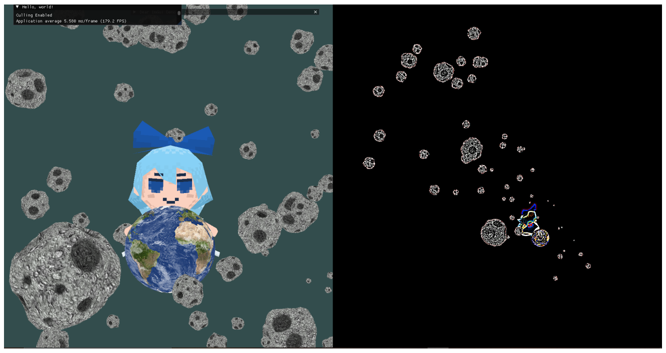
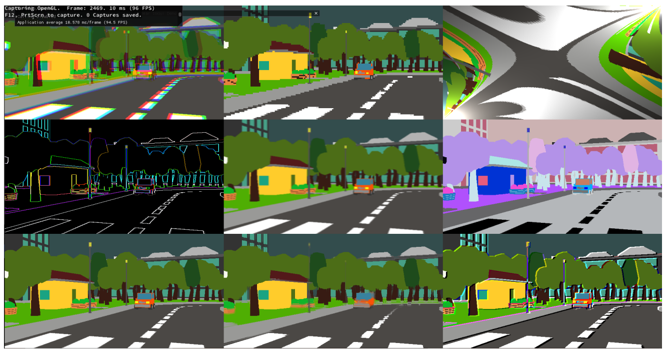
AlienGLRenderer: A simple 3D OpenGL renderer for simple scenes

Arrian Chi

***Abstract*— AlienGLRenderer is a renderer, written in C++ and OpenGL, designed to support the creation of simple scenes (inspired by the Quake Engine). The overall goal of this project is to streamline the rapid creation of simple scenes for artistic expression and to showcase evidence of technical knowledge. In this report, I showcase what features have been implemented, the technical challenges faced, and the future work to improve the renderer.**

I. Overview

Computer graphics is a field of computer science involved in the manipulation of data to generate visual content on a computer. Some may call it esoteric in the way that it combines a variety of paradigms from other fields of study, such as math, optics, and physics, for the overarching goal of producing purposeful images on a screen. Its applications in the film industry, the video game industry, and even in scientific visualization prove its importance in our society.

AlienGLRenderer is a hobby project implemented in OpenGL and C++ that I started in the summer of 2024 in hopes of pursuing a career in computer graphics. The project was inspired by the Quake Engine, a game engine developed by id Software in the 1990s. The engine was known for its simplicity and ease of use, which made it a popular choice for modders and indie developers. I wanted to create a similar engine that would allow me to experiment with graphics features and improve my skills in C++ and OpenGL, and also enable me to create simple graphics scenes in a easily configurable format.

In this paper, I would like to demonstrate the knowledge I have attained from the genesis of the project till now. I will discuss the process of the implementation of the renderer, the technical challenges I have faced through the course of its creation, and what future lies ahead for this renderer (including any remarks I have about OpenGL in the modern age).

A. Design

In the past, I have used OpenGL in a few other projects. The issue I found with these projects was that they were 1) too unoptimized and 2) hard to maintain. A few common tropes I found in these projects was encapsulation of rendering logic to the objects to be rendered (that is I implemented a render function for every cube, every sphere, etc.) and the frequent rebinding of buffers, shaders, and assets to render different objects (a buffer was initialized for every rendering primitive...). These issues arose from the blind usage of code from tutorials and introductory classwork and the misconception that object-oriented programming was the best way to conceptualize code.

The OpenGL API can be understood as a state machine that uses its current state to perform rendering tasks [1]. The user must bind state (buffers, shaders, textures, etc. all represented by an ID) to the OpenGL context before performing draw calls. One may conceptualize it like a switchboard, where the operator (the programmer) must switch the connections (the OpenGL state) to get the desired output (the rendered image). The reason the API is designed this way is a result of the history of graphics hardware (which could be expanded upon in its own paper). Simply put, graphics hardware started out with using fixed function pipelines that were highly customizable but required manual configuration [2]. The OpenGL API served as a high-level abstraction implemented by hardware vendors to allow programmers to configure the hardware without needing to know the specifics of the hardware.

So naturally, if one tries to religiously use object-oriented paradigms to encapsulate rendering logic, they would end up with a lot of overhead. For example, let us imagine we have a scene with many cubes, each having their own transforms. The naive programmer probably do this: for each object, bind its mesh data and transformation matrices to the OpenGL context. Then, issue a draw call to the GPU.

Note all this data lives in CPU RAM and must be copied to the GPU for it to be used. But CPU to GPU memory latency is relatively high. So if you are sending data to the GPU individually for every object, you are waiting for the GPU to finish rendering the object before you can send the next object. The CPU would be stalling while waiting for the GPU to finish rendering, resulting in significant overhead and bottlenecks.

Luckily, the OpenGL API permits the batching of state to minimize draw calls and memory transfers. In our example, the programmer would recognize that all cubes used the same mesh but had different transformations, so they instance all the cubes with the same mesh data and send every transformation matrix at once in a large matrix buffer to the GPU. In the vertex shader, the appropriate transformation matrix would be indexed and used for world space to model space transformations. This would reduce the number of draw calls and memory transfers, thus reducing the overhead.

