A SIMULATION PLATFORM FOR TESTING SOLAR CAR AERODYNAMICS AND CFD BASED ON UNITY

Weiyue Yan [1] , Moddwyn Andaya [2]

[1] Pacific Academy Irvine, 4947 Alton Pkwy, Irvine, CA 92604
[2] California State University, Sacramento, 6000 Jed Smith Dr, Sacramento, CA 95819

ABSTRACT

This project develops a Unity-based simulation platform that helps students and beginner designers analyze vehicle aerodynamics without expensive wind tunnel testing [6]. Air resistance plays an important role in vehicle performance, but traditional aerodynamic testing methods require costly equipment and complex professional software. To address this problem, the project introduces an interactive simulation environment where users can upload 3D car models, select or upload maps, and simulate airflow conditions [7]. The system is built around three main components: Properties, CFD simulation, and Map. These components work together to control inputs, generate aerodynamic visualization, and manage user interaction. The CFD system uses a simplified real-time approach based on dynamic pressure and precomputed surface responses to provide stable and interactive feedback. During development, challenges such as path design, model scaling, and collision detection were addressed through physics adjustments and automated systems. Experimental evaluation compared the system against ANSYS Fluent, showing an average error of 5.5%, indicating that the platform provides reasonably accurate results while maintaining high performance and accessibility.

KEYWORDS

CFD Simulation, Spline, 3D modeling, Unity

1. INTRODUCTION

Student solar car teams often face lots of challenges when they try to improve their car. Air resistance is crucial for determining how efficiently a car can run. A small change of shape may cause a big difference in speed. However, testing air resistance is very expensive and hard for high school students. For example, a time wind tunnel test may be costly for the high-school solar car term, since it costs 200 dollars to 1000 dollars per hour [1]. This limitation forces students and terms to rely on the computer simulation or some rough estimates without real data. Even though there are lots of professional software for CFD that can simulate airflow and drag forces, it is usually expensive, complex, and not friendly for the fresh, which causes the beginner term may not test their idea and change their design during an early stage of development. For instance, Autodesk CFD, a famous CFD app, still needs the beginner to get training in using it, which is still not so friendly for fresh [2]. This problem is important since accessible engineering tools can help more people participate in vehicle design and innovation. By providing a simple CFD-based tool to estimate airflow and air resistance, students and beginner designers can better understand how aerodynamic changes affect vehicle performance. In the long run, it let the vehicle design

David C. Wyld et al. (Eds): WiMNeT, NLDM, EDUPT, SIP, AISO – 2026
pp. 31-41, 2026. CS & IT - CSCP 2026 DOI: 10.5121/csit.2026.1601004

not just be limited for those skillful engineers but all the people who were interested in it and it helps students and designers try new ideas and change their design.

The first method is CFD simulation, which can simulate the airflow around a 3D digital model. It allows engineers to test many designs without building physical prototypes. However, professional CFD software usually requires strong technical knowledge and powerful computers, which makes it difficult for beginners to use.

The second method is to calculate the drag by using the drag equation. This formula estimates air resistance based on air density, velocity, drag coefficient, and frontal area. It is simple and does not require advanced tools. However, the equation relies on simplified assumptions and cannot fully represent complex airflow behavior around real vehicle shapes.

The third method is using a wind tunnel to test the air flow of a vehicle. In this method, a scale model of a car is placed in a controlled circumstance to measure aerodynamic forces and observe airflow patterns. This way can provide very reliable data, but it is too expensive and requiresbuilding a physical model which will change design hard.

The solution is a Unity-based simulation platform that allows teams to test solar car designs under simulated airflow conditions in an interactive environment.

This platform aims to provide an accessible way to eliminate the drag force without the wind tunnel or other expensive equipment for the student term. Users can import models of their designs and the map. They can design a path of the car and observe the car will move. This platform can also simulate air resistance in different conditions, and the color-view can help users easily to check where the air resistance is great. This approach helps solve the problem because it allows teams to test ideas early in the design process. Rather than relying only on theory or guessing which design might work better, students can experiment with multiple versions of a vehicle and observe how each one performs in the simulation. This makes it easier to notice inefficient shapes and refine the design before building a real one. The solution is effective due to its focus on accessibility and ease of use. While professional CFD programs can give very accurate results, they are often expensive and require significant training to operate [8]. This platform can run on common computers and provides a visual, interactive environment that is easier for students to understand and check. Although the simulation may be simpler than industrial software, it still can provide meaningful feedback that helps teams compare designs and make better engineering decisions during the early stages of development.

