Assessment of Turbulent Boundary Layer Detachment due to Wall-Curvature-Driven Pressure Gradient

by
David Paeres Castaño

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
Mechanical Engineering

UNIVERSITY OF PUERTO RICO
MAYAGÜEZ CAMPUS

2022

Approved by:

Guillermo Araya, Ph.D. President, Graduate Committee	Date
José E. Lugo, Ph.D. Member, Graduate Committee	Date
Wilson Rivera, Ph.D. Member, Graduate Committee	Date
Iván Baiges, Ph.D. Representative of the Office of Graduate Studies	Date
Ruben Díaz, Ph.D. Chairperson of the Department	Date

Assessment of Turbulent Boundary Layer Detachment due to Wall-Curvature-Driven Pressure Gradient

David Paeres Castaño
MASTER OF SCIENCE in Mechanical Engineering
University of Puerto Rico at Mayagüez
Dr. Guillermo Araya, Faculty Advisor, Mechanical Engineering

(ABSTRACT)

The present study provides fundamental knowledge on an issue in fluid dynamics that is not well understood: flow separation and its association with heat and contaminant transport. In the separated region, a swirling motion increases the fluid drag force on the object. Very often, this is undesirable because it can seriously reduce the performance of engineered devices such as aircraft and turbines. Furthermore, Computational Fluid Dynamics (CFD) has gained ground due to its relatively low cost, high accuracy, and versatility. The principal aim of this study is to numerically elucidate the details behind momentum and passive scalar transport phenomena during turbulent boundary layer separation resulting from a wall-curvature-driven pressure gradient. With OpenFOAM CFD software, the numerical discretization of Reynolds-Averaged Navier-Stokes and passive scalar transport equations will be described in two-dimensional domains via the assessment of two popular turbulence models (i.e., the Spalart-Allmaras and the $K-\omega$ SST model). The computational domain reproduces a wind tunnel geometry from previously performed experiments by Baskaran *et al.* (JFM, vol. 182 and 232 "A turbulent flow over a curved hill." Part 1 and Part 2). Only the velocity and pressure distribution were measured there, which will be used for validation purposes in the present study. A second aim in the present work is the scientific visualization of turbulent events and coherent structures via the ParaView toolkit and Unity game engine. Thus, fully immersive visualization approaches will be used via virtual reality (VR) and augmented reality (AR) technologies. A Virtual Wind Tunnel (VWT), developed for the VR approach, emulates the presence in a wind tunnel laboratory and has already employed fluid flow visualization from an existing numerical database with high temporal/spatial resolution, i.e., Direct Numeric Simulation (DNS). In terms of AR, a FlowVisXR app for smartphones and HoloLens has been developed for portability. It allows the user to see virtual 3D objects (i.e., turbulent coherent structures) invoked into the physical world using the device as the lens.

Evaluación del Desprendimiento de Capa Límite Turbulenta causado por Gradiente de Presión debido a Curvatura Superficial

David Paeres Castaño
Maestro en Ciencias en Ingeniería Mecánica
Universidad de Puerto Rico en Mayagüez
Dr. Guillermo Araya, Profesor Consejero, Ingeniería Mecánica

(RESUMEN)

El estudio presente proporciona un conocimiento fundamental sobre un tema en la dinámica de fluidos que no se entiende bien: separación del flujo y su asociación con el transporte de calor y contaminantes. En la región separada, un movimiento de remolino aumenta la fuerza de arrastre de fluido sobre el objeto. Muy a menudo, este no es deseado debido a que puede reducir seriamente el rendimiento de dispositivos ingenieriles, como aviones y turbinas. Además, la dinámica de fluido computacional (CFD, por siglas en inglés) ha ganado terreno debido a su costo relativamente bajo, alta precisión y versatilidad. El objetivo principal de este estudio es dilucidar numéricamente los detalles detrás del momento y los fenómenos de transporte escalar pasivo durante la separación de la capa límite turbulenta resultante de un gradiente de presión basado por la curvatura de pared. Con el software de CFD OpenFOAM, la discretización numérica de las ecuaciones Reynolds-Averaged Navier-Stokes y transporte de escalar pasivo va a ser descrita en dominios bidimensionales a través de la evaluación de dos modelos de turbulencia populares (i.e., los modelos Spalart-Allmaras y $K-\omega$ SST). El dominio computacional reproduce una geometría del túnel de viento de experimentos previamente realizados por Baskaran *et al.* (JFM, vol. 182 and 232 "A turbulent flow over a curved hill." Part 1 y Part 2). Allí solo se midieron la velocidad y la distribución de presión, que se utilizará para fines de validación en el presente estudio. Un segundo objetivo en el trabajo presente es la visualización científica de los eventos turbulentos y las estructuras coherentes a través de ParaView y Unity. Por lo tanto, los enfoques de visualización inmersiva se utilizarán a través de tecnologías de realidad virtual (VR, por siglas en inglés) y realidad aumentada (AR, por siglas en inglés). Un túnel de viento virtual (VWT, por siglas en inglés), desarrollado para el enfoque VR, emula la presencia en un laboratorio de túnel de viento y ya ha empleado la visualización de fluidos desde una base de datos numérica con alta resolución temporal/espacial existente (i.e., DNS). En términos de AR, se ha desarrollado una aplicación FlowVisXR para teléfonos inteligentes y HoloLens por portabilidad. Permite al usuario ver objetos 3D virtuales (i.e., estructuras turbulentas coherentes) invocadas en el mundo físico utilizando el dispositivo como lente.

Dedication

Este arduo trabajo merece ser dedicado a varias personas:
Como reza el dicho, "detrás de cada hombre exitoso hay una mujer", en este caso, la primera, es mi madre, que aún con la distancia me ha apoyado en mi carrera académica y me ha dejado saber lo orgullosa que esta de su único hijo. La segunda mujer, es mi pareja, que aún con los eventos históricos que han ocurrido paralelamente durante este trabajo, siempre ha estado al lado mío ofreciéndome su apoyo.
A mi consejero académico, el Dr. Guillermo Araya, que su profesionalidad, paciencia y disposición es de persona únicas en la Academia y estoy seguro que cualquier persona desearía estar cobijado bajo su mentoría y amabilidad.
A mi colega, maestro, modelo a seguir y hermano del alma, Christian Lagares, que por alguna razón siempre tiene conocimientos del tópico en el que estoy comenzando a trabajar y nunca me ha faltado su ayuda.
A mi familia que siempre me ha apoyado en mi travesía por la ciencias y reconocen mi esfuerzo.
Y por último, pero no menos importantes, a mis dos educadoras de la escuela preparatoria que desde un principio descubrieron mi potencial y además de motivarme de una manera u otra, me dirigieron hacia la ruta de la ingeniería. Gracias Claribel Pérez y Suannete Otero.

-David

Acknowledgments

First, I would like to express my sincere gratitude to my advisor, Dr. Guillermo Araya, for his outstanding and constant support during my research and for really introducing me to computational fluid dynamics while being a vital part of the past opportunity with patience and motivation towards me. I will always be deeply grateful.
I want to thank the Department of Mechanical Engineering of the University of Puerto Rico at Mayagüez (UPRM) for providing me with the resources to accomplish my research.
Also, I want to thank Dr. José Lugo and Dr. Wilson Rivera for accepting to operate as my M.S. advisory committee. I would like to thank all the staff and my colleagues from the Mechanical Engineering Department for their support.
Finally, I want to acknowledge NSF-CAREER award #1847241 and AFOSR grants #FA9550-22-1-0089 and #FA9550-17-1-0051 for financially supporting this research effort. This work was also supported in part by the Center for the Advancement of Wearable Technologies and the National Science Foundation under grant OIA-1849243. Furthermore, major achievements during my MSc degree's experience are summarized as follows:

- Selected as sponsored MSc student in the 2022 Computational Physics Summer Workshop organized by the Los Alamos National Laboratory Advanced Scientific Computing (ASC) Program during June 13 – August 19, 2022.

- One Journal paper: Energies. 2022; 15(16):6013
  https://doi.org/10.3390/en15166013

- Two Conference papers: AIAA SCITECH 2022 Forum (DOI: 10.2514/6.2022-0049), AIAA Scitech 2021 Forum (DOI: 10.2514/6.2021-1600)

- Three Video displays: 74th Annual Meeting of the APS Division of Fluid Dynamics
  (DOI: 10.1103/APS.DFD.2021.GFM.V0027) and
  (DOI: 10.1103/APS.DFD.2021.GFM.V0028),
  73rd Annual Meeting of the APS Division of Fluid Dynamics
  (DOI: 10.1103/APS.DFD.2020.GFM.V0045)

-David

This page has been intentionally left blank.

Table of Contents

List of Figures

List of Symbols and Abbreviations

Acronyms & Variables

2D	Two Dimensional
3D	Three Dimensional
3D+	Three Dimensional and Over
APG	Adverse Pressure Gradient
AR	Augmented Reality
AV	Augmented Virtuality
CFD	Computational Fluid Dynamics
C_f	Skin Friction Coefficient
C_p	Pressure Coefficient
c_p	Isobaric Specific Heat
C_s	Smagorinsky Constant
D	Flow Detachment
DNS	Direct Numerical Simulation
E_{ij}	Rate of deformation
F	Force $[N]$
F_b	Body Forces
F_s	Surface Forces
FOV	Field Of View
FPG	Favorable Pressure Gradient
fps	Frames Per Second
g	Gravitational Constant

GUI	Graphical User Interface
HMD	Head-Mounted Display
ID	Flow Incipient Detachment
ITD	Flow Intermittent Transitory Detachment
K	Turbulent kinetic energy
K	Kelvin Temperature Unit
k	Thermal Conductivity Coefficient
L	Characteristic Length
LES	Large-Eddy Simulation
m	Mass or Meters (we disambiguate as required throughout the text)
MR	Mixed Reality
ODE	Ordinary Differential Equation
P	Pressure $[Pa]$
PMR	Pure Mixed Reality
Pr	Prandtl Number
P_w	Static Wall pressure $[Pa]$
q_∞	Freestream Dynamic Pressure $[Pa]$
RANS	Reynolds-Averaged Navier-Stokes
Re	Reynolds Number
Re_θ	Momentum Thickness Reynolds Number
S	Suitable Source Term
s	Seconds Time Unit
SA	Spalart Allmaras turbulent model
Sc	Schmidt Number
SDTBL	Spatially-Developing Turbulent Boundary Layer

SST	$K-\omega$ Shear Stress Transport turbulent model
S_T	Suitable Source Term for Passive Thermal Scalar
St	Stanton Number
T	Temperature [K]
t	Time [s]
TPC	Two-Point Correlation
U	Velocity Vector Field
$U^* = \frac{U}{U_\infty}$	Dimensionless Velocity
$u_\tau = \sqrt{\frac{\tau_w}{\rho}}$	Skin Friction Velocity $[m/s]$
u	Streamwise Velocity Component
v	Wall-normal Velocity Component
VR	Virtual Reality
VWT	Virtual Wind Tunnel
X	Spatial Position Vector
x	x-axis Cartesian Dimension
XR	Extended Reality
y	y-axis Cartesian Dimension
$y^+ = \frac{y\tau_w}{\nu}$	Dimensionless Wall-normal Distance
w	Spanwise Velocity Component
z	z-axis Cartesian Dimension
ZPG	Zero Pressure Gradient

Greek Letters

α	Thermal Diffusivity Coefficient $[m^2/s]$
α_{eff}	Effective Thermal $[m^2/s]$ Diffusivity Coefficient

Δ	Difference
δ	Boundary-layer Thickness $[m]$
δ^*	Boundary-layer Displacement Thickness $[m]$
δ_{ij}	Kronecker Delta
ϵ	Turbulent Energy Dissipation Ratio $[m^2/s^3]$
η	Spatial Blasius Similarity Variable
θ	Dimensionless temperature or Momentum Thickness (we disambiguate as required throughout the text)
λ	Bulk Viscosity Coefficient $[Pa-s]$
μ	Molecular Dynamic viscosity $[Pa-s]$
μ_t	Turbulent Eddy Viscosity $[Pa-s]$
ν	Molecular Kinematic Viscosity $[m^2/s]$
ν_t	Turbulent Kinematic Viscosity $[m^2/s]$
ρ	Fluid Density $[kg/m^3]$
τ	Shear Stress $[Pa]$
τ_w	Wall Shear $[Pa]$
ω	Turbulent Energy Specific Dissipation Rate $[1/s]$
$\alpha_\omega, \beta, \beta^*, \sigma, \sigma^*, \sigma_k, \sigma_\epsilon, C_{1\epsilon}, C_{2\epsilon}$	Turbulent Models Constant Coefficients

Subscripts, Superscripts & Operators

$()_x$	x-axis Cartesian Direction
$()_y$	y-axis Cartesian Direction
$()_z$	z-axis Cartesian Direction
$()_s, ()_w$	Wall Surface
$()_i, ()_j, ()_k$	Arbitrary Directional Dimensions
$()^*$	Non-dimensionalized Value

$()_\infty$	Freestream Limiting value
$\bar{()}$	Time-averaged values
$()'$	Fluctuation about temporal mean
$()^+$	Inner scaled units
∇	Gradient Operator
∇^2	Laplacian Operator
$\nabla\cdot$	Divergence Operator

Chapter 1
Introduction

The present introductory chapter is intentionally elaborated to immerse readers into fundamental aspects of fluid dynamics, turbulence, and its modeling, as well as in scientific visualization of big data. Considering a possible scenario, the reader is not entirely educated with fluid mechanics or data visualization using any Extended Reality (XR) approach. This chapter will give a short history description and a comprehensive theoretical preparation in the first four sections. The topics' scopes are fundamental knowledge of the flow separation phenomena, passive scalar transport (e.g., heat and pollutants), and visualization of virtual objects through Virtual and Augmented Reality. The identified concepts as substantial are classified into four sections and discussed up to the necessary level: (i) Historical Context §1.1, (ii) Fluid Dynamics Theory §1.2, (iii) Computational Fluid Dynamics (CFD) §1.3, (iv) Data Visualization in XR §1.4. After an introductory journey for readers, the last section is devoted to the description of the project, objectives, and discussion of the intellectual merit of the present thesis §1.5.