The aggregation of state to reduce high latency computation is a common optimization technique in computer graphics and is used in many modern game engines. However, it requires that the architecture of the renderer be data-oriented (separating logic and data) rather than object-oriented (combining data and behavior). Thus, I decided that the renderer would be designed such that it knew about how to render every object primitive that could be renderered in a scene. This provides me with a lot of flexibility in how I can render objects and manipulate draw calls, which I will demonstrate when I discuss the features of the renderer. Note that I am not completely abandoning object-oriented programming, but rather thinking more about the data the application is manipulating.

II. Features Implemented

A. Renderer Mainloop

As mentioned before, I did not completely abandon object-oriented programming in the development of this renderer. The `Renderer` object itself contains all the data and logic needed to render a `Scene` object. A `Scene` object is a collection of data structures containing all the objects in a `SceneData` object that the renderer may directly read and use when rendering. A `SceneData` object contains that holds the uncompressed data read from a `.scn` file. Finally, `.scn` files are raw map data files (with a similar format to the Quake map format [3]) that contain all the scene objects and their properties. The static class containing the logic to load a `SceneData` from a `.scn` file and a `Scene` from a `SceneData` is called the `SceneLoader`.

To explain the algorithm of the render loop, the user supplies a path to the `.scn` file they want to render, which gets converted into a `Scene` object. The `Renderer` object is created and initialized with the size of framebuffer textures the renderer will render onto (more on this when discussing post-processing). While the window is open (the user has not closed the window), the application will first update the scene (which includes updating the camera, updating the cloth simulation, etc.), then call `renderer.render(scene)` for every renderer initialized. The application will then take each renderer's framebuffer texture and render it onto the main framebuffer texture. After that, the application will render any UI elements (using ImGui) on top of the main framebuffer texture. Finally, the application will swap the buffers, poll for events, and repeat the process.

B. What is done in the `renderer.render(scene)` call?

The `renderer.render(scene)` call is the most important call in the mainloop. It is where the renderer will render the scene onto the framebuffer texture. The renderer will first update the camera (if the camera is not locked), then update the cloth simulation (if the cloth simulation is enabled). The renderer will then iterate through all the rendering tasks in the scene and render them.

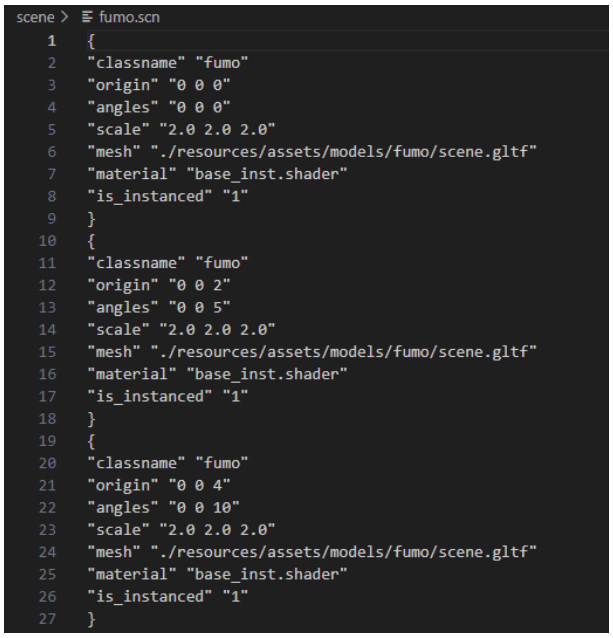
C. Direct State Access (DSA)

I previously stated that the OpenGL API is a state machine that requires the programmer to modify it to perform rendering tasks. In the past, this was not only limited to drawing, but also the manipulation of data. The programmer would bind the buffer to the global context, do whatever modifications they needed to the buffer, and then unbind it when not in use [4]. This is no different than being limited to a set number of global registers you are forced to use to modify your data. This could quickly become an issue because logic in different parts of the application must share the same global state in order to modify data in their local scope. This is a big problem especially if the app is organized in an object-oriented fashion.

Direct state access is a feature introduced in OpenGL 4.5 that allows the programmer to modify OpenGL objects without needing to bind them to the global context. . The relevant calls to this are `glCreateBuffers`, `glNamedBufferStorage`, and `glNamedBufferSubData`, which creates a buffer, allocate memory for the buffer, and modifies the data for the buffer, respectively. Semantically, these calls treat each buffer as its own piece of data, independent of the global state. All that is needed is the buffer's name, which is generated by `glCreateBuffers` to be used by other calls to identify the buffer of interest to use. In AlienGLRenderer, I use DSA for all data buffers (vertex, index, uniform, frame, etc.) to reduce the overhead of binding and unbinding buffers for every object to be rendered.

D. Scene Loading

AlienGLRenderer's scene loading was inspired by that of the Quake Engine and its relatives [3]. Fig. 1 shows an example file format of the `.scn` file. The format is pretty straightforward; every object in the scene stores its data as a series of key value pairs (kvps). The currently supported kvps include `classname`, `origin` (position), `angles`(rotation), `scale`, and `material`(currently just a shader name). The most important key is the `classname`,


scene > fumo.scn

```
{
"classname" "fumo"
"origin" "0 0 0"
"angles" "0 0 0"
"scale" "2.0 2.0 2.0"
"mesh" "./resources/assets/models/fumo/scene.gltf"
"material" "base_inst.shader"
"is_instanced" "1"
}
{
"classname" "fumo"
"origin" "0 0 2"
"angles" "0 0 5"
"scale" "2.0 2.0 2.0"
"mesh" "./resources/assets/models/fumo/scene.gltf"
"material" "base_inst.shader"
"is_instanced" "1"
}
{
"classname" "fumo"
"origin" "0 0 4"
"angles" "0 0 10"
"scale" "2.0 2.0 2.0"
"mesh" "./resources/assets/models/fumo/scene.gltf"
"material" "base_inst.shader"
"is_instanced" "1"
}
```

Fig. 1: An example of the `.scn` file format

which is used to identify the object (i.e. a cube, an entity, a camera, etc.), which indicates how it should be processed.

