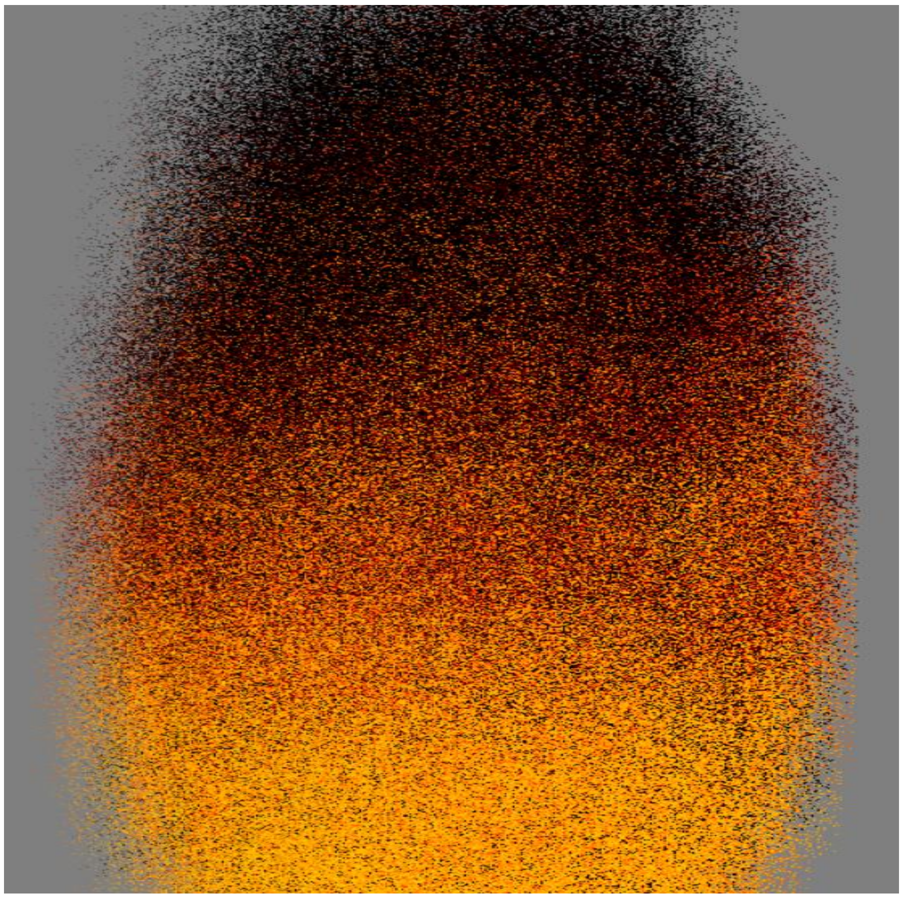


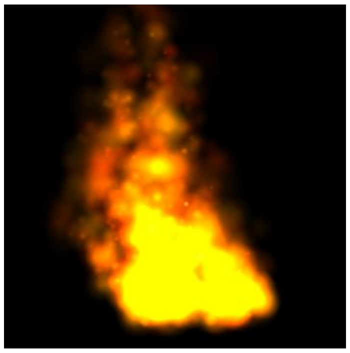
GPU Based Particle Systems (Fire Simulation)

Final Project Report

Author: Kyle Ahn

Professor: John Braico

INTRODUCTION



From games to movies, computer graphics have been used in order to render more detailed objects and simulation recently. Although rendering geometries such as “box”, “sphere” and “triangular meshes” are quite straightforward, rendering natural phenomena such as, fire, flames and fluid, is a bit more difficult because of how it’s animated based on the real-time. So, particle systems have long been recognised as a solution to render lively visual environments such as fire, flames and fluid. Also, to mention briefly about this history of particle systems, particle systems are unofficially said to have been used in video games as early as 1960’s using 2D rendering techniques. For the official record, however, it is believed that William T. Reeves used particle systems to model fire engulfing a planet in the film “*Star Trek II*”. The reason that he is considered as the creator of particle systems is because wrote a paper to explain how he rendered the fire engulfing a planet in 1983 in the film (Ginman and Malmros, 2013, p. 9). In order to introduce this interesting particle systems, this paper will discuss about; (1) the basic idea of how particle systems work; (2) a CPU based particle system and its limitation; (3) implementation details of a GPU based particle systems; (4) improvements of the implementation; and (5) an honest review of the implementation.