1.1 Historical Context

1.1.1 Resistance Notion

After thinking about the creation of arrows, boats, drainage systems, etc., it is recognized that fluid mechanics application has existed since prehistory. Still, the roots of this discipline (which by then was not named fluid mechanics) can be traced far back to the Classical Period of Ancient Greek. Aristotle (350 B.C.) suggested: "that a body moving through a continuum encounters a resistance" (Anderson Jr and Anderson 1998).

Although essential contributions to fluid mechanics occurred in the periods of Archimedes in 250 B. C. (Heath et al. 2002), the Alexandria School (Wahba 2016)(Landels 1979), Sextus Julius Frontinus of Ancient Rome (Deming 2020), Islamicate physicist and engineers of Middle Ages (Hill 2019); it was not until Leonardo Da Vinci (1490), Galileo Galilei (1600), Edme Mariotte (1673), and Isaac Newton (1687) emerged some statements regarding the direct proportionality of flow resistance (Anderson Jr and Anderson 1998).

After the universal law of gravitation, it is possible that the second most crucial contribution of Isaac was his three Newton's Laws of Motion (which implicitly included also calculus). For example, the Third Law predicts the presence of resistance in practically any scenario. In fluid dynamics, this resistance is called drag, which is a type of friction force. Also, the Second Law was one of the three Fundamental Physics Principles used to formulate the Navier-Stokes equations or momentum conservation equations. These were the formulation needed to describe a real fluid flow (Anderson Jr and Anderson 1998).

1.1.2 Beginning of Fluid Mechanics

By the end of the 19th century, Fluid Mechanics was strongly fractured into two sciences. The mathematically more elegant science was known as Theoretical Hydrodynamics and was built with Euler's equations of motion. The disadvantage of this science was its little applicability because the results seemed to contradict everyday experiences, for example, failing to estimate pressure losses and the drag on a body moving through a fluid. For this reason, there was also the empirical science called Hydraulics, which, based on experimental data, disagreed with Theoretical Hydrodynamics, yet the methods were those used by engineers for applications of Fluid Mechanics (Schlichting and Gersten 2017).

At the beginning of the 20th century, the german Ludwig Prandtl started the route in the unification of these two sciences. In the mid of 20th century, a high correlation between theory and experimentation was achieved. By then, it was already known that the discrepancies were explainable after ignoring the viscous effects of a flow. The advances in Fluid Mechanics were unimaginable to what there had been in the previous century. Prandtl using simple experiments together with theoretical considerations demonstrated that a flow past a body could be divided into two regions: a thin layer attached to the body of the object called the Boundary Layer, where the viscous effects could not be neglected, and the other remaining region, outside the Boundary Layer, which the viscosity effects could be neglected. The ingenious idea of Prandtl was not only a convincing physical explanation but also highly reduced the mathematical difficulty in the usefulness of viscous flows. Since the beginning of the last century, Prandtl's Boundary Layer Theory has been exceptional for developing and researching Fluid Mechanics (Schlichting and Gersten 2017).

1.2 Fluid Dynamics Theory

1.2.1 Navier-Stokes Equations

Newton's Second Law states that the sum of external forces of a system is equal to the change in time of momentum (where momentum is the product of the mass of a particle and its velocity). Although, it is commonly traduced as net force is equal to the mass and acceleration product because of the assumption of constant mass, shown in Equation (1.1).

$$F = ma \tag{1.1}$$

Suppose Equation (1.1) is rewritten to the original version but with also a constant mass. In that case, it leads to Equation (1.2), where the sum of external forces is divided into two portions: (i) F_s, the forces acting only at the surfaces, and (ii) F_b, the forces acting on the entire mass of the body itself (i.e., volumetric forces). If only the gravitational force is considered as F_b, then Equation (1.3) reads,

$$m\frac{D}{Dt}(U) = F_s + F_b \tag{1.2}$$

$$F_b = mg \tag{1.3}$$

F_s can be represented by the stress tensor τ_{ij}, as shown in Equation (1.4). Where τ_{ij} is stress acting in the area normal to i with a force of direction of j.

$$\tau_{ij} = \begin{bmatrix} \tau_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \tau_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \tau_{zz} \end{bmatrix} \tag{1.4}$$

The Figure 1.1 shows the conventional direction in which the stresses act only by looking at the three principal faces (front faces) of an infinitesimal element. Given that stress is force per unit area; after multiplying by the respective area, the force is obtained. With the intention to decompose the forces into the three Cartesian dimensions, the Equation (1.5) is obtained. Where each stress tensor component τ_{ij} is multiplied by the area normal to i resulting in forces acting in the direction of j.

$$\begin{Bmatrix} dF_x &=& \tau_{xx}(dydz) &+& \tau_{yx}(dxdz) &+& \tau_{zx}(dxdy) \\ dF_y &=& \tau_{xy}(dydz) &+& \tau_{yy}(dxdz) &+& \tau_{zy}(dxdy) \\ dF_z &=& \tau_{xz}(dydz) &+& \tau_{yz}(dxdz) &+& \tau_{zz}(dxdy) \end{Bmatrix} \tag{1.5}$$

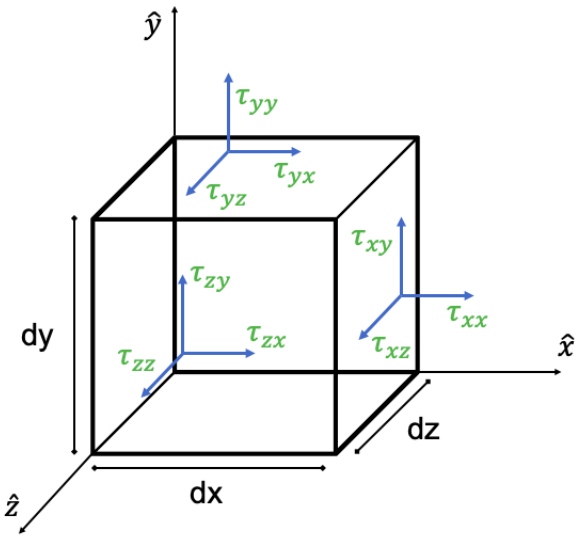


Figure 1.1: Diagram of the stress tensor τ_{ij} acting on an infinitesimal element (adapted from White (2011)).

Using Taylor's expansion up to first order, thus truncating to the first derivative, emerges an appropriate mathematical formulation able to calculate the net forces considering the stresses at the other three no-principal faces (back faces) (White 2011). Using as an example the forces acting in the direction of the x-axis, the Equation (1.6) is developed.

$$\left\{\begin{array}{lcll} \tau_{xx,net} & = & \tau_{xx,front} - \tau_{xx,back} & = \frac{\partial \tau_{xx}}{\partial x}(dx) \\ \tau_{yx,net} & = & \tau_{yx,front} - \tau_{yx,back} & = \frac{\partial \tau_{yx}}{\partial y}(dy) \\ \tau_{zx,net} & = & \tau_{zx,front} - \tau_{zx,back} & = \frac{\partial \tau_{zx}}{\partial z}(dz) \end{array}\right\} \tag{1.6}$$

Substituting Equation (1.6) in the x-component of Equation (1.5) results in Equation (1.7).

$$\left\{dF_x = \frac{\partial \tau_{xx}}{\partial x}(dx)(dydz) + \frac{\partial \tau_{yx}}{\partial y}(dy)(dxdz) + \frac{\partial \tau_{zx}}{\partial z}(dz)(dxdy)\right\} \tag{1.7}$$

Going back to Equation (1.2) and replacing Equations (1.3) and (1.7) of all three directions, brings Equation (1.8). Recognizing that the volume of the infinitesimal element is the product of the three Cartesian distances dx, dy and dz, the Equation (1.8) written

by unit volume and in tensor notation results in Equation (1.9).

$$m\frac{D}{Dt}(U) = mg + dF_{s,net} \tag{1.8}$$

$$\rho\frac{D}{Dt}(U_i) = \rho g_i + \nabla \cdot \tau_{ij} \tag{1.9}$$

Departing from Stokes's assumptions (White 2011):

- The fluid is continuous and the Stress Tensor τ_{ij} is a linear function of the Strain Rate Tensor ϵ_{ij} (following Hooke's Law).
- The fluid is isotropic, meaning the deformation is independent of any coordinates system axis selected.
- When the fluid is at rest, the deformation law reduces to only the hydrostatic pressure condition $\tau_{ij} = -P\delta_{ij}$. Where the δ_{ij} is the Kronecker delta.

The Law of Deformation for compressible flows results in Equation (1.10), being a crucial physical model granting the Navier-Stokes Equations, Eq. (1.11).

$$\tau_{ij} = -P\delta_{ij} + \mu\left(\frac{\partial U_i}{\partial X_j} + \frac{\partial U_j}{\partial X_i}\right) + \delta_{ij}\lambda\nabla \cdot U_i \tag{1.10}$$

$$\frac{D}{Dt}(\rho U_i) = \rho g_i + \nabla \cdot \left(- P\delta_{ij} + \mu\left(\frac{\partial U_i}{\partial X_j} + \frac{\partial U_j}{\partial X_i}\right) + \delta_{ij}\lambda\nabla \cdot U_i \right) \tag{1.11}$$

From Equation (1.11), the coefficient of bulk viscosity (λ) for small velocity divergence can be assumed as $\lambda = -2\mu/3$, where μ is the fluid dynamic viscosity (White 2011). If the flow is assumed incompressible, the velocity divergence ($\nabla \cdot U_i$) is negligible, and the fluid density (ρ) as constant (no buoyancy forces) can be taken out of the substantial derivative (i.e., $\frac{D}{Dt}$, also known as the material derivative).

1.2.2 Fluid Flow Types

To analyze a fluid flow is imperative to know its features. Identifying the characteristics and denominations the fluid flows comply with, could immensely simplify the analysis because of the selected modeling approach. However, uncountable fluid flows

classifications exist and can be divided into types, regimens, and much more. For example, fluid types can be separated by their substance properties or molecular interactions: ideal, ideal plastic, real, Newtonian/non-Newtonian, compressible/incompressible, and much more. Meanwhile, on the side of flow types, besides uniform/non-uniform, rotational/irrotational, steady/unsteady, one/two/three dimensional. Figure 1.2 shows other flow types in a general hierarchical order.