E. Instancing

Briefly covered in the previous section, GPU instancing is the idea of drawing multiple copies of the same model mesh with slightly varying data all at once (i.e. blades of grass with varying heights/positions). In terms of draw calls, GPU instancing reduces several draw calls of the same mesh into one single draw call ([5]). The benefit is that we reduce the expensive CPU to GPU communication overhead and the CPU stalls that come with it.

In the scene, entities (any object with a `classname` that does not have special handling) are the only objects that are instanced. It is your typical 3D object rendered into a scene consisting of a mesh, a shader, and a transform. The `SceneLoader` uses the concatenated mesh path and shader path as the key for an `entityInstanceMap` (really, should be called an `entityInstanceRenderTaskMap`). Each key is associated with an `EntityInstanceData` (should be called an `EntityInstanceRenderTask`), which aggregates all the entities with the same mesh and shader into a single rendering task. Each rendering task contains the instance count, a buffer of all the transformation matrices for each entity, the model data, and the shader data. The model data contains all the necessary meshes, materials, textures, and draw commands (more on this later) required to render its own mesh. So when the renderer is rendering the scene, it will iterate through all the rendering tasks and issue the draw calls with appropriate instancing data (i.e. the instance count and instance buffers). At first, this was done with `glDrawElementsInstanced`, but the call was later changed to `glMultiDrawElementsIndirect` to accomodate for indirect draws.

F. Model Loading

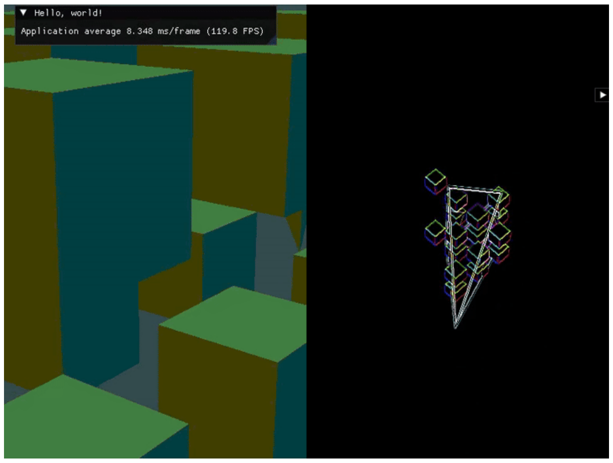
AlienGLRenderer currently only accepts `.glTF` files (graphics library transmission format) ([6]) as a valid model file. It leverages the fastgltf library to parse the `.glTF`. I decided to use the glTF file format because it stores data in the JSON format and use indices to reference data organized in the flat hierarchy ("scenes refer to root nodes by index, nodes refer to meshes (which are composed of primitives), which refer to materials, which refer to textures, which refer to images"). A static class named `ModelLoader` is responsible for loading the `.glTF` file and converting it into a `Model` object, creating the buffers for each category of data, as well as their indices. One key point to note is that every single `Model` is associated with a buffer of draw commands for each primitive (individual parts of a mesh) (more on this in the next section), the transforms of all nodes in the model, and a buffer of node primitive properties containing indices for the draw command to index the primitive's transform (that of the node their mesh is indiced in). (This really needs a figure). These draw commands are issued by `Renderer` per `EntityInstanceData` in the `Scene` when rendering, preserving the original transform hierarchy of the `.glTF` model.

G. Indirect Draw Commands

In pursuit of better performance (from both instancing and frustum culling), AlienGLRenderer uses indirect draw commands (`glMultiDrawElementsIndirect`) to render entities. When using indirect draw commands, draw calls are abstracted as command structs that contain parameters for the draw call (i.e. same parameters as `glDrawElements`). The command structs are stored in a buffer and sent to the GPU, and when `glMultiDrawElementsIndirect` is called, the GPU will submit draw calls for each struct in the buffer. This feature opens up the possibility of parallel draw command manipulation, either on the CPU (each thread can write to the command buffer) or the GPU (use the draw command buffer just like any buffer in a compute shader). In AlienGLRenderer, the latter case is showcased in the frustum culling feature to set the instance counts of every draw command in the buffer for an `entityInstanceData` struct. Every model has its own draw command buffer (a draw command for each primitive to be rendered) that stays on the GPU (no CPU readback) for the duration of the application's lifetime.