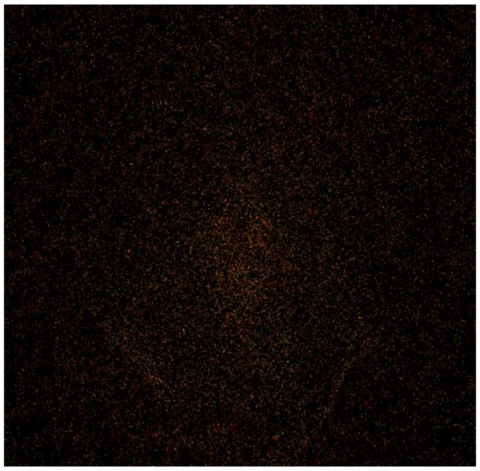
HOW A PARTICLE SYSTEM WORKS

Before explaining how a particle system works, it is important to understand what it means by “particles”. Particles are small and simple points, images or meshes that are displayed and animated in large numbers by a particle system(s). Each of these particles represents a small portion of a natural phenomena, such as fire, flames and fluid, and these particles gather together to create a complex looking entity. In that sense, a particle system is simply a system generates and controls these particles to render a natural looking scene with different factors. Each particle is given a predetermined *lifetime*, where it can undergo various changes such as colour, size or velocity changes. A particle system *emits* each of these particles with its lifetime at random or pre-set position. For a fire simulation, this starting position would be the root of the fire where the flames begin to appear. Then, the particle system is also responsible for *removing* the particles that have finished their lifetime so that they cease to appear on the screen. The rate of particles being emitted by a particle system is called an “*emission rate*”, which determines how many particles are emitted per second. Obviously, this emission rate should be set different depending on what the programmer would like to render in the scene. For example, one would want to set the emission rate relatively high in order to render a more realistic fire simulation. In addition to the lifetime, each particle also has a *velocity* vector (2D or 3D) to compute the distance and direction that it moves. This velocity vector is often influenced by the particle system's *forces* and *gravity* to simulate more natural movement that happens in the reality. Using these variables, for example, a fire can be simulated by using a high emission rate, letting the particles rise with positive gravity (upward) and gradually disappearing as they spend their lifetime. There are many other smaller factors that determine how a particle system works, but these are the basic variables that can be used to simulate many kinds of natural phenomena quite convincingly.

CPU-Based Particle System & its Limitation

So, now that the basic idea behind the particle systems are explained, it is time to look at how it is implemented and its performance more detail. In this section, the author will discuss how a CPU based particle system is implemented and what the limitation that it faces.

Just like many other graphics techniques, a particle system can also utilise either CPU or GPU, or both. In fact, the author has implemented both systems to learn more about how they work and what the impact of using CPU or GPU is. First of all, the author started with a simple CPU based particle system to render a real-time based fire simulation. Without much knowledge about the particle system at the moment, the author expected a decent output that is realistic and convincing enough using CPU only. As described in "How a Particle System works", a "Particle" class, "Particle System" class, and "Renderer" class were implemented using C++ to render the fire simulation. The Particle class contained the "position", "velocity", "acceleration" and "lifespan" of each particle; the "Particle System" class contained "gravity" and "update"; and the "Renderer" class contained anything else that was necessary to render the animation such as setting up a window, update per frame and such. Initially, the emission rate was set as "1000" per frame, but the program was not able to handle it at all, causing the FPS to drop to 5 ~ 10. Then, the emission rate was adjusted to "500" per frame, but the program was still not able to render the animation smoothly. Ideally, it was expected to see 60 FPS with an emission rate of