The experiment evaluated the accuracy of the Unity-based CFD system by comparing its drag force outputs with reference data from ANSYS Fluent. The test was designed by selecting a streamlined car model and running simulations under controlled conditions, where air density and wind speed were fixed while vehicle speed varied from 10 m/s to 50 m/s. Drag values from both systems were recorded, and percent error was calculated for each trial. The results showed that the system maintained a relatively consistent error range, with a mean error of 5.5% and a median of 5.0%. The lowest error occurred at mid-range speeds, while the highest error appeared at lower speeds. This suggests that the system performs more reliably when airflow conditions are stronger. The main reason for the variation is the simplified surface-based model, which cannot fully represent complex aerodynamic effects such as turbulence and flow separation.

2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

2.1. Ensuring Realistic User-Designed Driving Paths

Letting users design the path of a car by themselves is one of the challenges we face. If the user cannot design a path, which will cause the car model cannot run in an uploaded map from users and cannot see the different path will cause what for the car. However, during this function, we face some questions like unrealistic paths, or something like that. For example, sometimes if we design a path which is in air it may cause the car to fly in the sky, so we introduce gravity on the car which letit stay on the road.

2.2. Scaling and Previewing Uploaded 3D Car Models

Making a 3D model preview of the Uploaded car for the user is a difficulty that we face. It is very important since it can check how the model works for this map. After we achieve this function, we find that we cannot make sure all the car-model fits the size, and usually the model is tiny or tremendous. In order to solve this problem, we add a function that lets people change the size of the vehicle manual. It cannot just let users easily check different sizes of the car, but also avoid the car is too huge for the map and covering the camera.

2.3. Automating Collision Handling for User-Generated Maps

To keep the car moving in track and not falling down to the vacuum is also an issue we have experienced before. This function is crucial for this platform, since if the car is usually going down, it means our app cannot work. So as to solve this problem, we manually add the collision box for the system map. It works but the issue is we cannot let users add collision for their map by themselves, since it costs time and adds to the operation difficulty. To solve that, we make a code which can add the collision box for uploaded maps automatically.

3. SOLUTION

There are three major components in the program: Properties, CFD, and Map. These parts are connected through the main menu and work together to help users test their own car designs.

From start to finish, the program begins at the menu, where the user can choose different functions from the top bar. Users can run the platform without setting anything up at the beginning.

First, the user should choose a map from the list or upload a map from a folder. Then they can check it in the 3D preview and add a path for the car.

Second, the user can upload a car model or choose one of the 3D models that are already in the list. Then the 3D model will appear in the 3D preview, and the user can also adjust or scale the model.

After the 3D model is uploaded, the user can click Properties and choose either dynamic or static. For static settings, the user inputs a fixed value and observes the output under that condition. For dynamic settings, the input values change randomly to show how the system behaves under different situations.

Then the user can start running the program. The user can click the CFD button to simulate air resistance or change the viewing angle.

Finally, the user can open the Settings menu and quit the program. The program was developed using the Unity engine to create the 3D interface, simulation environment, and interactive controls [9].