H. Frustum Culling

If rendering is the bottleneck of an application, it is common to reduce the amount of objects to be rendered using a culling algorithm, which filters draw calls based on the visibility of the primitives relative to the camera before they are issued to the GPU. There are 3 noteble culling algorithms: back-face culling (culling the polygons whose points are in the opposite winding order i.e. if counter-clockwise winding is front-facing, then all polygons with



```
▼ Hello, world!
Application average 8.348 ms/frame (119.8 FPS)
```

Fig. 2: A visualization of frustum culling

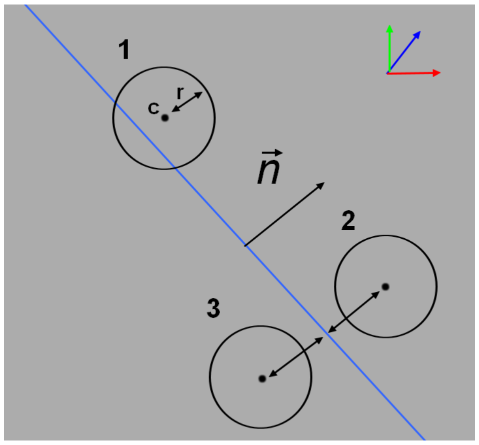


Fig. 3: A sphere may be inside the plane, forward, or backward. Compare the signed distance from the sphere's center to the plane check collision with the frustum.

clockwise winding are culled), occlusion culling (culling primitives that are hidden, or occluded, behind other visible primitives closer to the camera), and frustum culling (culling primitives that are outside the view frustum of the camera). AlienGLRenderer implements frustum culling (see Fig. 2), and in the sections below, I will discuss my process of implementing it.

1) CPU Frustum Culling: CPU frustum culling implies that the draw calls are filtered by the CPU before they are issued to the GPU. Because this was implemented before instancing and indirect draw calls were added, the implementation process was easy. First, bounding volumes (collision geometry) were specified for each object in the scene. In the `Model` class, a bounding sphere with a radius equal to the distance of the model's furthest point from the origin is specified. Each camera is associated with a frustum defined by 6 planes (a normal and a distance). The `Renderer` would use the collision geometry to check if the bounding sphere of the object is partially inside the view frustum of the camera (check the signed distance of the sphere's center to the plane is greater than the negative radius of the sphere for every plane, i.e. see Fig. 3). If it is, the draw call is issued, otherwise don't do anything. This process was repeated for every object in the scene every frame. An issue I have with this process is that the check was sequential (iterating through every object in the scene). I wanted to see if it is possible to parallelize this process and also incorporate it with instancing and indirect drawing.

2) GPU Frustum Culling: I had two versions of GPU frustum culling, one unoptimized and another that was. Both incorporated instancing and indirect draws in their implementions. To start, the naive approach required a buffer containing a boolean for every instance in the scene to store the visibility of that object. A compute shader would execute the collision check for all objects (in parallel) and set their corresponding visibility bit. In the vertex shader, the index of the instance (`gl_InstanceID`) would be used to index the visibility buffer to check if the vertex is visible. If it is, the transform of the instance is obtained and rendering goes on as usual, but if not, the vertex would be discarded. While the scene visually shows frustum culling is implemented, this process does nothing to help the performance of the renderer because the draw calls of the invisible instances are still issued to the GPU and executed.

The issue in the first version is that the we are using instancing for the number of instances in the scene instead of the number of the instances visible in the view frustum. The second version of GPU frustum culling takes care of that by atomically accruing the count of visible instances in the culling compute shader. Instead of a buffer of booleans, it uses a buffer of indices corresponding to those of visible instances. After the buffer is computed, another shader sets the instance counts for every draw command struct in the draw command buffers of the associated model. Finally, the indirect draw call is issued to the GPU, and like before, the vertex shader will use the `gl_InstanceID` to index the visibility buffer to obtain the transformation matrices of the instance, only this time, only the visible instances have their draw calls issued.

I. Texture loading

Textures in the `.gltf` file format are applied per primitive of a mesh. Because of the way instancing and indirect draw batch all the mesh draw calls together and the fact that OpenGL doesn't permit the user to create an array of textures (that would be way easier to implement because the fragments would select which texture they are using), the next best way to reimplement textures was to use a texture atlas. This required the model loader to calculate the size of all textures and pack them into a single texture. Additionally, some additional computation is required to recalculate the texture coordinates of each vertex on the model. I opted to do this calculation on the GPU because it was more intuitive for me (better interface to program with). But one may also recompute the same texture coords on the CPU so that this calculation is only done once.

J. Multiple Cameras and Render Targets

Before continuing, I should explain the architecture of the `Renderer` class itself. To begin, the `Renderer` class contains all the vertex array objects (objects that the OpenGL context use to determine the layout of the vertex stream) for each object type, the compute shaders for rendering operations (e.g. cull shaders), the uniform buffers available to every shader (e.g. camera view projection matrices) and the framebuffer textures the renderer must render onto. The renderer contains the logic to bind the appropriate data of each type of object to the OpenGL context and issue that call. The renderer also contains logic for any post-processing effects that should be applied onto the renderered image. As mentioned before, in the application mainloop, the app will blit the framebuffer textures onto the main framebuffer texture (adjacent to each other horizontally) after the render call is completed for every `Renderer` object. I am aware it may be better to have the `Renderer` specify where on the main framebuffer to write to, but I have not implemented that yet.

This design makes it easy for me to add multiple camera views to the scene. Each camera's data lives a static vector in the `Scene` object, so if we have two `Renderer` objects, each indexing a different camera in the scene, we can render the same scene with two different perspectives and display them side by side. In addition, each renderer can add their own post-processing effects to their framebuffer textures, so the user can see the scene in different ways. It is also possible to apply these framebuffer textures onto objects in the scene (i.e. a real-time security camera monitor).