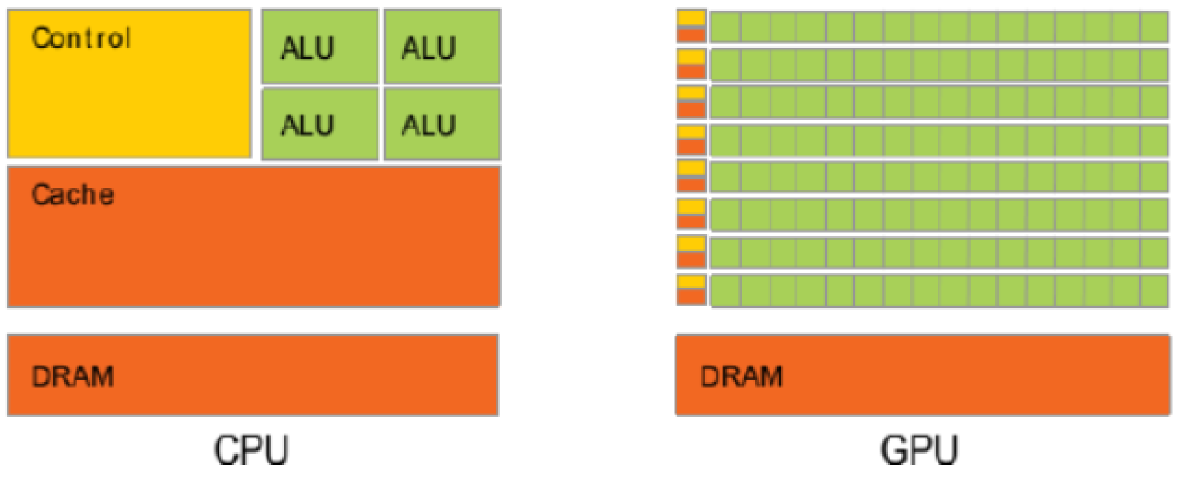
1000 points(pixels) to render a reasonably good looking fire simulation. However, the emission rate had to be decreased down to "200" to render the animation smoothly. As you may observe in the figure on the right hand side, 180 emission rate was not enough to render a decent fire simulation. There were just not enough particles to render a decent fire simulation with good density. According to calculation, the total number of particles that could be rendered smoothly (60 FPS) using CPU based particle system was only about 180 (emission rate) * 60 (FPS) * 4.0 (lifespan seconds) = 48,000 particles. Although it could have been possible to render a relatively small fire with 48,000 particles, it did not look convincing enough to be realistic. Furthermore, a CPU-based particle system has been widely used for a long time (2 decades at least), which wasn't quite the purpose of this project. Therefore, a GPU based particle system was considered a better option to choose to solve this limitation.

Implementation Details of a GPU-Based Particle System

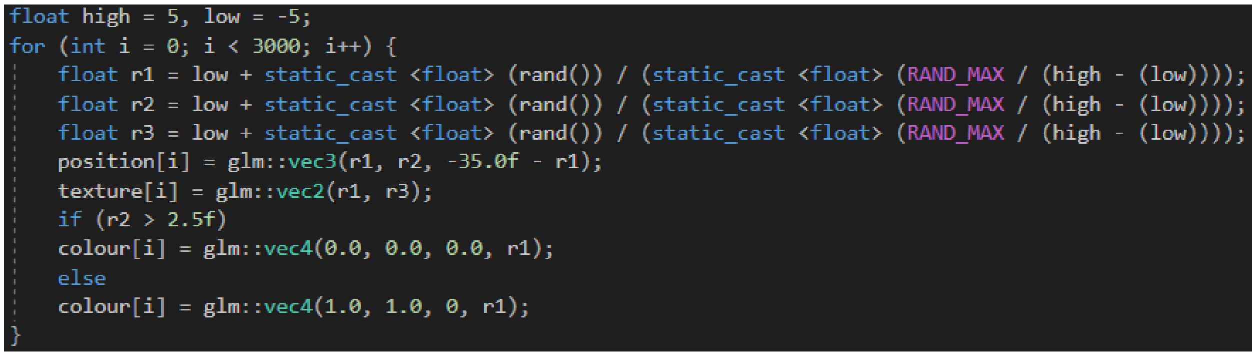
Before going in to the implementation details of the author's GPU based particle system, please note that the author's GPU based particle system is heavily based on "Visualisation of smoke using particle systems" by Ginman, V. & Malmros, K., and "Building a Million Particle System" by Latta, L. Although these are the two mostly referred sources, there are a few more sources that were helpful, which are all listed in the bibliography on the last page of this report.

