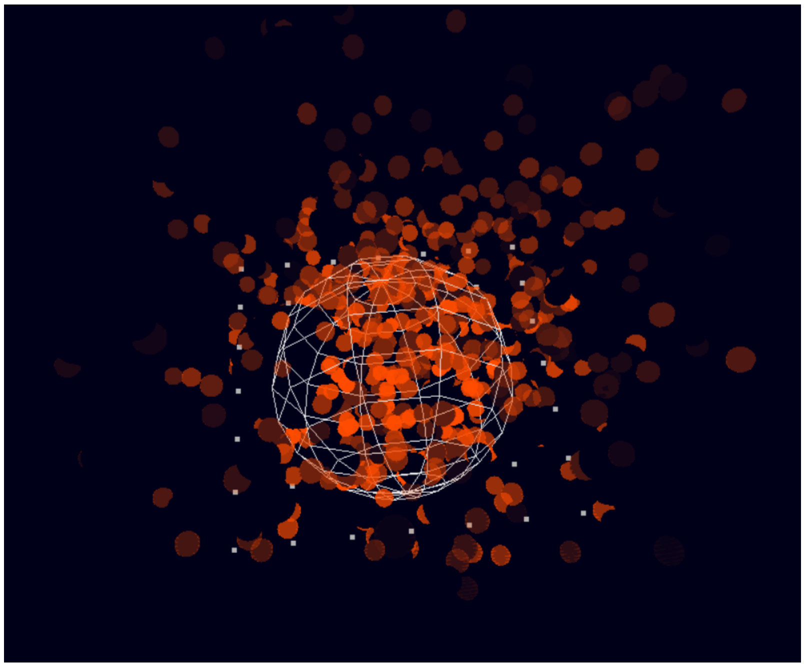

Figure 5: The first implementation of the fire spreading (with the rendered sphere)

For the final project, we have refactored the spread logic to utilize a spherical hitbox (3 dimensional distance formula) between ignition points rather then a single increasing sphere of fire. Points were assigned as potential fire sources (`Ignition Points` (IP)) that could catch fire and spread to other surrounding sources. Once set-alight, an IP instantiates a fire on itself which acts as a particle emitter. Before dying off, the IP will cyclically call it's spread method and check what is in range of itself, and attempt to ignite them. This means that instead one growing sphere, our team uses "flammable" points that each have a small fixed radius that determines how far they can spread to other IP's.

3.2 Final implementation

In this section we will provide a brief but concise overview of the hierarchy of classes used within our final application. Each class is described in detail from in descending order from largest abstract objects to individual particles.

3.2.1 Game Object (Abstract)

The system is organized in a hierarchy of objects each handling the diverse method calls of it's children. Every object that interacts on the scene inherits from the **Game Object** class. This abstract class serves to assure that any subclass has all the necessary functionality to be drawn, updated and listen to event calls that the main render loop will use on every frame. It also assures that an object will always have a 3 dimensional coordinate to serve as the objects origin. This origin mainly used for the draw functionality and in some cases also server in collision detection. These functions are the following:
virtual void draw();
virtual void update(float dt);
virtual void eventMouse(int button, int state, int x, int y);
virtual void eventMouseMotion(int x, int y);

3.2.2 Burn Grid

The Highest hierarchical object in the system is the **Burn Grid** which is instantiated in main. It inherits from `GameObject` thus having an origin and necessary methods to be rendered. It contains a vector of Ignition Points that are the source of the fire logic and it organises them in a square grid. Once instantiated, the Burn grid randomly selects one of it's ignition points and ignites it. The draw method is overwritten to draw all the subsequent ignition point's draw methods.

3.2.3 Ignition Point

An **Ignition Point** (IP) servers the purpose of a fire source. It can be easily drawn on the scene as it is also a Game Object. An ignition point can be lit on fire by using it's ignite method at which point it will instantiate an object of the Fire class in it's property list. At a given time, before the IP dies, it will call it's `spread` method. This method will use a spherical hit box detection (3 dimensional distance formula) with all the surrounding IP's and will handle a collision by igniting the colliding IP, causing a chain reaction of ignitions. An IP's draw and update methods also calls it's fire properties draw and update methods (if not null).