K. Post-Processing

Post-processing is when you take a rendered image (the process) and apply effects(shaders) to it (the post-process) before displaying it on the screen. As mentioned in the previous section, this is made possible through framebuffer textures in each `Renderer` object. Specifically, `Renderer` has a source texture and a destination texture. When `.render()` is called, it writes to the source texture when going through the scene. As the final step, the source texture is rendered onto a quad, the post-processing shaders are binded, and the output is written to the destination texture, which will be blitted onto the main framebuffer. To showcase the feature, I implemented a very simple edge-detection shader using kernel operators that highlights the edges of the primitives in the image.

L. Particle System and Cloth Simulation

For the final feature to date, and for the final project of a mechanical engineering class, I implemented a cloth simulation using the Discrete Elastic Plates / Shells algorithm[7] [8]. This system was implemented on top of a base particle system interface. To me, on the basic level, a particle system is an object that manages a collection of particles (objects with a mass, position, and velocity), usually through the application of forces. At every frame of the renderer, the forces are calculated and the positions are updated and rendered. The forces also adhere to hard set of rules that help create a cohesive system.

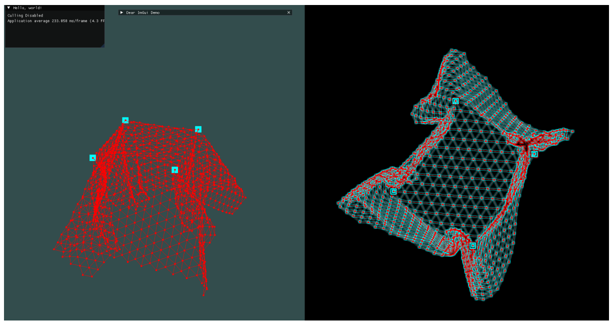

```
▼ Hello, world!
Culling Disabled
Application average 233.058 ms/frame (4.3 FP
```

Fig. 4: A cloth afixed as if it were a tablecloth.

This broad definition allows me to lump in many seemingly different effects together. For instance, streaks, stars, and smoke particle effects are no different than boids[9], fluid[10], and cloth simulations. All these systems are governed by a set of rules that dictate the behavior and interaction of the particles. The difference is what set of rules each abide by.

For the cloth simulation (see Fig. 4), the rules are derived from the DEP/S algorithm. The algorithm is a numerical simulation algorithm that simulates the behavior of a cloth as a series of connected plates and shells (see Algorithm 1). In short, The cloth is discretized into a mesh of particles, each connected to its neighbors by edges. Elastic energy is concetrated throughout the mesh as stretching energy (from two particles pulling on each other through an edge) and bending energy (through a set of 4 particles forming a hinge to rotate about). External forces act on every particle at every frame, which provide the potential energy for the cloth to move. AlienGLRenderer implements a version of the DEP/S algorithm that uses implicit integration (Newton-Raphson method) to solve for the next position of the particles, for accuracy purposes. However, it is possible to instead use an explicit integration method (like Euler's method) for speed purposes. Additionally, it leverages the Eigen library [11] to do matrix operations (OpenGL and glm does not provide functions for matrices greater than size 4) and the PARDISO library to solve the matrix system [12] [13] [14]. In terms of rendering, the cloth is treated as a mesh of particles rendered as a lines connecting points. The fixed points in the simulation are highlighted with a different color to show the user where the cloth is anchored.

III. Issues

In the course of development of this project, I encountered a few issues that I would like to discuss. These issues that may have hindered my progress and/or still exist in the project. Regardless, I am aware of their existence and am currently figuring out whether I should solve them or not / how to solve them.

A. Subtleties in OpenGL

I was caught up in a few subtleties in OpenGL. To start, OpenGL memory alignment rules that make working with vec3s harder than it should. In the cloth simulation, the DEP algorithm assumes that the DOF vector is a $3 \times n$ by 1 column vector, where n is the number of vertices. My initial naive implementation consisted of an array of 3 by n floats to represent the DOF vector used for calculating the forces and use the same DOF vector for drawing (specifying the vertex format to assume a vertex stream of vec3). This would work if I didn't use my DOF vector in the other (compute/debug) shaders (specifying the DOF vector as a shader buffer storage object (SSBO)).

However, according to the OpenGL specification, when using SSBOs and the std430 layout, a vec3 is aligned to 16 bytes (4 floats) in GPU memory. This means that in the shader, when I try to index into an array of vec3s, the GPU will retrieve the result with 3 floats plus 1 byte of padding. So when I copy the chunk of buffer data (the DOF vector) from CPU memory into GPU memory, I had to ensure that the total memory size of the DOF buffer is a multiple of 16 bytes. Otherwise, the GPU would read 3 floats from the buffer, skip one float part of another point, and then read the next 3 floats. This means that after my force calculation, I had to create a way to send the data to the GPU such that my buffer acted as a vector of vec4s. In other parts of the application other than the cloth simulation, you may also see me pack a struct consisting of a vec3 and a constant into a single vec4 for this reason (`B`oundingVolume structs in Plane and Sphere).

There is another solution to this issue without aligning the DOF vector to 16 bytes. Instead of having the GPU read the buffer as an array of vec4s/vec3s, I could pass the buffer to the GPU as an array of floats (avoiding the alignment issue entirely). Then the vertex stream for rendering for the shaders (specifically the highlight shader since the DOF positions must be read as a SSBO instead of a vertex stream) would be the indices of the particles themselves. In the highlight shader, I could use the particle indices to compute the indices of the components of each DOF for that particle.