GPU Based Particle Systems

As mentioned earlier, the performance was the issue with the CPU-based particle system. It was not able to render more than about 48,000 particles (points) at a time on the screen smoothly. The solution to render far more than 48,000 particles is to move the rendering part of the particle system

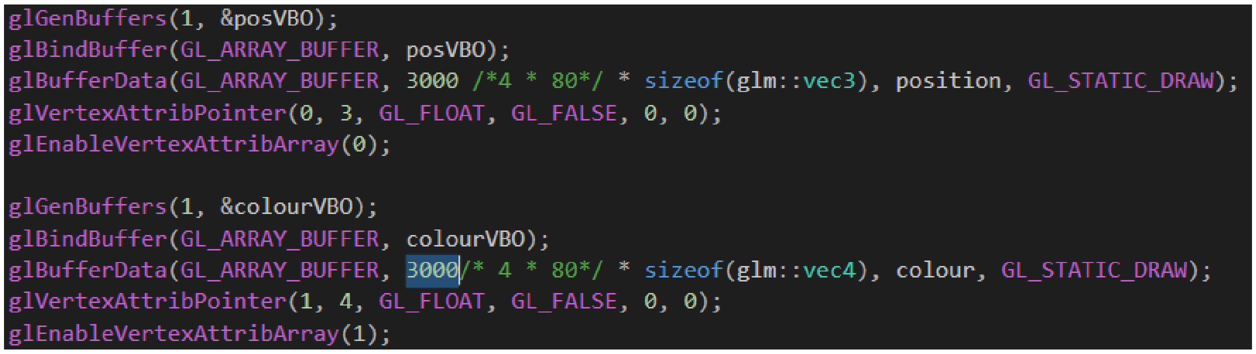


onto the GPU, leaving only the computation part of the particle system on the CPU. The reason why it is important to leave the computation part for the CPU, including updating velocity, position and colours, is due to the architecture of the CPU and GPU. As you observe in the figure above, a GPU is tailored for highly parallel operation, meaning that it is not able to compute sequential operations such as allocation problems quickly (Latta, 2004). In this case, the allocation of positions, velocity and colours as well as their computation cannot be handled efficiently by the GPU. Thus, the CPU, which is optimised for sequential operations (sequential code performance), is ideal for computing these. Reflecting on this fact, the implemented GPU based particle system let the CPU handle the allocation and computation problems as you see in the picture below. (ParticleSystem.cpp). What this code does is that it assigns a



```
float high = 5, low = -5;
for (int i = 0; i < 3000; i++) {
    float r1 = low + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX / (high - (low))));
    float r2 = low + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX / (high - (low))));
    float r3 = low + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX / (high - (low))));
    position[i] = glm::vec3(r1, r2, -35.0f - r1);
    texture[i] = glm::vec2(r1, r3);
    if (r2 > 2.5f)
    colour[i] = glm::vec4(0.0, 0.0, 0.0, r1);
    else
    colour[i] = glm::vec4(1.0, 1.0, 0, r1);
}
```

random value between -5 and 5 to position, colour and texture using the CPU so that we can load this data onto the GPU using VBOs like the picture right below. Notice that the size of the

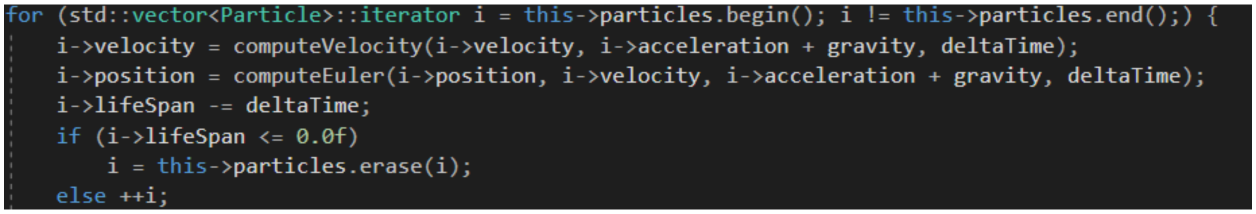


```
glGenBuffers(1, &posVBO);
glBindBuffer(GL_ARRAY_BUFFER, posVBO);
glBufferData(GL_ARRAY_BUFFER, 3000 /*4 * 80*/ * sizeof(glm::vec3), position, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);

glGenBuffers(1, &colourVBO);
glBindBuffer(GL_ARRAY_BUFFER, colourVBO);
glBufferData(GL_ARRAY_BUFFER, 3000/* 4 * 80*/ * sizeof(glm::vec4), colour, GL_STATIC_DRAW);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(1);
```

array is 3000, which means that the emission rate 3000, generating 3000 particles per frame. In comparison to the CPU based particle system which was only able to render about 200 particles per frame, 3000 particles are quite a difference.