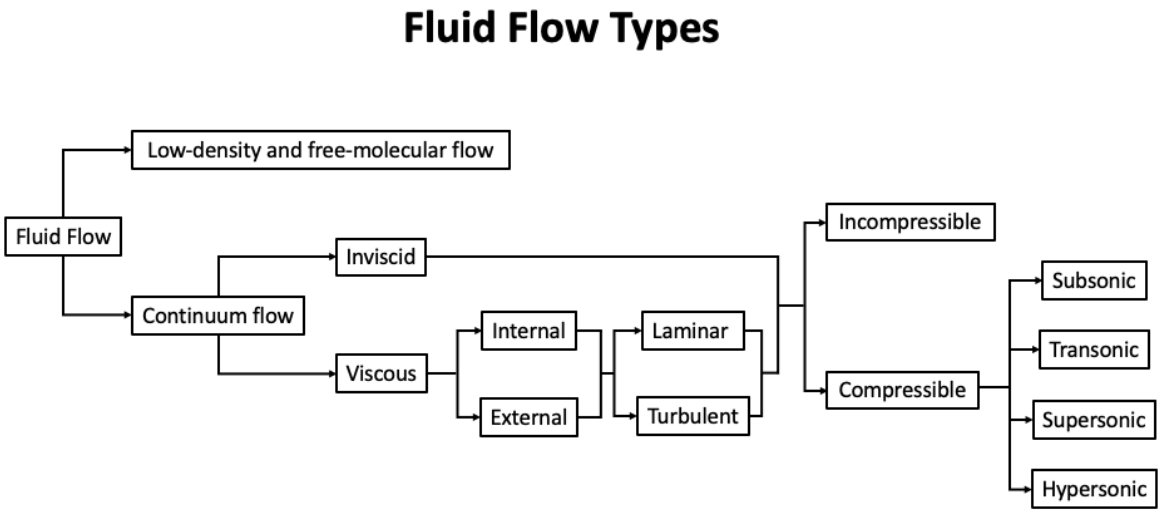


Figure 1.2: Schematic of general fluid flow types.

Because of the flow to be analyzed in this work, the relevant classification is a steady-Newtonian-2D-viscous-external-incompressible-turbulent flow. The following sources are recommended for detailed descriptions of the other concepts and more: White (2011) and Cengel and Cimbala (2014). A steady flow is a flow where all the properties and formula's terms are constant over time, meaning these are independent of it, and the time derivatives are set to zero. This is the case when one is interested in the response of the system (which is immersed into a fluid flow) at an infinite time. A Newtonian fluid is that one that follows Newton's observations of flow resistance made in those important years mentioned in the sub-section §1.1.1. He stated the Newtonian sine-squared Law for Aerodynamic force (theoretical verification that resistance is proportional to velocity squared) and resistance varies with the cross-sectional area of a body (Anderson Jr and Anderson 1998); translated into fluid mechanics: fluid's viscosity is linearly correlated to the strain rate, implicitly one of the Stokes' assumptions for the developing of the Navier-Stokes Equations Eq. (1.11).

A 2D flow means the velocity change in the third directional dimension is small enough to neglect the error caused by ignoring that dimensional direction ($dU_3 \approx 0$). If this condition is applied to Equation (1.11), this means the subscript i and j will only take the value of 1 and 2. Furthermore, if the coordinate system used is Cartesian, then the flow's domain takes a planar view. Incompressible flow is where the variations of density with pressure are negligible (Cengel and Cimbala 2014). A compressible fluid is defined as a substance (liquid or gas) that exhibits a volume variation (compression or expansion) due to changes in external static pressure. A compressible flow is related to the density variations as function of pressure variations (i.e., $\partial\rho/\partial P \neq 0$). Since the partial derivative $\partial\rho/\partial P$ is inversely proportional to the sound speed for a perfect gas or propagation speed of pressure perturbations, a commonly used compressibility indicator is called Mach number: the ratio of the local flow speed to the sound speed. For Mach numbers lower than 0.3, any flow generally can be assumed incompressible. The assumption of incompressible flow leads to simplifying formulations in the Equation (1.11): (i) The density variable can be pulled out from any derivative, (ii) The divergence term is zero since $\nabla \cdot U_i = 0$, and (iii) The thermal equation can be decoupled from the velocity equation when assuming the viscosity μ constant as well. Viscosity can be assumed constant if the density is also assumed not a function of the temperature (White 2011). Finally, resulting in a new and simplified momentum equation Eq. (1.12).

$$0 = \rho g_i - \nabla P + \mu \nabla^2 U_i \tag{1.12}$$

An external flow is an unbounded fluid over a surface such as a plate, hill, or pipe where a boundary layer grows indefinitely. On the contrary, an internal flow is confined and entirely bounded by solid surfaces (Cengel and Cimbala 2014). Identifying this characteristic is essential to calculate further a convenient and crucial parameter, the Reynolds number (Re). Re is a dimensionless parameter that primarily controls all viscous flows representing the ratio of inertial to viscous forces (White 2011), and its common formulation is Equation (1.13).

$$Re_L = \frac{\rho U L}{\mu} = \frac{U L}{\nu} \tag{1.13}$$

Where U is the characteristic flow velocity, ρ is the fluid's density, μ, and ν are the fluid's dynamic and kinematic viscosity, respectively. L is a characteristic length that sometimes acts with convenient liberty. A flow based on the Reynolds number can be divided into two regimes: laminar and turbulent. A laminar flow is smooth and orderly by clearcut layers, contrary to turbulent flow where chaos reigns and the layer-over-layer

appearance is impossible to see (Cengel and Cimbala 2014). Although a transition regime exists from laminar to turbulent, the Re has a critical value for each scenario where the importance of the turbulence becomes ever greater (Schlichting and Gersten 2017). In the transition regime, mathematical predictions of the flow are even more inaccurate. The transition from laminar to turbulent flow was first examined by Osborne Reynolds approximately in the year 1883 using a pipe flow (i.e., internal flow). By 1914, Ludwig Prandtl, using a flow over a sphere (i.e., external flow), showed that the boundary layer also can be laminar and turbulent and that this laminar-turbulent transition controls the flow's separation process and, thus, the drag's problem (Schlichting and Gersten 2017). Figure 1.3 shows on the left a crystal clear laminar flow while the turbulent flow on the right shows the chaotic unorganized flow.

Figure 1.3: Laminar flow on the left and turbulent flow on the right.

1.2.3 Boundary Layer Theory

Essential applications of the Boundary Layer theory are calculating the friction drag of bodies in a fluid flow. This theory is used to identify the optimum body shape to avoid or minimize flow separation, not only in flow past over a body but also in flows through a duct (i.e., internal flow); therefore, it is also used to describe the flow through blade cascades in compressors, turbines, diffusers, and nozzles, or for example, calculating the

maximum lift to drag ratio in an airfoil. The Boundary Layer theory is also crucial for heat transfer between a body and the fluid around it; since, as well as the boundary layer in the velocity field, a thermal boundary layer also forms (Schlichting and Gersten 2017) where transport phenomena of heat plays a crucial role due to the significant thermal gradients and forced convection.

Viscous effects can be observed in free shear flows, due to the presence of velocity gradients without the need for a typical surface with the no-slip condition (or a source of vorticity). Viscosity is a transport property that measures the fluid's ability to endure shear strain. Dynamic viscosity (μ) is a function of the density; therefore, the units are pressure multiplied by time (in the International System $[Pa \cdot s]$). Kinematic viscosity discards the density dependence, and the units are area over time (in the International System $[m^2/s]$). Viscosity is the cause of the no-slip condition since the fluid has zero relative velocity at the very immediate surface. Because of the continuum flow assumption at very small Knudsen numbers, from the no-slip condition to the freestream condition, there must exist a zone where the fluid develops its velocity profile. That *zone* is the boundary layer (δ), and there the viscous effects are predominant. The boundary layer contains the viscous flow regime. Outside this thin layer, the flow can be modeled as inviscid since all transport phenomena can be neglected (mass, momentum and energy). In a zero pressure gradient (ZPG) flow represented by the canonical flat plate, the boundary layer thickness is defined as that wall-normal distance where the local velocity reaches 99% of the freestream velocity (Cengel and Cimbala 2014). It is the most popular convention; however, some investigators from the fluid dynamics community prefer a 95% criterion. Figure 1.4 shows an example of a surface with zero velocity relative to the fluid flowing on top. The flow needs a wall-normal distance for the velocity to change from zero to the freestream value.

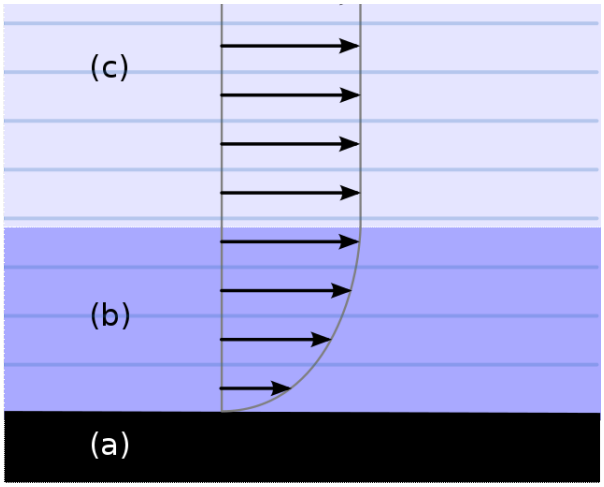


Figure 1.4: Boundary layer example. (a) The surface at zero velocity relative to the flow (b) Boundary layer thickness (c) Freestream region (image by MikeRun, distributed under a CC-BY-SA-4.0 license).

Generally speaking, flow parameters in the boundary layer theory are non-dimensionalized. This approach allows a fair comparison among different Reynolds number flows. Two major scaling functions are employed in turbulence: inner and outer scaling. In inner scaling, the characteristic length scale is defined as ν/u_τ. Here, ν is the fluid kinematic viscosity, and u_τ is the friction velocity expressed as $\sqrt{\frac{\tau_w}{\rho}}$, being τ_w the wall shear stress. Furthermore, the friction velocity represents the characteristic velocity in inner units. The friction temperature is the characteristic thermal scale in inner units, and it is defined as $\theta_\tau = q_w/(\rho c_p u_\tau)$, where q_w is the wall heat flux and c_p is the isobaric fluid specific heat. On the other hand, outer scaling considers the boundary layer thickness, δ, as the length scale (White 2011); whereas, the freestream velocity, U_∞, and freestream temperature, T_∞, are usually utilized as the characteristic velocity and thermal scale in outer scaling, respectively. For example, distance, velocity, pressure, and temperature are usually non-dimensionalized, as shown in Equations 1.14 and 1.15 for inner and outer scaling, respectively. Where x is a spatial distance in one dimension, T_s is the temperature at the surface, and P_{ref} is the pressure at the reference location.

$$\left\{x^+ = x\frac{u_\tau}{\nu} \quad U^+ = \frac{U}{u_\tau} \quad T^+ = \frac{T}{\theta_\tau} \quad P^+ = \frac{P-P_{ref}}{\rho u_\tau^2}\right\} \tag{1.14}$$

$$\left\{x^* = \frac{x}{\delta} \quad U^* = \frac{U}{U_\infty} \quad T^* = \frac{T-T_s}{T_\infty - T_s} \quad P^* = \frac{P-P_{ref}}{\rho U^2}\right\} \tag{1.15}$$

1.2.4 Boundary Layer Detachment

Flow separation or flow detachment does not precisely imply that the actual fluid ceases contact with the body where the flowing occurs (Simpson 1989). The term is used when the flow's boundary layer interaction with the body is modified by a reverse flow directly at the wall (Schlichting and Gersten 2017). This event most often gives rise to turbulent fluctuations, causing their enhancement. It is essential to highlight that flow separation can be induced either by geometrical singularities, for example, in the presence of sharp corners, or by smooth geometry variations, such as those occurring over a curved wall (Mollicone et al. 2017). Depending on the magnitude, the mere presence of a very strong adverse-pressure gradient (APG) might be sufficient to cause flow detachment and represents by far the most undesirable situation in the momentum/scalar transport. Flow separation is critical since the boundary layer parameters experience sudden changes due to an abrupt thickening of the rotational flow region or backflow close to the wall (Simpson 1989), which may cause a critical reduction in the performance of engineering devices, e.g., pressure drag increase in airfoils or heat transfer decrease in turbine blades.