Another subtlety is numerous amount of calls that look-alike, but do very different things. The DOF vector in the cloth simulation is an array of double precision floats (for accuracy and precision). In the beginning, I thought, for rendering, the vertex attribute format was specified with `glVertexAttribFormat` using the type parameter `GL_DOUBLE`. However, the correct call used to pass an array of doubles in as a vertex stream is `glVertexAttrib`**L**`Format` [15]. Of course, the type cast would still need to be done in the vertex shader since the vertex shader output for positions (`gl_Position`) is specified as a float type. I unfortunately realized this too late (it's a one character difference!), and therefore it is not implemented.

Finally, I initially proposed to use OpenGL compute shaders to do the cloth simulation on the GPU. The DEP algorithm (save for the matrix solve) is embarassingly parallelizable, so it would be a good candidate for a compute shader. However, OpenGL does not support matrix operations beyond 4 by 4 dimensions, which means I would need to implement this myself. The conjugate gradient solve [16] is also a parallelizable operation, but this requires significant time to debug (and GPU debugging is much different than CPU debugging since there is no way to step through programs). I concluded that this is a huge undertaking, so I decided to stick with the CPU implementation.

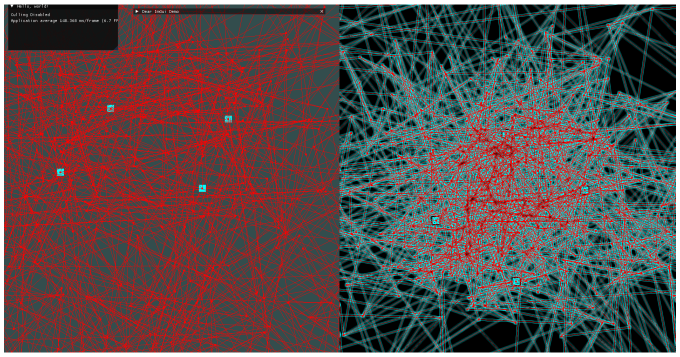
B. Monotlithic rendering architecture

As more types of objects are supported by my renderer, I found the architecture of the renderer to be monolithic. This is natural since the renderer contains all the logic for rendering every single object. However, this makes adding new features to the renderer difficult. For instance, when adding support for rendering particle systems, the renderer must be have a new vertex array object for it and new logic for the rendering. I had to scroll through lots of code and make sure that my particle system rendering did not intefere with the rendering of other objects. A more pronounced example is the support for instancing and indirect draw calls with culling. Because of the way each call is structured and the way data is supplied, I had to disable materials and non instanced rendering when debugging the application. This does make sense though because these features are involved in the method of rendering objects and are core to the rendering algorithm. But if anything, a lesson learned here is echoed by a central idea: there are no zero cost abstractions [17]. Because I decided to make my renderer data-oriented, I now burden myself with the explicit handling of data and all the rendering logic in the renderer.

C. Lack of oversight

If one were to read the code for the rendering, they may get confused about why I made a few questionable decisions. For starters, I stopped implementing destructors at one point. This is because I assumed that the lifetime of my objects were the same as the lifetime of the application. This is a bad assumption to make because it is possible that the application may need to be extended to support more features, and the objects may need to be destroyed and recreated. This is especially true for the cloth simulation because it makes it impossible for me to reset the simulation without restarting the application. Because I didn't implement destructors for the cloth simulation objects, if I did delete the cloth simulation objects, there would be memory leaks (from the opengl objects not being deleted). After realizing this, I swear would never do this in production code, nor again.

Another issue is the inconsistent usage of OpenGL types. This isn't a problem as long as my assumptions about what sizes the types I'm using align with the ground truth, or in other words, I support only a single platform. But it is important to note that C++ (as well as OpenGL) are specifications that are implemented on the platform they are running on. A `GLuint` is guaranteed to be 32 bits across all



```
Culling Disabled
Application average 148.368 ms/frame (6.7 FP
```

Fig. 5: A simulation of a cloth with 576 nodes that has reached a degenerate state

systems, but an `unsigned int` may sometimes be 64 bits or 32 bits depending on your platform's C++ implementation. This isn't a huge issue, since the intended main user of my renderer is only myself, but it is important to think about when writing production code.

D. Degeneracies in the Cloth simulation

It is known that the DEP/S algorithm is prone to some degeneracies, such as when an edge collapses or when the altitude collapses [18]. This may manifest int the calculation as a division by zero error during the solve, an infinite loop (the error never reaches the desired tolerance), or the cloth visually becomes a cloud of points (see Fig. 5). These may arise when the solver is unable to reach a valid solution. This is more common in simulations with more DOFs as there is more freedom for numerical instability to occur. In my own tests, the most nodes I found that would reach the resting state without reaching numerical instability is 529 nodes (1575 DOFs, side length = 22). Of course, numerical instability could be mitigated by using a smaller timestep, but this would slow down the simulation (more time steps per second and more work to compute the next position at the next second). Another potential solution would be to check the solver solution before it updates the DOF vector. If the solution is degenerate, special care can be taken (i.e. halve the time step used for the rest of the iterations). Regardless, this problem requires more investigation to find a more robust solution.