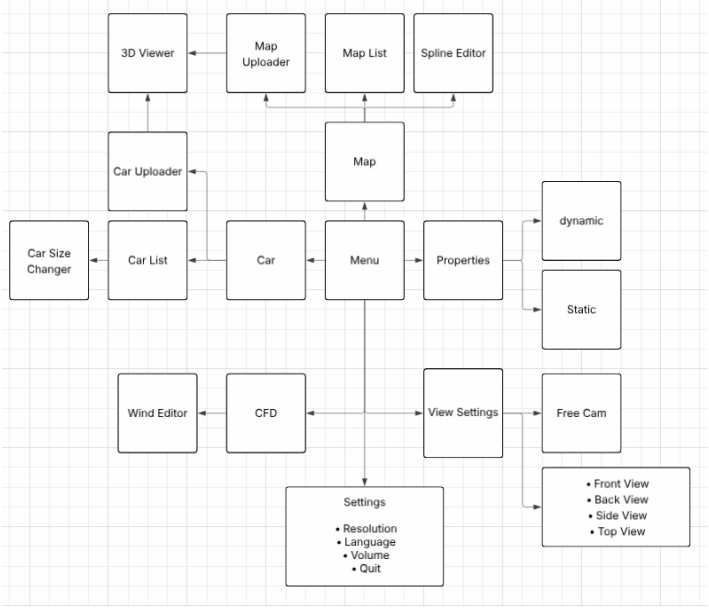

Figure 1. Overview of the solution

The Properties component acts as the control center for simulation inputs and outputs. Its purpose is to let the user change vehicle/environment parameters, combine those values with live runtime data when dynamic mode is enabled, and then display updated physics results in the UI. It uses Unity services such as MonoBehaviour lifecycle methods, TextMeshPro input/output fields, dropdown filters, and button events. It also uses a custom optional-value concept so dynamic values can override static values only when available. Broadly, this component connects user interaction, live simulation state, and the calculation engine into one continuous loop.

Figure 2. Concise description of system architecture or component shown

```
void CalculateAndRefresh()
{
    if (dynamicModeEnabled)
    {
        ApplyDynamicSpeedControl(false);
        ApplyDynamicWindControl(false);
    }
    else if (staticInputsDirty)
        ApplyStaticSpeedRangeControl(false);

    if (!dynamicModeEnabled && !staticInputsDirty)
        return;

    Properties_IOs_DynamicInput activeDynamicInput = dynamicModeEnabled ? dynamicInput : null;
    Properties_IOs_Resolver.ResolveInto(inputs, activeDynamicInput, resolvedInputs);
    CarSimCalculator.Compute(resolvedInputs, outputs);


    OnCalculateAndRefresh?.Invoke(resolvedInputs, outputs);
    if (!dynamicModeEnabled)
        staticInputsDirty = false;
}
```

Figure 3. Core refresh cycle logic of the Properties component

The code shown is the main refresh cycle for the Properties system. It runs once at startup and then repeatedly during gameplay, and it can also run after user interaction. First, it checks whether the system is in dynamic mode or static mode. In dynamic mode, it updates speed and wind controls from live values; in static mode, it applies user-defined speed range values only when those inputs are changed. It then decides whether recalculation is needed and exists early if nothing changes, which saves unnecessary processing. Next, it creates the effective input set by merging static values with dynamic overrides. After that, it sends the resolved inputs into the simulation calculator, which computes outputs like air density, drag, acceleration range, pressure values, and motor temperature. The method then loops through output definitions and updates UI fields. Single-value outputs are formatted directly, while range outputs are formatted as minimum-to-maximum values. It catches previous output values, so the UI only refreshes when displayed values actually change. Finally, it triggers an event so other systems can react and clears the static “dirty” flag after a successful static recompute.

The CFD component is responsible for turning vehicle motion and wind conditions into a live aerodynamic visualization on the car model [10]. Its purpose is to estimate where pressure builds or drops across surfaces and display that information as a heatmap so users can immediately see airflow effects during simulation. It uses Unity runtime systems such as mesh renderers, materials, shaders, coroutines, and per-frame updates, along with cached bake data to avoid expensive recalculation every frame. This component does not rely on neural networks or NLP [11]. Instead, it uses a physics-inspired heuristic approach: compute relative airflow, derive dynamic pressure, then blend precomputed directional surface states to produce a stable, real-time visual result.