For this reason, flow separation has been the topic of several theoretical, experimental and numerical studies in the past few decades (Simpson 1985) (Simpson et al. 1987). Unfortunately, some gray zones or *"terra incognita"* still exist and need further investigation, particularly in the near-wall region where most of the experimental techniques and turbulence models exhibit severe limitations. Flow separation at high Reynolds numbers is one of the most challenging types of turbulent flows and remains one of the significant unsolved fluid mechanics problems, according to Williams (1977). Simpson (1989) in his well-known review of turbulent boundary layer separation, claimed *"the effects of significant wall curvature are not well described quantitatively, although most separation cases occur on curved wall"*. According to Patrick (1987), more reliable data and large-scale models are required to better define the turbulence structure of the backflow region in very strong streamwise APG, which is responsible for the boundary layer detachment. Despite the significant progress performed in the last twenty years, modeling efforts for separated flows have been hindered due to the lack of information on the mechanisms that control the separation of the boundary layer by large-scale structures.

Figure 1.5 shows a schematic of the 2D turbulent boundary layer separation in low-curvature and flat surfaces according to Simpson (1989). The dashed line indicates $U = 0$, and detachment (D) occurs where the time-averaged wall-shear stress is zero. Moreover, the incipient detachment (ID) and intermittent transitory detachment (ITD) take place at locations where instantaneous backflow occurs 1% and 20% of the time,

respectively. As the boundary layer encounters an APG, the near wall region flow decelerates until some backflow first takes place at the ID point. It is worth highlighting that this reverse flow is attributed to *the transport of outer momentum toward the wall by the large-scale or coherent structures* (Simpson 1989). The major reasons why separation in turbulent flows is so problematic to model and measure are: (i) Large separation of scales, (ii) Highly non-linear phenomena, (iii) Challenges to determine near wall pressure fluctuations, and (iv) The inner and outer regions are highly communicated. Of all the different types of flow separation that occur around streamlined bodies, certainly, the one caused by wall curvature is the most intriguing. Curvature effects are usually an order of magnitude greater than would be predicted by dimensional analysis (Bradshaw 1973), with a significant impact on the Reynolds stresses (Moser and Moin 1984) (Moser and Moin 1987). One of the major goals of the present study is to shed some light on turbulent boundary detachment caused by surface curvature.

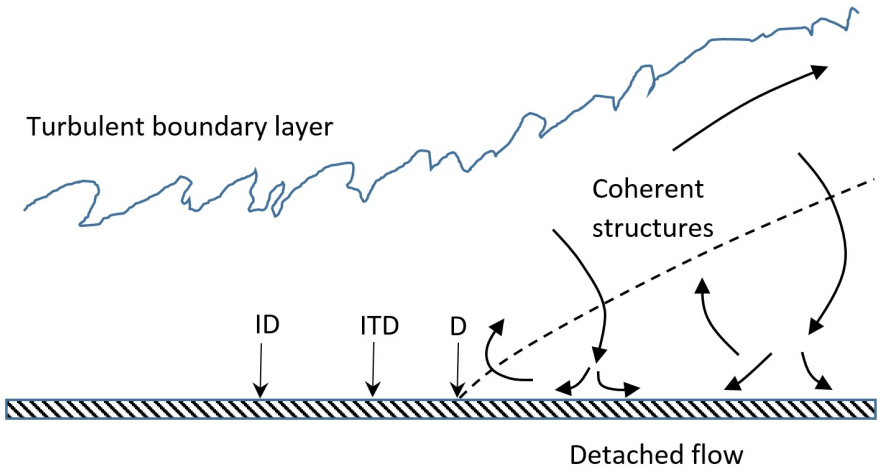


Figure 1.5: Cartoon of 2D turbulent boundary layer separation (adapted from Simpson (1989)).

1.2.5 Passive Scalar Transport

The turbulent transport of passive scalars is crucial in many industrial applications of technological importance, such as in turbine-blade film cooling, heat transfer in electronic/mechanical devices, chemicals dissolved in gases, and contaminant/humidity dispersed in atmospheric flow, to name a few examples. Furthermore, a passive scalar is defined as a diffusive contaminant that exists in such a low concentration in a flow that it does not affect the dynamics of the fluid motion (Warhaft 2000). However, that low concentration

of passive scalar is sufficient to cause a significant impact on energy expenditures, air pollution, and the design of chemical processes. On the contrary, an active scalar directly affects the flow's characteristics, for example, temperature as an active scalar means the fluid's properties such as viscosity and density are in function of the temperature, thus, the modeling must take into account the temperature contribution even in the momentum equation. The transport phenomenon in real-situation flows usually occurs under complicated external conditions, such as pressure gradients (favorable and adverse), complex geometry (concave/convex surface), high Reynolds numbers, and spatially-developing turbulent boundary layers (SDTBL). A passive scalar is mathematically modeled by its diffusive coefficient. For example, a fluid has its own viscous diffusion rate (kinematic viscosity ν). Suppose the diffusivity is related to a molecular or mass contaminant, the corresponding parameter is the Schmidt number (Sc) which measures the ratio of viscous diffusion over mass diffusion. For a passive temperature scalar, the parameter to be involved is the Prandtl number ($Pr = \frac{\mu c_p}{k} = \frac{\nu}{\alpha}$), which measures the ratio of viscous diffusion over thermal diffusion (α) (White 2011). Furthermore, in this thesis, several molecular Prandtl numbers are considered in order to cover nearly the entire spectrum of passive scalars, i.e., by assuming low Pr=0.20, unitary Pr=0.71, and high Pr=2.00.

1.3 Computational Fluid Dynamics (CFD)

Computational Fluid Dynamics (CFD) is a computer-aided numerical simulation tool for the analysis of systems involving fluid flows, with a crucial presence nowadays (Versteeg and Malalasekera 2007). This powerful engineering tool has extended the applicability from the aerospace industry to automotive, biology, chemical, marine, nuclear, and power generation industries, to name a few examples (Moukalled et al. 2016). CFD provides an approximated solution to complex numerical problems, and the accuracy of the solution depends on the discretization method selected, which generally transforms differential equations into algebraic equations. Some mainstream discretization methods are the Finite Difference, the Finite Element, and the Spectral methods. Nonetheless, the Finite Difference method is the most attractive and easy to understand by engineers because it is based on the physical conservation principle of relevant properties (Versteeg and Malalasekera 2007). There is a particular type within the Finite Difference called the Finite Volume Method, the most commonly used in CFD. This method integrates the flow's governing equations (by iterative algebraic approaches) over the complete computational domain discretized into finite control volumes. One big help that CFD provides to the technology enablers is the fast and accurate solutions to the tedious Navier-Stokes Equations, derived in the previous sub-section §1.2.1.

Numerically solving the governing equations for turbulent flows requires computational power, and often these resources are not readily available. Therefore, formulating models that significantly reduce the computational power requirement and maintain high precision between results and proper solution is a simply appreciable benefit (Paeres et al. 2022b). Among the existing CFD categories, the most commonly known are RANS (Reynolds-averaged Navier-Stokes), LES (large eddy simulation) and DNS (direct numerical simulation) (Moukalled et al. 2016). As stated in the name, DNS directly solves the governing formulation of all fluid flows, the Navier-Stokes Equations; it does not use turbulent models and requires the most significant computational resources. The act of directly solving DNS gives the most exact solution, where the errors are simply influenced by computational limitations like rounding and memory. However, the computational power required for an acceptable representative time-spatial sample is significantly elevated. To overcome this problem, some flows models approaches have been proposed to reduce the computer resource requirements with a high impact. RANS and LES implement these turbulent models maintaining high precision. Figure 1.6 shows independent examples of the quality of the details obtained in each of the approaches mentioned. Although DNS shows tremendous high-resolution results, a CFD user might consider LES or RANS approaches with minor field quality results for the sake of reducing computational resources and process time while reaching solutions accurate enough for applications. Furthermore, with the existence of hybrid RANS-LES methods, maybe these are the ultimate industrial methods for fluid flow simulations in the engineering field, according to Schlichting and Gersten (2017).

If one desires to perform DNS, it is mandatory to employ a highly scalable and efficient flow solver, not to mention if the idea is to predict spatially developing turbulent boundary layers (SDTBL) implying accurate turbulent time-dependent inflow conditions. Therefore, DNS requires researchers' skills, expertise, and abilities in HPC and parallel programming not only during the running but also in the postprocessing stage (Lagares et al. 2021). With the advent of powerful supercomputers, it has become easier to push the boundaries of turbulent boundary layer simulations at higher Reynolds numbers via DNS (Lagares and Araya 2021) (Schlatter, P. and Orlu, R. 2010). On the other hand, LES models the dynamics and influence of Kolmogorov scales but is reasoned in using a borderline as a spatial filter; large-scale motions (called large eddies) are computed directly, and only the small-scale motions (considered eddies of thermal energy dissipation) are modeled, resulting in a significant reduction in computational resources compared to the DNS approach. However, this effort reduction depends on the analyzed geometry: it is well known that in wall-bounded flows even the "inertial subrange scales" located in the near wall region could be very small. Thus, the computing and running effort reduction by considering LES with respect to DNS could be limited to one order of

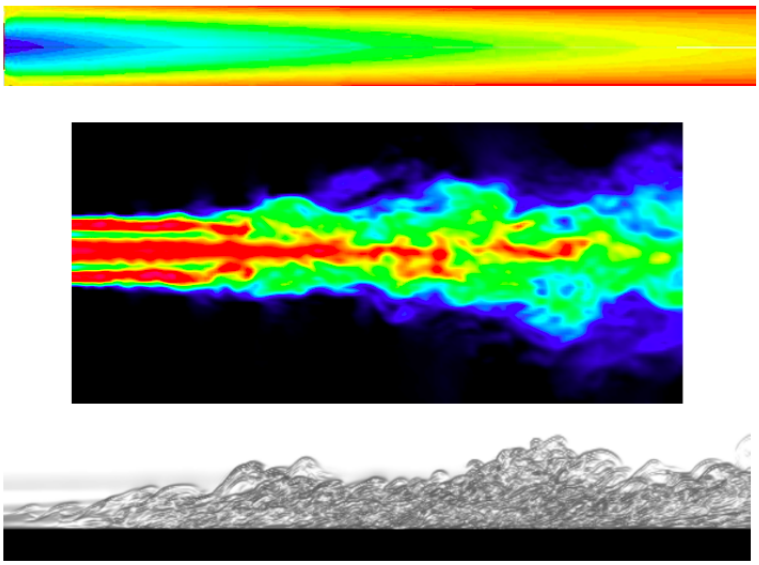


Figure 1.6: Qualitative comparison of the details for three CFD simulation methods. Top for RANS and middle for LES (images respectively by Jpolihronov and Charlesreid1, distributed under a CC BY-SA 3.0 license). The bottom shows DNS (image by Andreas Babucke, distributed under a CC BY 3.0 DE license).

magnitude, at most (Paeres, Lagares, and Araya 2022b). It is worth highlighting that any LES approach must be performed over a three-dimensional domain and unsteady simulations. Araya and Lagares (2022) performed Implicit LES (iLES) over supersonic flat-plate spatially developing turbulent boundary layers at moderate Reynolds numbers. First order statistics were somehow well captured by iLES in comparison with DNS at similar flow conditions. However, a poor performance of iLES was detected in the near wall region for second-order statistics since it significantly overpredicted peak values of streamwise velocity fluctuations, and consequently, Reynolds shear stress peaks in the buffer layer. A good agreement of the resolved Reynolds stresses by iLES with respect to DNS predictions was observed in the outer region. More importantly, the total computational resource saving by iLES was estimated to be approximately in the order of 100 in comparison with DNS results.

Among these three CFD categories, RANS is the approach where the simulation framework is highly simplified since the whole power spectra of flow fluctuations are modeled. Despite its simplicity and limitations on complex geometries, RANS may supply important insight into the flow (Moukalled et al. 2016). Within RANS literature for turbulent flows, there are four most popularly used turbulent models, namely: $K-\epsilon$, standard $K-\omega$, $K-\omega$ shear stress transport, and Spalart–Allmaras.

OpenFOAM® is an open-source CFD software extensively used in academia. One of the reasons is the benefits of controlling the coded equations and enabling custom turbulence models for the user. This software has implemented the just mentioned simulation approaches' general schemes and the most common turbulent models. Because of these characteristics, OpenFOAM® has been selected as the CFD tool for the presented work. The main course to be performed is a two-dimensional (2D) scenario of a curved hill with the RANS formulation. An existent DNS database from our research group (HPCV 2018) will be used for the RANS validation in canonical flat-plate turbulent boundary layers. Furthermore, experimental data (such as skin friction, velocity, pressure coefficient, and pressure distribution) from Baskaran, Smits, and Joubert (1987) over a curved hill is also utilized as a validating tool.