Now that the initial position, colour and texture are set (birth of particles. The lifespan of all the particles pre-set by the particle system.), it is necessary to have a function to update the velocity, position and lifespan of the particles. Update() function the particles is a key to the natural-looking real-time based fire simulation. As mentioned above, the author's implementation of the particle system lets the CPU do all the sequential operations, thus updating the velocity is also done on the CPU. In each frame, the particle system integrates over the gravitational acceleration and advance the velocity and

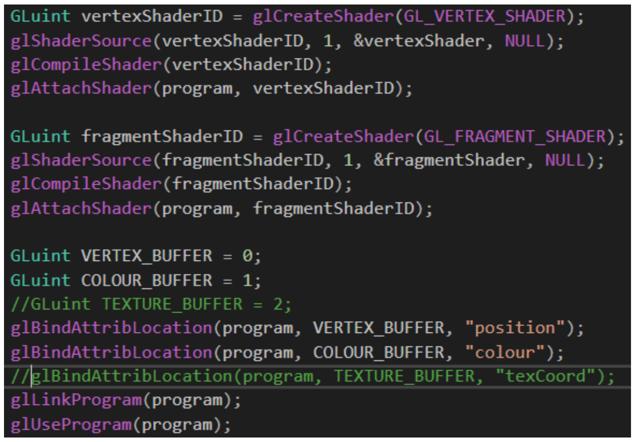


```
for (std::vector<Particle>::iterator i = this->particles.begin(); i != this->particles.end();) {
    i->velocity = computeVelocity(i->velocity, i->acceleration + gravity, deltaTime);
    i->position = computeEuler(i->position, i->velocity, i->acceleration + gravity, deltaTime);
    i->lifeSpan -= deltaTime;
    if (i->lifeSpan <= 0.0f)
        i = this->particles.erase(i);
    else ++i;
```

position of the particles according to the time elapsed since the last frame. As you see in the code above, update() function iterates every particle in the particle system and calculates the new velocity and position using two helper functions, "computeEuler" and "computeVelocity". What computeVelocity function calculates is a parametric equation to calculate the new velocity: $\mathrm{v} = \bar{v} + a \cdot \Delta t$, where v is the newly calculated velocity, $\bar{v}$ is the current velocity, a is the acceleration and Δt is the time elapsed (Latta, 2004). Notice here that "a" = "a particle's current acceleration vector" + "the particle system's pre-set gravity". Then, computeEuler function simply calculates new position using the velocity and the acceleration with the delta time: $newPos = pos + (\bar{v} + a \cdot \Delta t)\Delta t = post + v * \Delta t$ (Latta, 2004). Finally, we update the lifespan simply by adding time delta time, removing any particles that had already spent their lifespan.

Next step is sorting the particles based on the distance. In this implementation, the sorting is done simply by the distance in the z axis between the camera and the particle. In this implementation, a complex sorting method such as merge sort was not really necessary because particles that are hidden behind another particle could be just ignored.