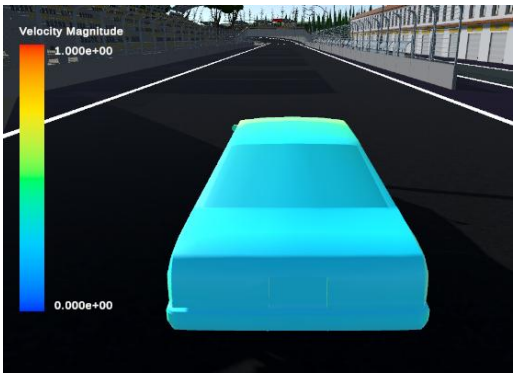

Figure 4. Accurate description of CFD component or visualization result

```
void UpdateLiveState()
{
    float vehicleSpeed = ResolveVehicleSpeed();
    currentAmbientWindSpeed = ResolveAmbientWindSpeed();
    currentAirDensity = ResolveAirDensity();

    Vector3 vehicleVelocityWorld = ResolveVehicleVelocityWorld(vehicleSpeed);
    Vector3 ambientWindVelocityWorld = ResolveWindDirectionWorld() * currentAmbientWindSpeed;
    currentRelativeAirVelocityWorld = ambientWindVelocityWorld - vehicleVelocityWorld;

    float minRelativeAirSpeed = Mathf.Max(0.05f, runtimeDisplay.minRelativeAirSpeed);
    if (currentRelativeAirVelocityWorld.sqrMagnitude < minRelativeAirSpeed * minRelativeAirSpeed)
    {
        Vector3 fallbackDirection = ResolveWindDirectionWorld();

        if (fallbackDirection.sqrMagnitude < 0.0001f)
            fallbackDirection = -transform.forward;

        currentRelativeAirVelocityWorld = fallbackDirection.normalized * minRelativeAirSpeed;
    }

    currentRelativeAirSpeed = currentRelativeAirVelocityWorld.magnitude;
    currentDynamicPressurePa = 0.5f * currentAirDensity * currentRelativeAirSpeed * currentRelativeAirSpeed;

    Transform localFlowTransform = vehicleTransform != null
        ? vehicleTransform
        : activeModelObject != null
            ? activeModelObject.transform
            : transform;

    currentLocalFlowDirection = localFlowTransform.InverseTransformDirection(currentRelativeAirVelocityWorld.norm
    currentFlowYawDeg = Mathf.Atan2(currentLocalFlowDirection.x, -currentLocalFlowDirection.z) * Mathf.Rad2Deg;
    currentFlowPitchDeg = Mathf.Asin(Mathf.Clamp(currentLocalFlowDirection.y, -1f, 1f)) * Mathf.Rad2Deg;
}
```

Figure 5. Real-time airflow computation logic for CFD visualization

The smaller code section runs during visualization updates and computes the live airflow state that drives the CFD color output [13]. First, it reads three core inputs: vehicle speed, ambient wind speed, and air density. Next, it converts those values into world-space velocities for the car and the wind, then subtracts them to get relative air velocity. That subtraction is important because aerodynamic effects depend on how air moves relative to the vehicle, not just how fast the car is moving alone. The method then enforces a minimum airflow magnitude so the visualization remains numerically stable and directional even when both car and wind are near zero. After that, it computes relative air speed and dynamic pressure using the standard relationship where pressure scales with one half density times speed squared. It then converts the world airflow direction into local space so the system can determine yaw and pitch of flow relative to the model orientation. Those local angles are stored as runtime state and later used to choose and blend the nearest pre-baked CFD directional states. In simple terms, this code is the live physics bridge between motion data and the heatmap shading logic.

The spline editor component is an in-game map authoring tool that lets users create, adjust, and remove spline points directly in the scene while the simulation is running. Its purpose is to provide an interactive route-editing workflow instead of relying only on editor-time tools. It uses runtime raycasting to place points on surfaces, a spline library to store and evaluate curve geometry, and transform gizmo handles so points can be moved and reoriented visually. It also controls editing modes, line rendering, and point-handle feedback so the user always sees what state they are in. This component does not use machine learning or external backend services; it is a deterministic interaction system driven by user input, scene collisions, and spline math [12].

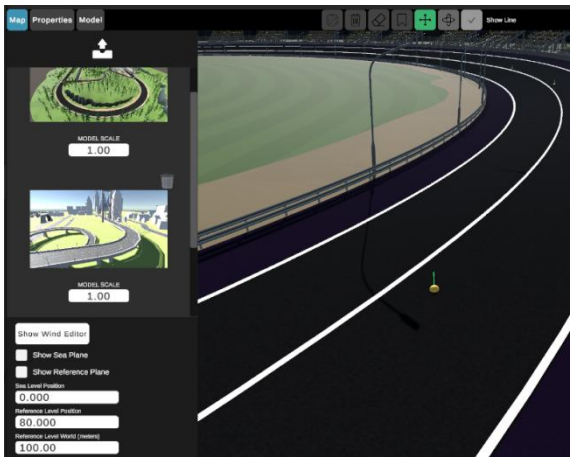

Figure 6. Clear description of spline editor or interaction system

```
void HandlePlacementAndDeletionInput()
{
    if (isDeleting)
    {
        hasPreviewHit = false;
        SetPreviewActive(false);
        UpdateDeletionInput();
        return;
    }

    if (isPlacing)
    {
        UpdatePlacementPreview();
        if (hasPreviewHit && Input.GetMouseButtonDown(0) && !IsPointerOverRuntimeUI())
            AddPointAt(previewPosition, previewNormal);
        return;
    }

    hasPreviewHit = false;
    SetPreviewActive(false);
    UpdateSelectionInput();
}
```