1.3.1 Reynolds Averaged Navier-Stokes Equations (RANS)

RANS is based on the Reynolds decomposition, i.e., the mathematical notion that each instant property can be written as the sum of mean and fluctuations fields; for example, $U = \overline{U} + U'$, U being an instantaneous flow property, $\overline{U}$ the average of U in time and U' the fluctuations of U. In RANS, the mean flow field which is time-averaged (or ensemble-averaged in flows with time-dependent boundary conditions), is resolved whereas the fluctuations field is modeled. The concept in RANS is to only focus on the mean flow and the effects of turbulence on its properties since for most engineering applications it is not necessary to resolve the turbulent fluctuations' details (Versteeg and Malalasekera 2007). For general compressible fluid flows, the governing equations (expressed following Einstein notation) are:

Conservation of mass:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \left(\rho U_j\right)}{\partial x_j} = 0 \tag{1.16}$$

Conservation of momentum:

$$\frac{\partial (\rho U_i)}{\partial t} + \frac{\partial \left(\rho U_i U_j\right)}{\partial x_j} = -\frac{\partial P}{\partial x_i} + \frac{\partial}{\partial x_j}\left(\mu \frac{\partial U_i}{\partial x_j}\right) + S_{m,i} \tag{1.17}$$

Conservation of passive scalar, being T a scalar assumed to follow a similar form of the Navier-Stokes Equations:

$$\frac{\partial (\rho T)}{\partial t} + \frac{\partial \left(\rho T U_j\right)}{\partial x_j} = \frac{\partial}{\partial x_j}\left(k \frac{\partial T}{\partial x_j}\right) + S_T \tag{1.18}$$

where S are the source terms. If the steady-state, the incompressibility assumption, and the Reynolds decomposition are applied, the RANS models for Newtonian fluids have the following equations:

Continuity equation:

$$\frac{\partial \overline{U}_j}{\partial x_j} = 0 \tag{1.19}$$

Conservation of momentum:

$$\rho \frac{\partial (\overline{U_i U_j})}{\partial x_j} = -\frac{\partial \overline{P}}{\partial x_i} + \frac{\partial}{\partial x_j}\left(\overline{\tau_{ij}} - \rho \overline{U_i' U_j'}\right) + \overline{S_{m,i}} \tag{1.20}$$

Note in Equation (1.20) contrasted with Equation (1.17) how, by definition, the derivatives of mean variables with respect to time disappear but still is the emergence of a new term for the fluctuations field variables. Additionally, the density parameter at each partial derivative is pulled out due to the constant assumption of the incompressible condition. The term containing the velocity field fluctuations (i.e., $-\rho \overline{U_i' U_j'}$) is directly related to the well-known Reynolds stresses, and it gets that name because it only emerged after applying the Reynolds decomposition. In 1877, Boussinesq proposed that Reynolds stresses might be proportional to mean rates of deformation (Versteeg and Malalasekera 2007). In RANS, the mean parameters are computationally solved, but the fluctuations terms remain unknown. It is precisely here where the Boussinesq hypothesis acts, enabling the models' proposal for estimating the unknown terms and balancing the equations. For a detailed derivation of the RANS equations refer to Appendix A.

1.3.2 Turbulence Modeling

As said early in the current section, there are four most popular RANS models used. To understand these models is essential to describe four important parameters briefly. The first and principal is called turbulent kinematic viscosity, generally written as ν_t. Also known as eddy viscosity per density (μ_t/ρ), it is responsible for identifying the correct energy dissipation due to flow turbulence based on Boussinesq postulated (Versteeg and Malalasekera 2007). ν_t is an apparent property for viscosity but accounts for the turbulent phenomenon. The second parameter is the turbulent kinetic energy (K), which is the kinetic energy per unit mass of turbulent fluctuations and is mathematically defined by:

$$K \equiv \frac{1}{2}\overline{U_i' U_i'} = \frac{1}{2}(\overline{u'^2} + \overline{v'^2} + \overline{w'^2}) = \frac{3}{2}\overline{U'^2} \tag{1.21}$$

The third is the turbulent dissipation ratio (ϵ), defining the ratio at which turbulent kinetic energy is converted to internal thermal energy. The mathematical description of ϵ is:

$$\epsilon \equiv \nu \overline{\frac{\partial U_i'}{\partial X_j} \frac{\partial U_i'}{\partial X_j}} \tag{1.22}$$

The last parameter is the specific dissipation rate of turbulent energy (ω). The variable ω is the ratio of the turbulent kinetic energy converted into internal thermal energy per unit of time and volume. For precisely having Hertz units, it is also known as frequency turbulence average. The relationship between ω to K and ϵ is generally written as Equation (1.23) where β^* is a model constant that, for example, can be around 1.0 or even 0.09.

$$\omega = \frac{\epsilon}{K\beta^*} \tag{1.23}$$

The turbulent model $K-\epsilon$ initially proposed by Launder and Spalding (1983) is considered a high Reynolds number model and this indicates that its specialty is in good results away from near-wall (Lee 2018). This model is uses the scalar conservation Equation (1.18) to estimate K and ϵ. In essence, it says that the rate of change of K (or ϵ) plus the transport of K (or ϵ) by convection is equal to the transport of K (or ϵ) by diffusion plus the production rate of K (or ϵ) minus the destruction ratio of K (or ϵ). The estimations are performed following Equation (1.25) for K and for ϵ Equation (1.26). Then, the eddy viscosity (μ_t) is calculated with Equation (1.24), where E_{ij} is the strain-rate tensor and the coefficients' common values are contained in Table 1.1.

$$\frac{\mu_t}{\rho} = \nu_t = \frac{\beta^* K^2}{\epsilon} \tag{1.24}$$

$$\frac{\partial(\rho K)}{\partial t} + \frac{\partial(\rho K U_i)}{\partial X_i} = \frac{\partial}{\partial X_j}\left[\frac{\mu_t}{\sigma_K}\frac{\partial K}{\partial X_j}\right] + 2\mu_t E_{ij}E_{ij} - \rho\epsilon \tag{1.25}$$

$$\frac{\partial(\rho \epsilon)}{\partial t} + \frac{\partial(\rho \epsilon U_i)}{\partial X_i} = \frac{\partial}{\partial X_j}\left[\frac{\mu_t}{\sigma_\epsilon}\frac{\partial \epsilon}{\partial X_j}\right] + C_{1\epsilon}\frac{\epsilon}{K}2\mu_t E_{ij}E_{ij} - C_{2\epsilon}\rho\frac{\epsilon^2}{K} \tag{1.26}$$

$$2E_{ij}E_{ij} = (\frac{\partial U_i}{\partial X_j} + \frac{\partial U_j}{\partial X_i})\frac{\partial U_i}{\partial X_j} \tag{1.27}$$

Table 1.1: Term's representation and common coefficients values for $K-\epsilon$ model

β^*	σ_K	σ_ϵ	$C_{1\epsilon}$	$C_{2\epsilon}$
0.09	1.00	1.30	1.44	1.92

The standard $K-\omega$ turbulence model was initially proposed by Wilcox (1988). This model is known as a low-Reynolds model, which means being good at predicting results in the near-wall (Lee 2018). Similar to the $K-\epsilon$ model, the standard $K-\omega$ turbulence model uses two scalar conservation equations to predict K and ω. The turbulent kinetic energy and specific dissipation rate are solved with Equations (1.29) and (1.30), respectively. The model's common coefficients values are shown in Table 1.2 while turbulent kinematic viscosity is calculated with Equation (1.28). Considering incompressible flow:

$$\nu_t = \frac{K}{\omega} \tag{1.28}$$

$$\frac{\partial K}{\partial t} + U_j \frac{\partial K}{\partial X_j} = \tau_{ij} \frac{\partial U_i}{\partial X_j} - \beta^* K\omega + \frac{\partial}{\partial X_j}\left[(\nu + \sigma^* \nu_t)\frac{\partial K}{\partial X_j}\right] \tag{1.29}$$

$$\frac{\partial \omega}{\partial t} + U_j \frac{\partial \omega}{\partial X_j} = \alpha_\omega \frac{\omega}{K} \tau_{ij} \frac{\partial U_i}{\partial X_j} - \beta\omega^2 + \frac{\partial}{\partial X_j}\left[(\nu + \sigma \nu_t)\frac{\partial \omega}{\partial X_j}\right] \tag{1.30}$$

Table 1.2: Recommended values for the coefficients in Standard $K-\omega$ model

α_ω	β	β^*	σ	σ^*
5/9	3/40	0.09	0.5	0.5

Menter (1994) proposed his shear stress transport model, also known as $K-\omega$ SST, where it unifies the advantages of the standard $K-\omega$ and $K-\epsilon$ models without inheriting their weaknesses (Lee 2018). Proposing a hybrid model, he reuses Reynolds stress computation and K-equation from Wilcox's original $K-\omega$ model and transforms the ϵ-equation into a ω-equation. A decade later, Menter, Kuntz, and Langtry (2003) published a revised version of the turbulent model. The updated version, commonly known as SST-2003 (Rumsey 2021), is shown in Equations (1.31), (1.32), and (1.36) and the revised model coefficients are shown in Table 1.3. In the additional functions, Equations (1.33)–(1.35), y is the distance to the nearest wall. The SST also uses the scalar conservation formulation twice, for that reason, the three previous turbulent models are classified as two-equation models.

$$\frac{\partial(\rho K)}{\partial t} + \frac{\partial(\rho U_i K)}{\partial X_i} = \tilde{P}_K - \beta^* \rho K\omega + \frac{\partial}{\partial X_i}\left[(\mu + \sigma_K \mu_t)\frac{\partial K}{\partial X_i}\right] \tag{1.31}$$

$$\frac{\partial(\rho\omega)}{\partial t} + \frac{\partial(\rho U_i \omega)}{\partial X_i} = \frac{\alpha \tilde{P}_K}{\nu_t} - \beta\rho\omega^2 + \frac{\partial}{\partial X_i}\left[(\mu + \sigma_\omega \mu_t)\frac{\partial \omega}{\partial X_i}\right] + 2(1-F_1)\rho\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial K}{\partial X_i}\frac{\partial \omega}{\partial X_i} \tag{1.32}$$

$$F_1 = \tanh\left(\left(\min\left[\max\left(\frac{\sqrt{K}}{\beta^* \omega y}, \frac{500\nu}{y^2\omega}\right)\frac{4\rho\sigma_{\omega 2}K}{CD_{K\omega}y^2}\right]\right)^4\right) \qquad F_2 = \tanh\left(\left(\max\left(\frac{2\sqrt{K}}{\beta^* \omega y}, \frac{500\nu}{y^2\omega}\right)\right)^2\right) \tag{1.33}$$

$$CD_{K\omega} = \max\left(2\rho\sigma_{\omega 2}\frac{\partial K}{\omega \partial X_i}\frac{\partial \omega}{\partial X_i}, 10^{-10}\right) \qquad \tilde{P}_K = \min\left(\mu_t \frac{\partial U_i}{\partial X_j}\left(\frac{\partial U_i}{\partial X_j} + \frac{\partial U_j}{\partial X_i}\right), 10\beta^* \rho K \omega\right) \tag{1.34}$$

$$S_{ij} = \frac{1}{2}\left(\frac{\partial U_i}{\partial X_j} + \frac{\partial U_j}{\partial X_i}\right) \qquad \gamma = \gamma_1 F_1 + (1 - F_1)\gamma_2 \qquad \forall \qquad \gamma \equiv \text{constant} \tag{1.35}$$

$$\mu_t = \frac{a_1 \rho K}{\max(a_1\omega, F_2\sqrt{2S_{ij}S_{ij}})} \tag{1.36}$$

Table 1.3: Revised values for the coefficients in $K - \omega$ SST model

α_1	β_1	σ_{K1}	$\sigma_{\omega 1}$	α_2	β_2	σ_{K2}	$\sigma_{\omega 2}$	β^*	a_1
5/9	3/40	0.85	0.5	0.44	0.083	1.0	0.856	0.09	0.31