IV. Future Work

In addition to resolving the issues above at some point in my life, I have a few more ideas I believe the renderer would benefit from. These ideas are the following:

- Implement a more robust particle system
- Implement occlusion culling, LOD systems, and other culling algorithms
- Use a triangle instead of a quad for post-processing (slightly more efficient)
- Implement shader graphs for post-processing
- Implement lighting, shadows, and reflections
- Implement physically based rendering and deferred shading
- Screen space reflections, ambient occlusion, and global illumination
- Rename the variables to be more descriptive
- Use a compute shader for the cloth simulation
- Implement raytracing for the renderer
- Use Vulkan

Each of these bullet points are a feat on their own (I am aware having all these features is overly ambitious), but I think of most interest would be the last bullet point. Vulkan [19], a graphics API developed and managed by the Khronos Group, is a verbose API that exposes much of the GPU's capabilities to the programmer. It is a low-level API that provides more control over the GPU pipeline, which opens up opportunities for reducing CPU overhead and optimizations. Vulkan is also a more modern API than OpenGL, and it is more likely to be supported by future hardware. But most importantly, it (along with DirectX 12, Metal, and WebGPU) are APIs that have been taking off in the industry as OpenGL is being deprecated from platforms like Apple [20]. I believe exposing myself to a more verbose API would further my understanding of graphics hardware more than OpenGL ever could and make me a more competitive candidate for a graphics programming position. Thus, I believe for now, the next step in my renderer may be to future proof my renderer by porting it to Vulkan.

V. Conclusion

In this report, I have discussed the design, implementation, and features of my capstone project, AlienGLRenderer. I have highlighted the issues I encountered during its development and what future work lies ahead, including expressing my desire to port this renderer to Vulkan. The project has been a valuable learning experience, allowing me to deepen my understanding of computer graphics and OpenGL. Moving forward, I aim to address the identified issues, implement additional features, and explore more advanced graphics APIs to further enhance the capabilities of AlienGLRenderer and my knowledge.

Algorithm 1 Discrete Elastic Plates

```
Require: $q(t_i), \dot{q}(t_i)$                                      DOFs and velocities at $t_i$
    $e(m), h(n)$                                     m edges and n hinges present in the plate
    $m, M$                                   masses of each particle (as a vector and matrix)
    $l_k, k_s$                                     undeformed length / stretching for each edge
    $\bar{\theta}, k_b$                                   rest angles of each hinge, bending stiffness
    $\Delta t$                                                                       time step
    $\mathbf{f}^{\mathbf{ext}}$                                                             external forces
    free_index                                                       Index of the free DOFs

Ensure: $q(t_{i+1}), \dot{q}(t_{i+1})$                            DOFs and velocities at $t = t_{i+1}$

 1: function COMPUTE_STRETCHING($q, e, l_k, k_s$)
 2:     $\mathbf{f}^{\mathbf{stretch}}, \mathbf{J}^{\mathbf{stretch}} \leftarrow 0, 0$                                        Initialize
 3:     for $i \leftarrow 1$ to $m$ do
 4:         x0, x1 $\leftarrow q(e(k,1)), q(e(k,2))$                     Get DOFs of particles in edges
 5:         $\mathbf{f}^{\mathbf{grad}}, \mathbf{J}^{\mathbf{hess}} \leftarrow$ GRADES_HESSES(x0, x1, $l_k(i), k_s(i)$)       Refer to Appendix of [21]
 6:
 7:         $\mathbf{f}^{\mathbf{stretch}}(e(k)) \leftarrow \mathbf{f}^{\mathbf{stretch}}(e(k)) - \mathbf{f}^{\mathbf{grad}}$                 Update total force/Jacobian
 8:         $\mathbf{J}^{\mathbf{stretch}}(e(k), e(k)) \leftarrow \mathbf{J}^{\mathbf{stretch}}(e(k), e(k)) - \mathbf{J}^{\mathbf{hess}}$
 9:     return $\mathbf{f}^{\mathbf{stretch}}, \mathbf{J}^{\mathbf{stretch}}$
10:
11: function COMPUTE_BENDING($q, h, k_b$)
12:     $\mathbf{f}^{\mathbf{bend}}, \mathbf{J}^{\mathbf{bend}} \leftarrow 0, 0$                                              Initialize
13:     for $i \leftarrow 1$ to $n$ do
14:         x0, x1, x2, x3 $\leftarrow q(e(k,1)), q(e(k,2)), q(e(k,3)), q(e(k,4))$   Get DOFs of particles in hinges
15:         $\mathbf{f}^{\mathbf{grad}}, \mathbf{J}^{\mathbf{hess}} \leftarrow$ GRADEB_HESSEB(x0, x1, x2, x3, $l_k(i), k_s(i)$)   Refer to Appendix of [21]
16:
17:         $\mathbf{f}^{\mathbf{bend}}(e(k)) \leftarrow \mathbf{f}^{\mathbf{bend}}(e(k)) - \mathbf{f}^{\mathbf{grad}}$                     Update total force/Jacobian
18:         $\mathbf{J}^{\mathbf{bend}}(e(k), e(k)) \leftarrow \mathbf{J}^{\mathbf{bend}}(e(k), e(k)) - \mathbf{J}^{\mathbf{hess}}$
19:     return $\mathbf{f}^{\mathbf{bend}}, \mathbf{J}^{\mathbf{bend}}$
20:
21: function DISCRETE_ELASTIC_PLATES($q, \dot{q}$)
22:     Guess: $q^{(1)}(t_{i+1}) \leftarrow q(t_i)$
23:     $n \leftarrow 1$
24:     while error > tolerance do
25:
26:         $\mathbf{f}^{\mathbf{stretch}}, \mathbf{J}^{\mathbf{stretch}} \leftarrow$ COMPUTE_STRETCHING($q, e, l_k, k_s$)
27:         $\mathbf{f}^{\mathbf{bend}}, \mathbf{J}^{\mathbf{bend}} \leftarrow$ COMPUTE_BENDING($q, e, l_k, k_s$)
28:
29:         $\mathbf{f}^{\mathbf{tot}} \leftarrow \mathbf{f}^{\mathbf{bend}} + \mathbf{f}^{\mathbf{stretch}} + \mathbf{f}^{\mathbf{ext}}$                                  Aggregate forces
30:         $\mathbf{f} \leftarrow \frac{1}{\Delta t} m \odot \left[\frac{1}{\Delta t}\left[q^{(n)} - q^{(1)}\right] - \dot{q}\right] - \mathbf{f}^{\mathbf{tot}}$
31:
32:         $\mathbf{J}^{\mathbf{tot}} \leftarrow \mathbf{J}^{\mathbf{bend}} + \mathbf{J}^{\mathbf{stretch}}$                                       Aggregate Jacobians
33:         $\mathbf{J} \leftarrow \frac{1}{\Delta t} M - \mathbf{J}^{\mathbf{tot}}$
34:
35:         $\mathbf{f}_{\text{free}} \leftarrow \mathbf{f}$(free_index)
36:         $\mathbf{J}_{\text{free}} \leftarrow \mathbf{J}$(free_index, free_index)
37:
38:         $\Delta q_{\text{free}} \leftarrow \mathbf{J}_{\text{free}}^{-1} \mathbf{f}_{\text{free}}$
39:         $q^{(n+1)}$(free_index) $\leftarrow q^{(n)}$(free_index) $- \Delta q_{\text{free}}$                  Update free DOFs
40:         error $\leftarrow$ sum($|\mathbf{f}_{\text{free}}|$)
41:         $n \leftarrow n + 1$
42:
43:     $q(t_{i+1}) \leftarrow q^{(n)}(t_{i+1})$                                 Update DOFs for next time step
44:     $\dot{q}(t_{i+1}) \leftarrow \dot{q}^{(n)}(t_{i+1})$
```