Figure 7. Input routing logic for spline editor interaction modes

This method is the central input router for the spline editor and runs during the normal frame update while the tool is active. Its job is to decide which interaction mode should respond to mouse input at that moment: deleting points, placing points, or selecting existing handles. The first branch checks whether delete mode is active. If it is, the method disables placement preview visuals, marks preview hit as false, and forwards control to deletion logic, so a click removes the point under the cursor. If delete mode is not active, it then checks placement mode. In placement mode, it updates the preview object by raycasting into the scene so the user sees where the next point would land and what normal orientation it would get. If the preview is valid and the player clicks while not over UI, it creates a spline point at the preview position with the preview normal. If neither deleting nor placing is active, the method falls back to selection behavior: it hides preview state and calls selection logic so clicking can choose an existing point handle for transform-gizmo editing. In short, this method prevents mixed behaviors and guarantees one clean interaction path per frame, which keeps the spline editor predictable and user-friendly.

4. EXPERIMENT

A potential blind spot is the accuracy of the simplified CFD system compared to ANSYS Fluent. This is important because inaccurate results may mislead users when evaluating aerodynamic performance.

The experiment compares drag force outputs between the Unity-based CFD system and reference data obtained from ANSYS Fluent simulations. Three car shapes are tested: streamlined, semi-streamlined, and box shaped. Air density is fixed at 1.225 kg/m³ and wind speed is set to 0 m/s. Vehicle speed is tested at 10 m/s, 20 m/s, 30 m/s, 40 m/s, and 50 m/s. For each condition, drag force values are recorded from both systems. The ANSYS Fluent data serves as the control baseline. The percent error between the Unity CFD and ANSYS results is calculated to evaluate accuracy across different speeds and geometries.

Speed (m/s)	ANSYS Fluent Drag (N)	Unity CFD Drag (N)	% Error
10	120	130	8.3%
20	480	510	6.3%
30	1080	1120	3.7%
40	1920	2000	4.2%
50	3000	3150	5.0%

Figure 8. Comparison of drag force between Unity-based CFD system and ANSYS Fluent across tested speeds

Percent Error Values: 8.3%, 6.3%, 3.7%, 4.2%, 5.0%
Mean: (8.3 + 6.3 + 3.7 + 4.2 + 5.0) / 5 = 27.5 / 5 = 5.5%
Median: Ordered → 3.7%, 4.2%, 5.0%, 6.3%, 8.3% → 5.0%
Lowest: 3.7%
Highest: 8.3%
The data shows that the simplified CFD system maintains consistent accuracy across tested speeds, with errors remaining within a relatively small range. The highest error occurs at lower speeds, likely due to weaker airflow reducing the effectiveness of the surface-based approximation. At higher speeds, the results align more closely with ANSYS Fluent because the dynamic pressure term dominates and stabilizes the output. One unexpected observation is that the lowest error appears at mid-range speeds rather than at the highest speed. The most significant factor affecting accuracy is the simplified surface heuristic model, which does not fully capture complex flow behavior such as turbulence and separation that are resolved in ANSYS Fluent simulations.

5. RELATED WORK

In the scholarly source from Arpit Gary, the researchers used CFD to simulate airflow around the car and the air resistance before making the physical car, which allows him to make a change easily [3]. This method is good at allowing people to test the aerodynamics of their model before they really build one. It saves time and energy. However, it is hard for the individual and student's term, since this method requires professional equipment, knowledge and skills. It also ignores the usability. Our project improves this by making a platform based on Unity, which allow users easily to check and use it without long-term training.

Based on the scholarly source, aerodynamic drag can also be estimated using hand calculations [4]. NASA provides a drag equation that allows people to estimate air resistance based on factors such as air density, velocity, drag coefficient, and frontal area. This method does not require advanced technological tools and can be used to obtain a basic estimation of aerodynamic forces. However, the equation relies on simplified assumptions and represents an idealized situation. As a result, it does not fully capture the complexity of real airflow around vehicles. Our project improves on this approach by providing a simulation environment that can visualize airflow

behavior more realistically. For example, users can observe how different parts of the vehicle experience different levels of drag through color visualization in the simulation.