Very different is the SA turbulent model, proposed Spalart and Allmaras (1992) and classified as a one-equation model since it only solves for a working variable of the turbulence model, $\tilde{\nu}$. The principal parameter $\tilde{\nu}$ is computed by the scalar conservation Equation (1.37). The eddy viscosity μ_t is calculated with Equation (1.38), where f_{v1} is a wall-dumping function that causes zero value at the wall and reaches unity at high Reynolds number cases (Versteeg and Malalasekera 2007). Additionally, $\tilde{\Omega}$ is the production rate of $\tilde{\nu}$ related to the local mean vorticity as shown in Equation (1.39). The Spalart-Allmaras model has earned high popularity and is often used in the aerospace and aerodynamics industry for its high reduction in computational effort (Milidonis et al. 2018; Zhao et al. 2022). In addition, empirical adjustments to the eddy-viscosity turbulence model were performed in Spalart and Shur (1997) to account for system rotation and streamline curvature effects based on the idea of Knight and Saffman (1978). In Smirnov and Menter (2009), the rotation-curvature correction of Spalart and Shur (1997) was extended to the SST model. The new proposed version (SST-CC) was successfully

tested on both wall-bounded and free shear turbulent flows with rotation and/or surface curvature. Of current interest, in a 2D channel flow with a U-turn, the SST-CC model performed well; however, it was stated that the model still predicted a slow flow recovery downstream of the separation bubble. The SA turbulent model's recommended coefficient values are in Table 1.4.

$$\frac{\partial \tilde{\nu}}{\partial t} + \nabla \cdot (\tilde{\nu} U) = \frac{1}{\sigma_v} \nabla \cdot \Big((\nu + \tilde{\nu}) \nabla \tilde{\nu} + C_{b2} (\nabla \tilde{\nu})^2 \Big) + C_{b1} \tilde{\nu} \tilde{\Omega} + C_{w1} \Big(\frac{\tilde{\nu}}{\kappa y} \Big)^2 f_w \tag{1.37}$$

$$\mu_t = \rho \tilde{\nu} f_{v1} \tag{1.38}$$

$$\tilde{\Omega} = \sqrt{\frac{1}{2} \left(\frac{\partial U_i}{\partial X_j} - \frac{\partial U_j}{\partial X_i} \right)^2} + \frac{\tilde{\nu}}{(\kappa y)^2} f_{v2} \qquad f_{v1} = \frac{(\tilde{\nu}/\nu)^3}{(\tilde{\nu}/\nu)^3 + C_{v1}^3} \qquad f_{v2} = 1 - \frac{(\tilde{\nu}/\nu)}{1 + (\tilde{\nu}/\nu) f_{v1}} \tag{1.39}$$

Table 1.4: SA turbulent model coefficients recommended values

σ_v	κ	C_{b1}	C_{b2}	C_{w1}	C_{v1}
2/3	0.4187	0.1355	0.622	$C_{b1}/\kappa^2 + (1 + C_{b2})/\sigma_v$	7.1

1.3.3 Passive Scalar Transport Modeling

According to Pirozzoli et al. (2016), the transport equation for passive scalar fields (e.g., temperature or mass concentration) with finite diffusion is given by Equation (1.40). Note the similarity with Equation (1.18), where the two terms on the left side are the material derivative of the transported variable (i.e., T or θ), and on the right side, S_T is a suitable source term. Equation (1.18) is expressed for a general compressible flow; that is why ρ and k are inside the derivatives. However, because Equation (1.40) is for incompressible flows, the first term of the right side has its parameters outside the derivatives. Pr is either the Prandtl number or the Schmidt number; meanwhile, note that Re contributes ν to the numerator and to the denominator a characteristic length and the velocity field U_i. The product of Re and Pr is called the Péclet number (Pe), regardless if Pr is the Prandtl number or Schmidt number. The Péclet number is defined as the ratio of the advective transport rate of a physical quantity to its diffusive transport

rate (Moukalled et al. 2016).

$$\frac{\partial T}{\partial t} + \frac{\partial (TU_i)}{\partial x_i} = \frac{1}{RePr}\frac{\partial}{\partial x_i}\left(\frac{\partial T}{\partial x_i}\right) + S_T \tag{1.40}$$

If the passive scalar to model is temperature, the thermal diffusivity coefficient (α) can be obtained from the viscous diffusivity (kinematic viscosity ν) and the parameter Prandtl number by division. By recalling $\alpha = \frac{k}{\rho c_p}$, and the emergence of the fluctuations term after applying Reynolds averaging, the Equation (1.41) is obtained.

$$\rho c_p \frac{\partial (\bar{T})}{\partial t} + \rho c_p \frac{\partial (\bar{T}\bar{U}_i)}{\partial x_i} = \frac{\partial}{\partial x_i}\Big(k\frac{\partial}{\partial x_i}(\bar{T}) - \rho c_p \overline{U_i' T'}\Big) + \bar{S_T} \tag{1.41}$$

In response and accounting for the effects of both molecular and turbulent transport, an effective thermal diffusion coefficient (α_{eff}) is introduced for simplicity. Therefore, the temperature as a passive scalar is modeled by Equation (1.42).

$$\frac{\partial T}{\partial t} + \nabla \cdot \Big(U_i T\Big) = \alpha_{\text{eff}} \nabla^2 T \tag{1.42}$$

From a computational point of view, flow separation or recirculation is one of the most challenging problems to tackle due to its high level of intermittency, huge turbulent scale separation, evident three-dimensionality, and unsteadiness. Several LES and RANS studies (in isolated or in combined "hybrid" form) have been carried out in the past few decades with the purpose of shedding some light on the boundary layer detachment problem caused by strong adverse streamline curvature-driven pressure gradient (Chaouat 2017; Radhakrishnan et al. 2006; Purohit et al. 2021; Zhang et al. 2020). Major conclusions in previously cited works can be summarized as follows: overall, LES and RANS approaches may predict quite well first-order statistics in turbulent boundary layers subject to strong APG, which significantly degrades for higher-order statistics (Paeres et al. 2022b).

1.4 Data Visualization in XR

1.4.1 Scientific Visualization

According to Friendly and Denis (2001), scientific visualization *"is primarily concerned with the visualization of 3D+ phenomena (architectural, meteorological, medical,*

biological, etc.), where the emphasis is on realistic renderings of volumes, surfaces, illumination sources, and so forth, perhaps with a dynamic (time) component". More suggested literature includes Hansen and Johnson (2011), Nielson, Hagen, and Müller (1997) and McCormick (1987). A half-century ago of flow visualization, the only technique available to describe coherent structures in turbulent flows was smoke and dye injection (Brown and Roshko 1974; Winant and Browand 1974). However, visualization techniques have substantially evolved along with the world's technology, spanning all disciplines (William R. Sherman and Bushell 1997). Today visualization usually leans on computers because, in 20 years, the computing capacity has increased over three orders of magnitude (Paeres et al. 2021).

1.4.2 Virtual Reality, Augmented Reality and MR

To clarify the difference between Virtual Reality (VR) and Augmented Reality (AR) is vital to start with the concept "virtuality continuum", introduced 28 years ago by Milgram and Kishino (1994). They stated the continuum with four regions and argued that the taxonomy should be based on three sub-concepts. (i) "Extent of World Knowledge: The amount of information bleeding from the real environment into a virtual environment". (ii) "Reproduction Fidelity: an attempt to quantify the image quality of the displays". Moreover, (iii) "Extent of Presence Metaphor: a term to encapsulate the degree of immersion". Years later, Flavián et al. (2019) proposed a new taxonomy in 5 divisions, as shown in the following Figure 1.7.

Real Environment	**Augmented Reality**	**Pure Mixed Reality**	**Augmented Virtuality**	**Virtual Environment**
Complete physical world	Virtuality overlaps reality	Virtuality and reality are coalesced	Reality overlaps virtuality	Complete digital world

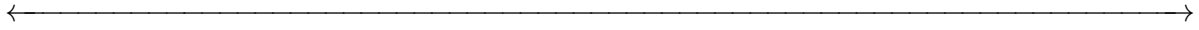


Figure 1.7: Virtuality continuum, reproduced from Flavián et al. (2019).

From left to right in Figure 1.7, the **Real Environment** is the physical world where technology is not needed for its existence. Second is the **Augmented Reality** (AR), where virtuality overlays in reality yet does not interact with the whole physical world. In the midway is the **Pure Mixed Reality**, where virtuality and reality are fused, explained as the generation of virtual objects with the complete awareness of the real environment. The fourth is **Augmented Virtuality** (AV), where reality overlaps with virtuality, for example, a computer-generated display crowded with digital images but controlled by physical commands. The last taxonomy is the **Virtual Environment**,

also referred to as Virtual Reality (VR), where the complete environment is virtual. Any input from the real world has to be translated into a digital expression. The junction from AR to AV is called Mixed Reality (MR). At the same time, Extended Reality (XR) covers AR to VR, commonly used as an unspecified reality mix (Paeres et al. 2021).

1.4.3 XR Benefits and Applications

A high number of studies conclude that XR enhances the education process (Saidin et al. 2015). Radianti et al. (2020) reviewed how to optimize VR applications for educational purposes, and it was concluded that students tend to retain better information after VR exercises. The performance is improved after applying what had been learned. Consequently, one could believe MR technology needs to be developed until reaching a fully immersive feeling. Rosenblum (2000) outlined key areas where this technology's development should focus, from ultra-lightweight haptic feedback sensors to realistic interfaces hard to distinguish between realities. Besides improving sensors for better virtual environment perspectives, the hardware and software need to be fast and smooth enough without having data memory size constraints (Albert et al. 2017; Elbamby et al. 2018).

Virtual Reality applications fully immerse the user in a 3D virtual world (Craig et al. 2009). VR and MR technologies attempt to blur the line between the physical world and the digital environment. Furthermore, it provides options in the research environment for difficult or costly experiments. Regarding flow visualization, such technologies are beneficial for visual representation. It enables a researcher to perform scientific analysis and to gain a better insight into complex flows in complicated geometries hard to perceive with the naked eye at a fraction of the effort and cost required in a physical setup (Paeres et al. 2021).

Game engine platforms and VR technology can perfectly merge for scientific procedures. For example, Brookes et al. (2020) researched the development of an experimental platform for behavioral scientists with VR and the game engine software Unity 3D. Brookes et al. designed and proposed the Unity Experiment Framework (UXF) app providing the *"nuts and bolts that work behind the scenes of an experiment developed within Unity ... that allow logical encoding of the common experimental features required by the behavioral scientist."* (Brookes et al. 2020). UXF gave an optional graphical user interface (GUI), allowing the experimenter to design examinations easily, define dependent and independent variables, and edit the display interface; without requiring high-level knowledge of coding languages. A helpful feature of UXF was also the cloud-based platform; in experiments performed remotely from a laboratory, the software collected the data and stored it

in servers. To confirm the UXF reliability, the authors performed a study between adults and children to examine their postural sway in response to an oscillating virtual room. The validation experiment resulted in the children presenting less postural stability than adults, as was expected from previous research done by others (Brookes et al. 2020); also endorsing the integration of XR and game engines into scientific methodologies.

For starters, the term "virtual" is not new. References to this concept can be dated back to 1991 (Byron and Levit 1991), and the idea was refined and presented again by Bryson (1993). These early works were pioneering but held back by the limited technology of the time. Today, Virtual Wind Tunnel (VWT) is the name of an entirely virtual environment presented in this work and created to enhance data visualization by immersively interacting and glancing with any virtual object, ideally CFD simulation results (Paeres, Santiago, Lagares, Rivera, Craig, and Araya 2021). On the other side, FlowVisXR leans more toward Augmented and Mixed Reality because it is a device application to invoke virtual objects in the physical world and enable awareness of it (Paeres et al. 2021). Applications of XR for the present work are: (i) The creation of a virtual wind tunnel, (ii) The in-house app FlowVisXR, and (iii) The development of a complete methodology for obtaining XR virtual objects from CFD results.

1.4.4 Virtual/Augmented Reality in Flow Visualization

The procedure of flow visualization with extended reality starts with the information post-processing of CFD results (generally ASCII or binary format). The data is transformed via programming (i.e., Python or MatLab scripts) or visualization toolkit (i.e., ParaView) into a file type of 3D virtual objects. During this data transformation, it is expected that the data might require manipulation and filtering, such as iso-contour for the desired parameter field. 3D virtual objects have wide options of file formats, just to name a few: Alembic (.abc), Autodesk FBX (.fbx), GL Transmission Format (.gltf), Wavefront OBJ (.obj), and CAD (computer-aided design) software extensions.The correct process depends on compatibility with the software and devices used for the visualization. Unfortunately, the options for immersive visualization like VR and AR are quite limited. The present study uses the extension USD (Universal Scene Description). This type of file (i.e., .usdz) has been developed by Apple and Pixar and launched in 2018. Although USDZ files were exclusively designed for AR applications, it will be shown and demonstrated how to employ them in VR environments, while keeping the perk of being a binary file format, meaning a file's storage-size reduction. The virtual environments for the present work are going to be developed with the Unity game engine.