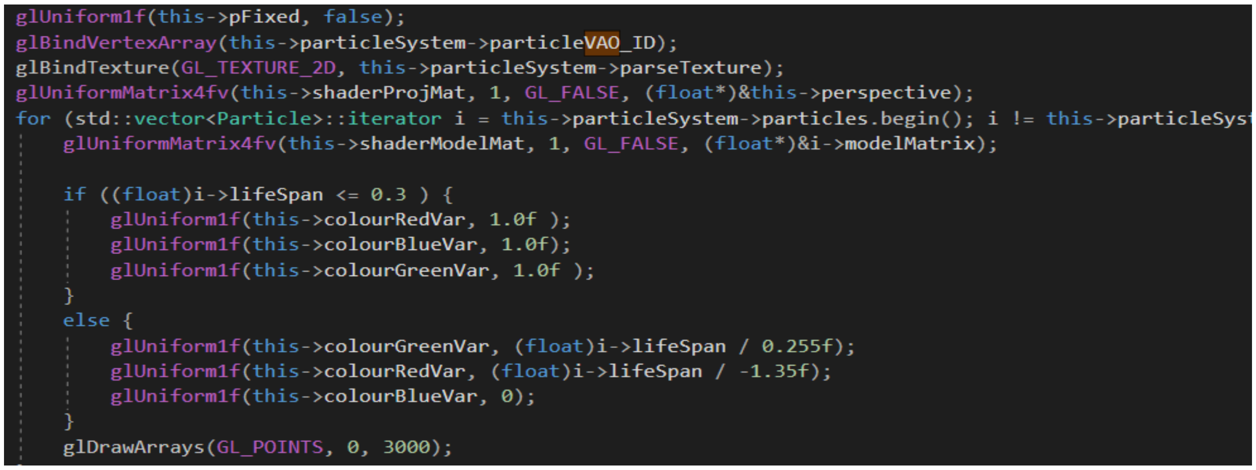
After sorting the particles, here comes the most interesting part of this implementation. The position data dealt so far should now be copied from a texture to vertex data in order to truly utilise the GPU. Before doing so, it is necessary to initiate the vertex shader and fragment shader. Without these, of course, it is not possible to utilise the GPU at all. The vertex shader and fragment shader are stored as .glsl files. Initiating the shaders should be quite straightforward if one knows how to use OpenGL, but a screenshot is included just to help you understand what the author has done to initiate the shaders. After the initiation of the shaders, a VAO is used in the code below. For the current implementation in this section, however, it does not really require a VAO since



```
GLuint vertexShaderID = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShaderID, 1, &vertexShader, NULL);
glCompileShader(vertexShaderID);
glAttachShader(program, vertexShaderID);

GLuint fragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShaderID, 1, &fragmentShader, NULL);
glCompileShader(fragmentShaderID);
glAttachShader(program, fragmentShaderID);

GLuint VERTEX_BUFFER = 0;
GLuint COLOUR_BUFFER = 1;
//GLuint TEXTURE_BUFFER = 2;
glBindAttribLocation(program, VERTEX_BUFFER, "position");
glBindAttribLocation(program, COLOUR_BUFFER, "colour");
//glBindAttribLocation(program, TEXTURE_BUFFER, "texCoord");
glLinkProgram(program);
glUseProgram(program);
```

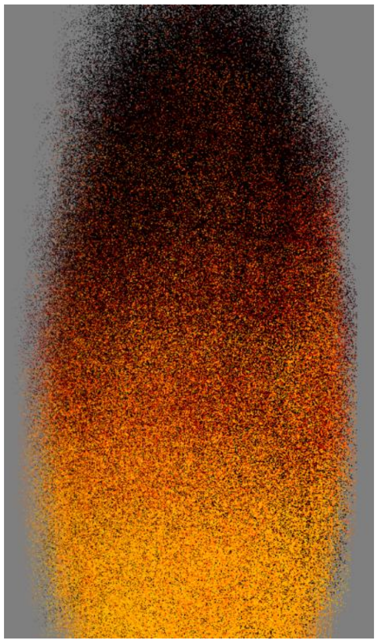


```
glUniform1f(this->pFixed, false);
glBindVertexArray(this->particleSystem->particleVAO_ID);
glBindTexture(GL_TEXTURE_2D, this->particleSystem->parseTexture);
glUniformMatrix4fv(this->shaderProjMat, 1, GL_FALSE, (float*)&this->perspective);
for (std::vector<Particle>::iterator i = this->particleSystem->particles.begin(); i != this->particleSys
    glUniformMatrix4fv(this->shaderModelMat, 1, GL_FALSE, (float*)&i->modelMatrix);

    if ((float)i->lifeSpan <= 0.3 ) {
        glUniform1f(this->colourRedVar, 1.0f );
        glUniform1f(this->colourBlueVar, 1.0f);
        glUniform1f(this->colourGreenVar, 1.0f );
    }
    else {
        glUniform1f(this->colourGreenVar, (float)i->lifeSpan / 0.255f);
        glUniform1f(this->colourRedVar, (float)i->lifeSpan / -1.35f);
        glUniform1f(this->colourBlueVar, 0);
    }
    glDrawArrays(GL_POINTS, 0, 3000);
```

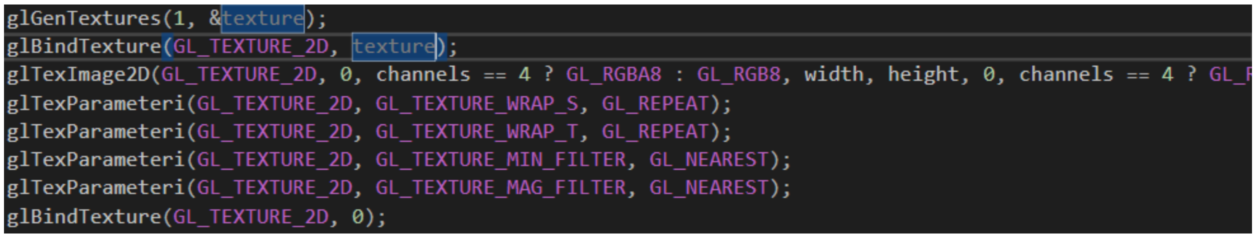
the particle system is only rendering points rather than a complete rendered object, which consists of multiple VBOs. This VAO part of the code is utilised in order to render texture-mapped objects and it will be discussed in the next section, “Difficulties & Improvements”.