One common method used to analyze vehicle aerodynamics is wind tunnel testing. According to Hucho, wind tunnels are important experimental tools used by vehicle aerodynamicists to study airflow around cars [5]. In this method, a scale model of the vehicle is placed in a controlled airflow environment, and sensors are used to record aerodynamic data. This method is effective because it provides reliable experimental results and allows engineers to observe airflow patterns around the vehicle. However, wind tunnel testing is expensive and requires building a physical model first, which makes design changes more difficult. In comparison, our project is more cost-effective and allows users to easily modify 3D models in the simulation before building a final design.

6. CONCLUSIONS

One limitation of our project is that the CFD simulation is simplified [14]. The airflow calculation mainly shows general air resistance and wind direction, but it doesn't include all real-world aerodynamic factors such as turbulence, surface roughness, or changing weather conditions. The result may not fully represent the real environment, because of this.

Another limitation is that the car models and maps are limited. Users can upload their own models, but the built-in library of maps and vehicles is still small. This limits the variety of tests users can perform.

The user interface could also be improved [15]. Some functions require multiple steps and may not be intuitive for new users.

If we had more time, we would improve the CFD system by adding more accurate physics calculations and better visualization tools. We would also expand the map and vehicle libraries and redesign the interface to make the program easier to use. Also, if it is possible, I will add more functions to let the project do more stuff.

This project demonstrates how a simulation platform can help users analyze car aerodynamics through an interactive system. By combining 3D models, maps, and CFD simulations in Unity, the program provides a useful tool for testing vehicle designs and understanding how aerodynamic factors affect performance.

REFERENCES

[1] Sunny, Shohanur Rahaman. "AI-augmented aerodynamic optimization in subsonic wind tunnel testing for UAV prototypes." Saudi Journal of Engineering and Technology (SJEAT) 10.9 (2025): 402-410.

[2] Bartsch, Sarah M., et al. "The Potential Health Care Costs And Resource Use Associated With COVID-19 In The United States: A simulation estimate of the direct medical costs and health care resource use associated with COVID-19 infections in the United States." Health affairs 39.6 (2020): 927-935.

[3] Garg, Arpit. "Aerodynamic Investigation of a Three-Wheeled Vehicle Using a CFD Approach: A Study on Drag and Lift Variations." (2025).

[4] Cullen, Scott. "Trees and wind: A practical consideration of the drag equation velocity exponent for urban tree risk management." Arboriculture & Urban Forestry (AUF) 31.3 (2005): 101-113.

[5] Hucho, Wolf-Heinrich, ed. Aerodynamics of road vehicles: from fluid mechanics to vehicle engineering. Elsevier, 2013.

[6] Bottasso, Carlo L., Filippo Campagnolo, and Vlaho Petrović. "Wind tunnel testing of scaled wind turbine models: Beyond aerodynamics." Journal of wind engineering and industrial aerodynamics 127 (2014): 11-28.
[7] Gahan, Andrew. 3D automotive modeling: an insider's guide to 3D car modeling and design for games and film. Routledge, 2012.
[8] Zawawi, Mohd Hafiz, et al. "A review: Fundamentals of computational fluid dynamics (CFD)." AIP conference proceedings. Vol. 2030. No. 1. AIP Publishing LLC, 2018.
[9] Bartneck, Christoph, et al. "The robot engine—Making the unity 3D game engine work for HRI." 2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN). IEEE, 2015.
[10] Hadjadj, Abdellah, and A. Kudryavtsev. "Computation and flow visualization in high-speed aerodynamics." Journal of turbulence 6 (2005): N16.
[11] Alshemali, Basemah, and Jugal Kalita. "Improving the reliability of deep neural networks in NLP: A review." Knowledge-Based Systems 191 (2020): 105210.
[12] Jordan, Michael I., and Tom M. Mitchell. "Machine learning: Trends, perspectives, and prospects." Science 349.6245 (2015): 255-260.
[13] Sorokes, James M., James Hardin, and Brad Hutchinson. "A CFD Primer: What Do All Those Colors Really Mean?." (2016).
[14] Blocken, Bert, Ted Stathopoulos, and Jan Carmeliet. "CFD simulation of the atmospheric boundary layer: wall function problems." Atmospheric environment 41.2 (2007): 238-252.
[15] Sridevi, S. "User interface design." International Journal of Computer Science and Information Technology Research 2.2 (2014): 415-426.