3.2.4 Fibre

A **Fibre** is a class that represents a physical connection between two IP's. A **Fibre** object takes two IP pointers into its constructor, and it uses the position of those IP's as `left` and `right` positions. The fibre is then drawn simply as a line between these two attributes. A fibre has essentially three stages before it disappears, and these are illustrated in the color used to draw them. The first being its starting state where it is static and connected to both points. In this state the fibre is colored bright green to clearly contrast with the fire. When either `left` or `right` IP is declared `dead` the color of the fibre is changed to dark green to show it is partially burned away. The position mapped to this dead IP is also pushed down by a gravity force. However, this force only affects the detached point slightly so the fibre remains fixed at a certain length. Finally when both points are dead, the fibre changes color once more to a burnt orange while falling at a randomized velocity, implemented in a similar fashion to the **Particle's** `randomVelocity()` function.

3.2.5 Fire

The **Fire** class serves as an aggregate class to control the particles generation, essentially acting as the emitter of the particle system. It contains a vector of Particles that are instantiated at a predetermined rate (trickle) and are recycled in order to avoid performance problems. When the fire dies, it will keep rendering it's remaining particles that are alive to avoid spontaneously disappearing Particles before their time to to live (ttl) expires.

3.2.6 Particle

The smallest object of the hierarchy is the **Particle**. This class remains largely unchanged from the second assignment. Upon instantiation it has a position, and an upward velocity vector that it will follow every time the update method is called. To avoid tying the animation to the frame rate, we instead multiplied it's movement to the Delta Time (time between each frame render) to ensure the animation is smooth and consistent. Visually, the particle has a couple characteristics that change over time from the moment it spawns at the origin to when it disappears. These characteristics are its size and color. At its origin, the color is set to an orange-yellow (**glm::vec4(1.0, 0.6, 0, 1.0)**), to make a more realistic appearance of fire, the color then gradually becomes a stronger red. The same occurs with the size, that is, the size of the particle decreases over time as it moves away from the origin.

3.3 Other features

To illustrate another effect that the environment can have on our system, we applied a wind force to the parts of our system that move. In our case, the particles have an upward velocity in a random direction, and the fibre have a downward velocity due to the force of gravity. To display an effect of wind on these two elements, we added a force vector in the negative z direction to the existing velocity. This wind force vector is initially set to 0 at the start of the application, and then set to **windForce = glm::vec3(0, 0, -50);** when the key 'w' is pressed. This **windForce** vector is also then multiplied by the delta time to make it consistent with the frame rate.

Another effect we wanted to test was the spreading of the fire to an object outside of our initial burn grid. For this we added a simple cube that can be moved with the "WASD" keys in the x and y directions. The cube has an ignition point on each face, and each ignition point is tested for it's distance with the current points of our grid that are on fire. If an ignition point is in the range, we initialize a fire on it using the `ignite()` function of `IgnitionPoint`. The result can be seen on Figure 6.

3.4 Difficulties and Issues

The previous sections have been detailing the development process of our particle system, but have not yet addressed the cloth simulation that it was intended for. Simply put, the two applications were too different to be put together. The cloth simulation by Dayeol Lee [4] utilized shaders and different libraries from what we had used in our particle system. The methodologies for how the two different applications render their objects resulted in the cloth simulation rendering as normal, but the particle system and `BurnGrid` would always be positioned in relation to the current position of the camera. Therefore what Figure 8 is showing, is that the particles in view are glued to the camera. That is, if the camera moves around the scene that

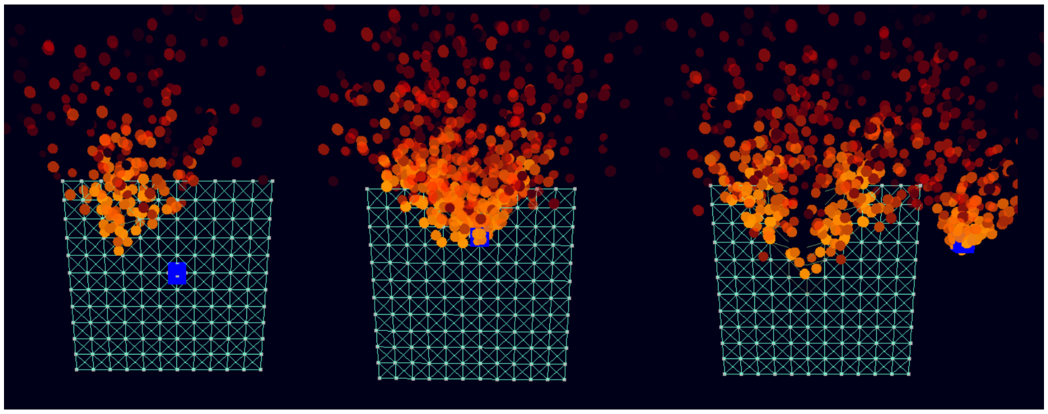

Figure 6: The cube catching on fire

the particle system will follow the same transformations.