Thus, for the current implementation, we could simply loop all the particles and draw the points (pixels) to finally render a fire simulation. In this code, a “colourXXVar” variables, which is used inside the fragmentShader to variate the colours, is defined so that the fire starts with yellow at the root, becomes more of red as it rises, and gets darker like smoke at the end of the lifespan. The picture on the right hand side is the output that this implementation can get. The fire is real-time based, and the number of particles to render this fire is much larger than the CPU based particle system. In fact, approximately, 60 (FPS) * 3000 (Emission Rate) * 1.75 (≈ lifespan) = 315,000 particles are constantly being rendered on the screen through birth and death of the particles. Although this seems to be a good improvement from the CPU based particle system, there could surely be more improvement in order to render a bit more realistic fire simulation.



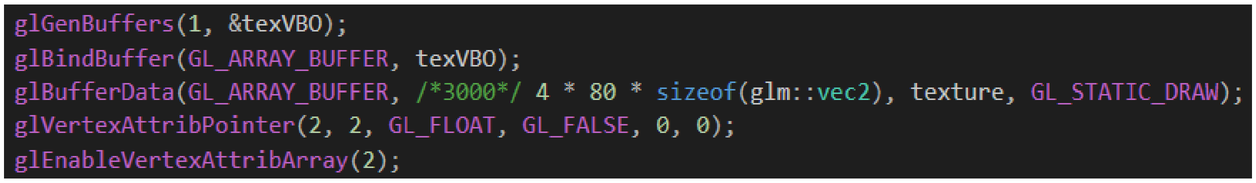
Improvements

In the last section, a particle represented a point/pixel in a particle system, but there is also a limitation to this approach. If you look closer to the particle system which uses a point/pixel as a particle, the natural phenomena does not really look as realistic. (It is quite difficult to render convincing fire simulation only with points.) It might have been a bit confusing as to why there was commented “texture” at all in the previous codes. In fact, on top of the current implementation, a decent amount of effort was made in order to improve more realistic looking fire simulation, and that is precisely why



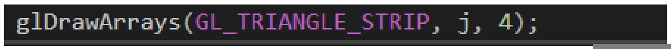
```
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
glTexImage2D(GL_TEXTURE_2D, 0, channels == 4 ? GL_RGBA8 : GL_RGB8, width, height, 0, channels == 4 ? GL_
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glBindTexture(GL_TEXTURE_2D, 0);
```

“texture” was included afterwards. The idea is to load in a texture and map it to render as a geometry, a quad in this implementation. The process is very similar except that it is now necessary to map the texture using OpenGL code again like the above picture. Then, the initialised texture information should be moved onto the GPU using VBO. At this point, it is probably easy to notice where it is heading. The



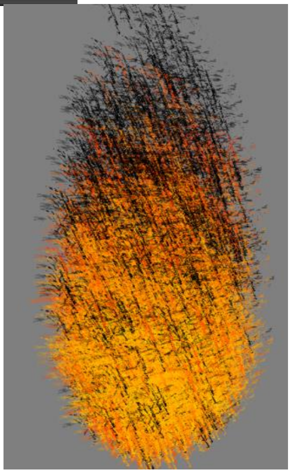
```
glGenBuffers(1, &texVBO);
glBindBuffer(GL_ARRAY_BUFFER, texVBO);
glBufferData(GL_ARRAY_BUFFER, /*3000*/ 4 * 80 * sizeof(glm::vec2), texture, GL_STATIC_DRAW);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(2);
```

next step is VAO, which was briefly mentioned in the previous section. Since quads are being rendered rather than points, VAO should be used here to make the program more efficient. Then, texture-mapped quads are drawn on the screen using GL_TRIANGLE_STRIP. Here are the actual texture that was



```
glDrawArrays(GL_TRIANGLE_STRIP, j, 4);
```

used and the final output of the improved GPU Based particle system to render a fire simulation using texture mapping. Although the new fire simulation does not look so much better than the previous implementation in this picture, the improvement look much clearer in the actual simulation.