Regarding the Virtual Wind Tunnel, a workstation (see Fig. 1.8(a)) gathers the information of all input gadgets. Controllers, head-mounted display (HMD), and base stations sensors identify the user's spatial movement and field of view (FOV), as seen in Figure 1.10, and render the virtual objects that belong inside the FOV. Images in the display have to be rendered to a speed of at least 30 frames per second (fps) to achieve an immersive impression of the virtual environment. Once the frames are rendered, they are sent to the HMD, which outputs a stereoscopic image providing the user with the 3D illusion of depth. Fully immersed visualizations are performed on the HTC Vive VR kit (see Fig. 1.8(b)).

(a) (b)

Figure 1.8: (a) Workstation Dell Tower 5810 and (b) HTC Vive VR toolkit.

On the other hand, obtaining AR flow visualization starts with the same post-processing and data transformations as VR. However, it differentiates depending on the final goal and interactive degree with the virtual objects. Suppose the idea is only to invoke virtual objects in the real world via iOS devices. In that case, the USDZ file extension is the best option because Apple devices have a built-in app capable of reading the USDZ files. After achieving the final extension, the last step is to share the file with the devices to use. However, if the device to use is not iOS (e.g., Android and HoloLens), a compatible app will be needed. Here, the Unity game engine comes to the rescue. The benefits of designing the apps with Unity are not only extending the applicability to smartphones and XR devices but also it gives opportunities to develop more interactive manipulations,

such as image recognition and much more. Microsoft HoloLens (see Fig. 1.9) apps and other XR app creations, will be shown and discussed in the corresponding Chapter §3.

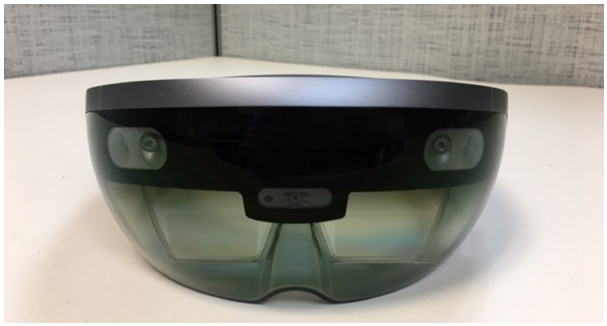


Figure 1.9: Image of Microsoft HoloLens 1st gen. used for AR applications.

1.5 Project Description

1.5.1 Expected Outcomes

With the completion of this work, the outcomes intended to be achieved are summarized below:

- Enhance acquired knowledge of turbulent momentum/passive-scalar boundary layers subject to strong wall-curvature-driven pressure gradient with eventual flow separation.
- Evaluate turbulence model's performance in passive scalar transport subject to strong wall curvatures.
- Sharpen the reader's fluid mechanics knowledge and scientific data visualization with Extended Reality (XR).
- Develop a tool for immersive visualization of CFD data results, enabling manipulation of virtual objects (i.e., 2D and 3D) for clear examination.

1.5.2 Objectives

The principal objectives of the proposed body of work are as follows:

- Perform laminar and turbulent CFD simulations over a flat plate as turbulence precursors.

- Simulate the turbulent flow over a curved hill in a replicated scenario from the original experiments of Baskaran, Smits, and Joubert (1987) (see Figure 2.1), considering the inclusion of a passive scalar and evaluating the effect of different Prandtl numbers.
- Exploit OpenFOAM/C++ toolbox's capabilities in pre-processing utilities (mesh development) and numerical solvers for the solution of the governing equations of viscous flow.
- Develop numerical post-processing tools in Python for boundary layer parameter calculation.
- Define a simple, but innovative, methodology for VR and AR data visualization.
- Present the Virtual Wind Tunnel (see Figure 1.10) and FlowVisXR app as the enhancement approaches for scientific data visualization.

Figure 1.10: User visualizing a flow in the VWT.

1.5.3 Intellectual Merit

The focus of the present body of work is on incompressible turbulent boundary layers subject to strong surface curvature (i.e., convex and concave curvatures) prone to flow separation. Additionally, several passive scalars are considered and scrutinized under such external conditions in complex flow. *CFD RANS modeling will shed important light*

on the boundary layer detachment due to wall curvature as well as on the passive scalar transport by paying a fraction of computational resources as compared to DNS and LES approaches (Paeres, Lagares, and Araya 2022b), (Paeres, Lagares, and Araya 2022a). Furthermore, this study considers state-of-the-art visualization techniques such as Virtual and Augmented Reality originally applied to flow visualization (Paeres, Lagares, Santiago, Craig, Jansen, and Araya 2020) (Paeres, Santiago, Lagares, Rivera, Craig, and Araya 2021) (Paeres, Lagares, and Araya 2021), which can be extended to any other discipline once developed (i.e., biology, math, architecture, etc.).

1.5.4 Broad Impact

Genuinely understanding turbulent flow separation is of great importance in naval, aeronautical, and other engineering applications. While drag reduction and heat transfer control are commonly in the optimization focus, it is crucial to discern a turbulent flow from the shear or boundary layer point of view. Due to the large flow gradients present inside boundary layers, most of the transport phenomena occur in that thin region. Flow separation, very often, is unwanted and can seriously reduce the performance and efficiencies of engineered devices such as aircraft and turbines. In worst-case scenarios, may cause accidents and fatalities.

There are times when there is an urgent need to move, rotate or manipulate an object to appreciate it visually. It is also sometimes necessary to represent these objects on different scales, enlarged or miniaturized, to internalize concepts and phenomena accurately. The Virtual Wind Tunnel (VWT) development can undoubtedly appease the expression "Seeing is believing." (Paeres et al. 2021). As the necessary equipment to create a virtual environment can be challenging to obtain for certain people, an affordable option is the development of the FlowVisXR app since it can visualize the virtual objects from a smartphone device. The FlowVisXR app and the VWT will be excellent open-source solutions to enrich the data visualization process truly.

At the end of this work, the reader will learn about boundary layer separation, passive scalar transport, and the basic data management procedure to perform XR data visualization in their own methods or approaches.

Chapter 2
Turbulent Boundary Layers Subject to Surface Curvature

This chapter presents and discusses the CFD results of a flow over a canonical flat plate and over a complex geometry (i.e., a curved hill). For turbulent boundary layers, closure equations are used in the RANS approach by testing two well-known turbulence model: the Shear Stress Transport (SST) model (Menter 1994) and the Spalart-Allmaras (SA) model (Spalart and Allmaras 1992). The purpose of running a turbulent flat plate or precursor is to obtain more realistic and developed flow solutions at the domain inflow of the curved hill scenario. This is aligned with one of the objectives for the present thesis to replicate the results of Baskaran, Smits, and Joubert (1987) experiments, the flow's developing entry length calculations are performed to match the flow conditions and the spatial reference points. Although the experiments by Baskaran, Smits, and Joubert (1987) only consisted of air as the working fluid ($Pr = 0.71$), the present work will extend the Prandtl number scope (i.e., $Pr = 2.0$ and $Pr = 0.2$). After scrutinizing the flows and passive scalars' behaviors over the curved hill, major conclusions of this chapter based on statistical analysis can be found in §2.5. Furthermore, the scientific visualization is performed in the next Chapter §3.

2.1 Assessment, Motivation and Approach

In the case of turbulent flow over a curved hill, separation zones will occur due to the strong adverse-pressure gradient (APG) or flow deceleration caused by the presence of convex wall curvature. The assessment's inspiration is from the experimental work of Baskaran, Smits, and Joubert (1987). In their work, a horizontal air stream at 20 m/s passed over a simple curved convex hill of 1.284 m long with a radius of 1.08 m. The protuberance's entrance and exit had a concave surface of -0.40 m and -0.48 m, respectively. Figure 2.1 shows an example of the geometries. In Baskaran et al. (1987), the momentum boundary layer parameters (first and second-order statistics) were measured.

Boundary layer thickness, displacement thickness, momentum thickness, and other integral parameters calculated were used to describe the flow's characteristics and behavior. Other relevant information was also measured, such as the wall's streamwise pressure and skin friction, velocities profiles at specific streamwise locations, and Reynolds stresses. Their main objective was "to study the response of turbulent boundary layers to sudden changes in surface curvature and pressure gradient" (Baskaran et al. 1987). Although the experimental data of the velocity field are used as a point of validation of our numerical approach, it is worth highlighting that the present study also considers temperature as a passive scalar which was not explicitly accounted for in the original experiment.

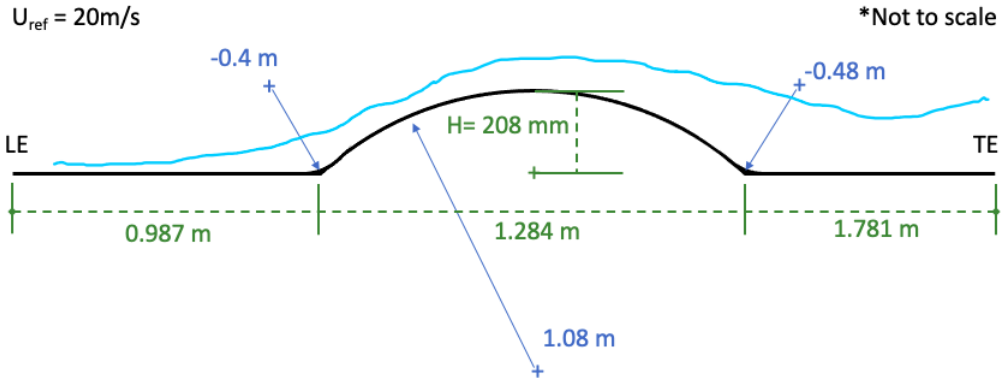


Figure 2.1: Curved hill diagram (partially reproduced from Baskaran et al. (1987)).

Two wall scenarios were considered to properly begin with the simulations and establish the accurate reference point in the numerical approach compared to the experiments by Baskaran et al. (1987). The first scenario was a horizontal flat plate with two flow conditions: (i) Laminar flow (i.e., streamwise inlet velocity of 1 m/s) and (ii) Turbulent flow with the same inlet velocity as in Baskaran et al. (1987). The second scenario was the turbulent 20 m/s streamwise inlet velocity over the curved hill. The 2D numerical assessment of turbulent flow was performed with scope to the flow's separation zone velocity and with the presence of temperature in the simulation as a passive scalar. Turbulence is inherently a three-dimensional unsteady phenomena, particularly boundary layer detachment. However, the curved hill domain is modeled as a 2D phenomenon at a "statistically steady state" for simplification purposes, while still capturing the mean flow distortion (and Reynolds shear stresses) in complex geometries. Future work would involve an unsteady 3D analysis via LES and publish elsewhere. Therefore, the present scenario is taking advantage of the flow homogeneity in the spanwise direction (z), whereas the flow is highly anisotropic along the streamwise and wall-normal direction (in particular, the latter one). The numerical approach used a RANS (Reynolds-averaged Navier-Stokes)

approach. Using open-source CFD software OpenFOAM®, the velocity field was solved in the scenario of considering a molecular Prandtl of 0.71 and 0.90 turbulent Prandtl number (i.e., air as the working fluid) and two turbulent models $K-\omega$ shear stress transport (SST) (Menter 1994) and Spalart-Allmaras (SA) (Spalart and Allmaras 1992). The inclusion of temperature is done based on the theory of passive scalar transport as in Li et al. (2009).