We attempted to address this issue with multiple different approaches but the result either broke the application all-together, resulted in no particles rendering, or no change at all. Some of these attempts involved setting the coordinates of the `IgnitionPoints` to the coordinates of the vertices on the cloth which was stored in a vector of `vec4` which was (unintuitively) named "particles". However part of the difficulty in accomplishing this came from the fact that the `vec4` class used by Dayeol Lee [4] came from a header file named **Angel.h** which meant that a vec4 from this library would not easily be compatible with the data structures used from the textbfglm [1] library we used in the development of our particle system. Even if the team managed to properly store the coordinates of a vertex from the cloth into a `IgnitionPoint`, the particles either would not render at all, or remain stuck to the camera's view, and this is likely because of the way the vertex shader in Dayeol Lee's application computed the position by multiplying it with their Model view, Projection and LookAt matrices, in addition to how the methodology of how our `Camera` object operated differed from that of Dayeol Lee's [4] `Camera` class.

When it was obvious this approach would not work, we attempted to implement our own shaders to work along with Dayeol Lee's [4] `GLSLShader` files. Following various resources online [7] [6], we adjusted our particle system to render using shaders, however this too failed to address the main issue of properly mapping the particle systems onto the piece of cloth. Ultimately, we had to abandon integrating Dayeol Lee's [4] cloth simulation with our own particle system, and so we focused on modifying and implementing a static mesh with what we had already created, and that at a future date, could be implemented to simulate cloth-like physics.

4 Results

4.1 Visual results

After our first discussion with the instructor regarding our project, our initial goals were as follows:

1. Have a character or object prop in clothes
2. Clothes burn at certain rate removing cloth springs and nodes and particles gradually
3. Integrate libraries that do clothes and particle systems where the data structures between the two adapted for the desired simulation interaction
4. The character should move / run with the burning cloths simulation to follow
5. Fakenect / keyboard for character animation from the sample repository

After we encountered issues with cloth simulation as discussed in section 3.5, we focused our effort on our goal #2, the spreading of the fire on a mesh of points connected by fibre links. The result can be seen on Figure 7.