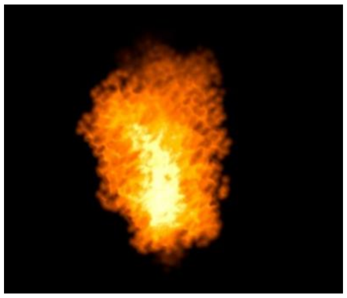


HONEST REVIEW & DIFFICULTIES

This project was all about building a GPU-based particle system from scratch in order to learn how the OpenGL works and how to utilise the functionalities that OpenGL provides. As someone who had not had any experience with OpenGL at all, choosing a project topic that requires a lot harder calculation and more detailed knowledge of OpenGL could have ended up causing me a big trouble, possibly not being able to finish the project at all. In that sense, I would say that this project topic is fair enough. In addition, I would like to add a few more comments about this project to discuss difficulties/challenges that I have had.

GPU Based Particle Systems

To begin, the most difficult part during the implementation was to figure out how to move my CPU-Based particle system onto the GPU to improve the performance. Initially, I had thought OpenGL is a library that merely helps programmers to develop any graphics technology. I did not understand how OpenGL API interacts with a GPU to achieve hardware accelerated rendering. More precisely, it was very difficult to understand how the OpenGL pipeline works, from the initialisation(glGen), binding(glBind) and rendering(glDraw). Furthermore, the concept of VBO and VAO were also very vague to me even though it was discussed in the class. By doing some further research and reading articles, I was able to understand how the different architectures between the CPU and GPU would improve/influence the performance of the particle system. I would say that I had to spend at least 20~30% of the time just to understand the OpenGL pipeline and how to use it to implement a GPU-based particle system. Furthermore, there was a couple of limitations to render a good fire simulation using the texture-



mapped particle system. Initially when I first read about the texture mapping from "Building a Million Particle System", I was expecting more realistic fire simulation such as the image on the left hand side. As you see, it looks more natural with good transparency, good depth of view and colour changes. I was not able to render something like this, and I suspect there has been a few issues with my implementation. First, I have not implemented a proper algorithm that determines the colour variation of the fire as it rises. As described in the previous section, my implementation changes the colour of the fire as the particles move upward. Thus, most of the particles would actually have quite a uniform colour variation even though each of the particles' lifespan is different. However, a proper implementation would be partially randomising the colour variation depending on its velocity, acceleration and the position from the fire root. Second, velocity and acceleration are randomised so that they determine more random and natural-looking fire simulation. Third, there should be multiple gravity points where the particles are attracted to. In my current implementation, I have only one constant gravity to the north of the fire root, which forces the fire to rise quite unnaturally, only upward. Setting multiple gravity points with random values for each frame should help rendering much smoother and natural looking fire simulation. Fourth, using texture-mapped objects have a limitation in 3D rendering. Since this texture-mapped objects are rendered as a single 2D quad, it was not able to render the fire properly if camera was at a different angle. A good solution to fix this problem would be generating a 3D tri-mesh for each particle so that each particle would be a 3D object. However, most of the game engines still use this 2D quad way, but rendering them at many different angles to allow users to see a proper fire at any angle. The problem with 3D object way is the performance. It is quite expensive to create 3D texture mapped object for each and every particle.

Overall, I would honestly rate my project as 80~85% of completion simply because the topic was not a relatively new to others colleagues who have picked very recent technique. I believe my implementation is about 85~90% as per the sources that I have referred to. Although there are some improvements that could be made further using more recent techniques, these techniques were not something that was present in these two main sources. I think this project was very helpful for me to learn a lot more understanding in how the graphics work and how to use OpenGL to utilise the GPU.

Bibliography

- Ginman, V. and Malmros, K. (2013). Visualization of smoke using particle systems. Retrieved fromhttp://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand13/Group9Petter/report/Veronica.Ginman.Kim.Malmros.report.pdf.
- Latta, L. (2004). Building a Million Particle System. Retrieved from https://pdfs.semanticscholar.org/ec64/9d2d46990e72f23606a8376101484fcc0d4a.pdf.
- Ogldev, (2015). Particle System using Transform Feedback. Retrieved from http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html.
- Scharf, P. (2015). CPU vs GPU particles – from 20 to 200 FPS. Retrieved from https://www.gamasutra.com/view/feature/130535/building_a_millionparticle_system.php?print=1.