References

[1] "Portal:opengl concepts." Available at https://www.khronos.org/opengl/wiki/Portal:OpenGL_Concepts, Sep 2017.

[2] J. Peddie, *The history of the GPU - eras and environment*. Springer International Publishing Springer, 2022.

[3] "Quake map format - quake wiki." Wiki available at https://quakewiki.org/wiki/Quake_map_format.

[4] J. Stephano, "Direct state access (dsa)." Available at https://ktstephano.github.io/rendering/opengl/dsa.

[5] J. d. Vries, "Instancing." Available at https://learnopengl.com/Advanced-OpenGL/Instancing, Jun 2014.

[6] T. K. D. F. W. Group, "Gltf 2.0 specification." Available at https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html, Oct 2021.

[7] D. Baraff and A. Witkin, *Large Steps in Cloth Simulation*. New York, NY, USA: Association for Computing Machinery, 1 ed., 1998.

[8] E. Grinspun, A. N. Hirani, M. Desbrun, and P. Schröder, "Discrete shells," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, (Goslar, DEU), p. 62–67, Eurographics Association, 2003.

[9] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *SIGGRAPH Comput. Graph.*, vol. 21, p. 25–34, Aug. 1987.

[10] J. Stam, "Stable fluids," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, (USA), p. 121–128, ACM Press/Addison-Wesley Publishing Co., 1999.

[11] G. Guennebaud, B. Jacob, *et al.*, *Eigen: A C++ template library for linear algebra*. Eigen Contributors, 2024. Accessed: 2024-11-20.

[12] D. Pasadakis, M. Bollhöfer, and O. Schenk, "Sparse quadratic approximation for graph learning," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 9, pp. 11256–11269, 2023.

[13] A. Eftekhari, D. Pasadakis, M. Bollhöfer, S. Scheidegger, and O. Schenk, "Block-enhanced precision matrix estimation for large-scale datasets," *Journal of Computational Science*, vol. 53, p. 101389, 2021.

[14] L. Gaedke-Merzhäuser, J. van Niekerk, O. Schenk, and H. Rue, "Parallelized integrated nested laplace approximations for fast bayesian inference," 2022.

[15] Khronos Group, *OpenGL Documentation*. Khronos Group. Accessed: 2024-11-20.

[16] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, p. 917–924, July 2003.

[17] C. Caruth, "There are no zero-cost abstractions." CPPCon 2019, 2019.

[18] R. Tamstorf and E. Grinspun, "Discrete bending forces and their jacobians," *Graph. Models*, vol. 75, p. 362–370, Nov. 2013.

[19] The Khronos Vulkan Working Group, "Vulkan 1.4.303 - a specification." Available at https://registry.khronos.org/vulkan/specs/latest/html/vkspec.html, 2024.

[20] J. Horwitz, "Apple defends end of opengl as mac game developers threaten to leave — venturebeat." Available at https://venturebeat.com/games/apple-defends-end-of-opengl-as-mac-game-developers-threaten-to-leave/, Jun 2018.

[21] S. MK Jawed, Lim, "Discrete simulation of slender structures," 2023.