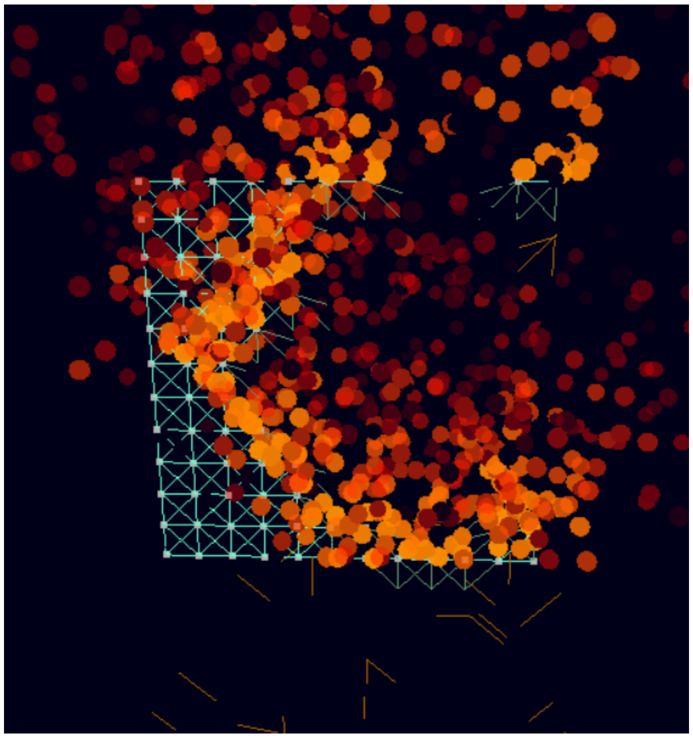

Figure 7: Final result of the fire spreading to the mesh

4.2 Limitations

The objective of the project was to implement a simulated piece of cloth material, which would interact and animate around its environment. The main interaction was planned to be setting the cloth on fire. Unfortunately as mentioned in section 3.5, we did not manage to map our particle system to the cloth vertices. Therefore, our current spreading of the fire is only applied on a static grid of points, which do not move. The reason we had difficulty implementing this feature was primarily because the library imported for the cloth simulation used shaders to both draw and calculate particle positions while our custom particle system used Glut to draw our points in camera space. Theoretically, our particle system should be able to produce a spreading fire effect across a mesh, which include disintegrating and falling fibres, fire particles affected by winds, and a fire that can even spread to separate objects provided the objects contain `IgnitionPoints`.

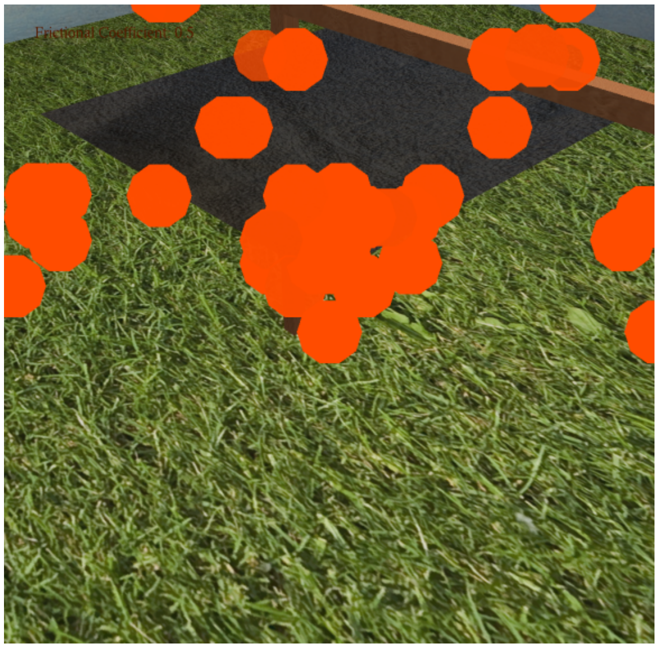

Figure 8: Failed attempt at mapping our particle system to the cloth simulation example

4.3 Further development

Continued development on this project would naturally allow for the implementation of soft body simulation on the burn grid. Expanding the ignition points and connecting fibers with a Mass-Spring System would yield cloth-like characteristics. This would greatly expand the usefulness of the simulation to allow clothing, hair and other objects with dynamic bones to built on the scene. With the softbody simulation in place, we could expand the forces that are applied to the fire to the burn grid such as wind force and gravity.

Additionally, the fire spread logic could be expanded to allow the particles to ignite other objects upon contact with an ignition source. Objects are composed of different materials that have various properties to fire resistance or vulnerabilities to ignition. This would give the spread a more accurate representation to how realistic fires expand and could be useful in fire prediction technology such as firefighting simulations.

5 Conclusion

The final product of this project is a successful implementation of a particle system that visually demonstrates the effects of fire spreading throughout a grid-like material. Though our initial attempts to implement a Mass-Spring System within our project proved unsuccessful, we are convinced that it would be possible to add this functionality to the current state of the application which would greatly expand its usefulness in both animation and practical simulation.

References

[1] G-Truc Creation. *OpenGL Mathematics*. [online]. https://glm.g-truc.net/0.9.9/index.html. 2019.

[2] Mark-Zackary Rodrigues-Gillich Cynthia Cherrier Erik Smith. *comp477-f19-06*. [online]. https://bitbucket.org/comp477f19team06/comp477-f19-06/src/master/. 2019.

[3] Mark Kilgard. *The freeglut Project*. [online]. http://freeglut.sourceforge.net/. 2019.

[4] Dayeol Lee. *Cloth Simulation*. [online]. https://github.com/dayeol/clothsimulation. 2015.

[5] Nigel Stewart. *The OpenGL Extension Wrangler Library*. [online]. http://glew.sourceforge.net/advanced.html. 2019.

[6] Joey de Vries. *Particles*. [online]. https://learnopengl.com/In-Practice/2D-Game/Particles. 2014.

[7] Joey de Vries. *Shaders*. [online]. https://learnopengl.com/Getting-started/Shaders. 2014.