Both turbulence models were not selected arbitrarily. It has been chosen the best representatives from one and two-equation turbulence models. The Spalart-Allmaras (SA) model (or one-equation model) has been described to be incomplete by Wilcox (2006) since any turbulence model would require at least two scales: a velocity scale and a length scale. However, the SA model has shown simplicity, robustness, and versatility in attached turbulent boundary layer simulations (Spalart and Allmaras 1992; Spalart and Rumsey 2007; Shapiro et al. 2021; Raheem et al. 2019) including high-speed turbulent boundary layers (Paciorri et al. 1998; Lagares et al. 2021). Previously mentioned studies have emphasized a degradation in the SA model's accuracy for more complex geometries with flow separation. In particular, the SA model under-predicted the size of the separation bubble. On the other hand, the hybrid SST model by Menter involves blending two well-known two-equation turbulence models: the standard $K-\omega$ model in the near wall region and the $K-\epsilon$ model in the outer region and freestream of the boundary layer to overcome the strong freestream sensitivity of the $K-\omega$ turbulence model. It is worth highlighting that the SST eddy viscosity model involves a further improvement based on the Johnson–King model, which assumes that the transport of the main turbulent shear stresses is critical in the simulations of strong adverse-pressure gradient (APG) flows prone to boundary layer detachment.

In summary, and to the best of author's knowledge, the combined effect of flow separation by strong streamline curvature-driven pressure gradients and passive scalar transport has not been fully addressed in the past, and knowledge on this matter is relatively scarce. Furthermore, the performance of widely used RANS models on very strong surface curvatures and separated flow is of great interest to the broader computational fluid dynamics community. To this end, the Reynolds-averaged Navier-Stokes equations will be resolved over a curved hill with experimental data points to assess the performance of two widely used CFD models. Furthermore, it will assess where each model excels and where they fail to capture the physics of the flow using the experimental data as ground truth. Some of the research questions to be addressed in this work are as follows: (i) Are the evaluated eddy viscosity models able to capture the outer peaks in Reynolds shear stresses (i.e., $<u'v'>$ where the angle brackets mean time-averaged) as well as the inclination of the constant shear layer caused by strong APG?, (ii) Is the flow separation

bubble dominated by positively correlated u' and v'?, (iii) What is the temperature profile trend inside the flow separation bubble?, (iv) Is the Reynolds analogy preserved in strong streamline curvature-driven pressure gradients?, and (v) Is the flow showing quasi-laminar features in zones where is highly decelerated?

2.2 Laminar Flow Over a Flat Plate

A simulation of laminar flow over a flat plate has been conducted to self-validate OpenFOAM® utilization and begin the journey of CFD programming for the presented work. A 2D domain of 3.85 m in the streamwise direction and 0.183 m in the wall-normal direction was designed with a short entry length of no-wall presence followed by a 3.7 meters-long flat plate with the no-slip condition. Three mesh resolutions were defined to perform a mesh dependency test. Coarse, mid and fine resolutions were respectively (900×134), (1800×268), and (3600×536) cells count. Figure 2.2 shows the resulting domain of medium mesh with cells' volume size and velocity field contour. Figure 2.3 shows fine mesh cells' distribution example. Even though it is a 2D simulation, the CFD method used the Finite Volume Method. The inlet boundary condition for velocity is 1 m/s. At the same time, the temperature is 323.15 K, and the pressure's boundary condition is zero-gradient. Top, laterals, and outlet boundary conditions were set zero-gradient for the three fields (i.e., velocity, temperature, and pressure) except for the outlet's pressure gauge fixed to 0 value. Iso-thermal condition with the value of 293.15 K was specified at the wall.

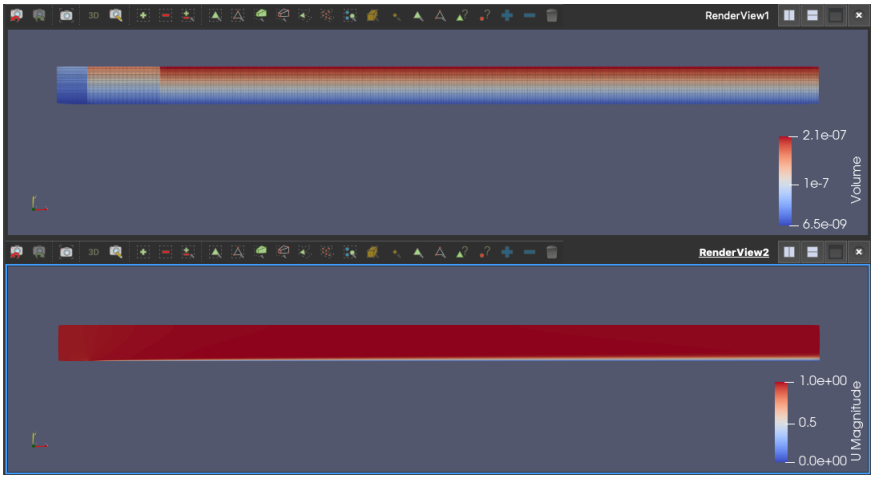


Figure 2.2: Cells' volume size and velocity field contour of medium mesh.

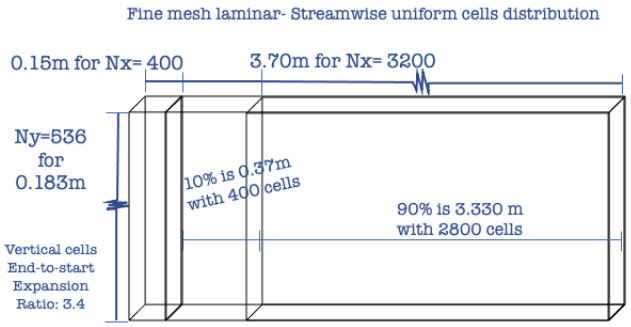


Figure 2.3: Schematic of the fine mesh dimensions and resolution.

Results were compared to the theoretical Blasius solution for a zero-pressure-gradient (ZPG) flat plate. With the understanding that the velocity distribution at the very-near-wall is similar for almost all laminar and turbulent flows, one prominent parameter is the dimensionless wall-normal distance (y^+). y^+ is the y distance non-dimensionalized in wall units by the friction velocity (u_τ) and the fluid kinematic viscosity, ν. The formulation used was $y^+ = yu_\tau/\nu$, where the friction velocity is defined as $u_\tau = \sqrt{\tau_w/\rho}$ and τ_w is the wall shear stress. Figure 2.4 shows the first off-wall cell's height in the dimensionless wall-normal distances (Δy^+) along the streamwise direction of the flat plate. Even though the mesh's initial design aimed to be $\Delta y^+ = 0.3$ (as it happens near the outlet), it can be seen that at the inlet's critical zone, maximum values were close to $\Delta y^+ = 1$. Usually, in CFD simulations of wall-bounded flows, $\Delta y^+ < 1$ means the simulation had enough near-wall quality to solve the boundary layer correctly.

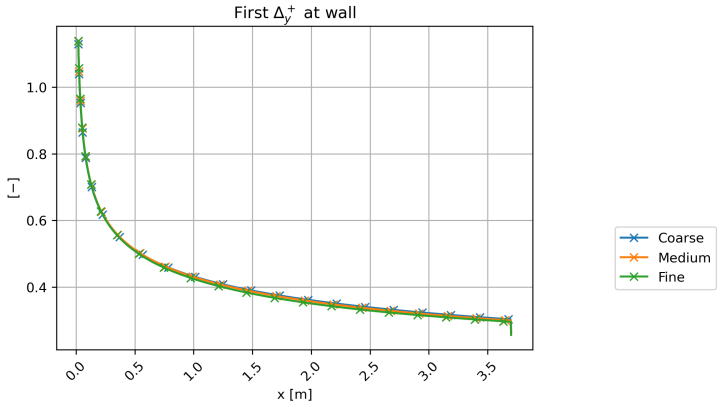
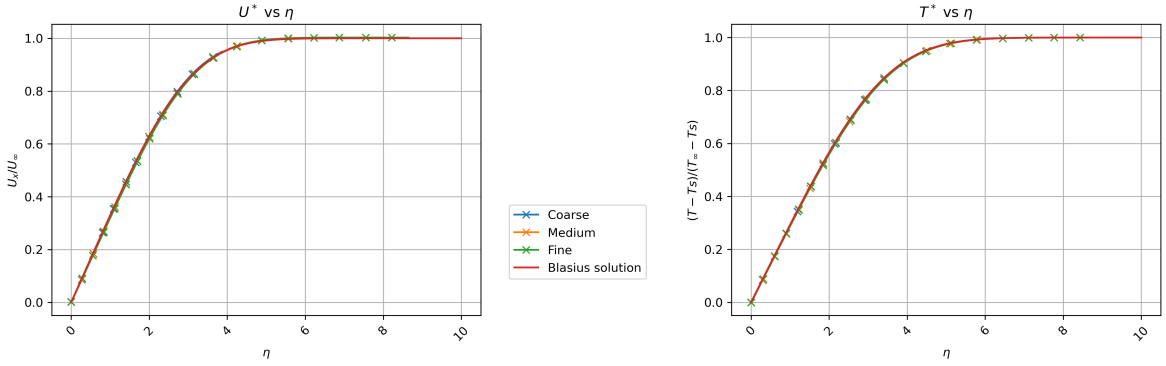


Figure 2.4: First off-wall cell height in wall units.

The Blasius solution is presented by a third-order non-linear ordinary differential equation

(ODE) using non-dimensioned similarity variables. In other words, the independent variables (i.e., spatial coordinates x & y) can be united and reduced from 2 to 1, creating the variable η. By definition, a streamline is a flow's path where fluid's properties remain constant along the path. This dimensionless η represents each flow's streamlines and is calculated by the formula $\eta = y\sqrt{U_\infty/x\nu}$. Because of Blasius's brilliance, he defined the function $f(\eta)$ to be solved so that the first derivative resulted in being $f' = U_x/U_\infty$. Following his cleverness, if the ODE is expanded also a dimensionless temperature T^* can be obtained, defined by $T^* = (T - T_s)/(T_\infty - T_s)$. The η is related to streamline length distance, U^* and T^* with the velocity and temperature profiles, respectively. The streamwise velocity profile, U_x/U_∞ or $f' = U^*$ in Blasius variables, is shown in Figure 2.5(a). An excellent agreement of the three meshes can be observed with the quasi-analytical Blasius solutions over a laminar boundary layer. Clearly, the curves take flat behavior at the order of $\eta \approx 5$. Beyond that, i.e., for $\eta > 5$, the streamwise fluid velocity equals that of the freestream, thus $U^* = 1$. Figure 2.5(b) shows the dimensionless temperature with an outstanding performance of the coarse, medium, and fine mesh as compared to the thermal Blasius' solution for $Pr = 0.71$. Here, a thermal flat curve over $\eta \approx 5.5$ can be seen, which is consistent with the Prandtl number lower than one, implying a faster diffusion of the thermal field as compared to the momentum's diffusion.



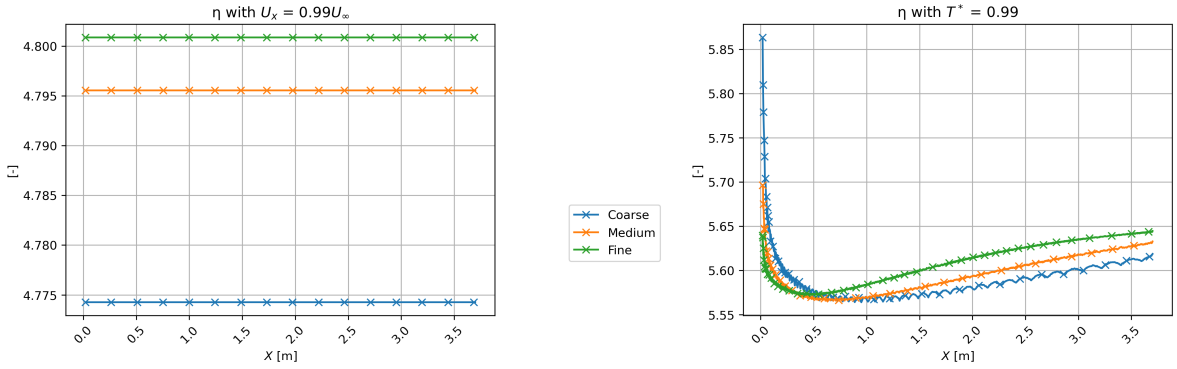
(a) (b)

Figure 2.5: Normalized results of laminar flow over flat plate compared to the Blasius solution: **(a)** Streamwise velocity profile; **(b)** Temperature profile.

As is seen in Figure 2.6(a), the η values corresponding to the velocity boundary layer's streamline are constant over the plate length and approach the expected value ≈ 5. Similarly, Figure 2.6(b) has the the η values corresponding to the temperature boundary layer's streamline. Although the η values are not constant for the temperature's boundary layer, it can be noted how finer the mesh is, the precision increments, showed by sooner

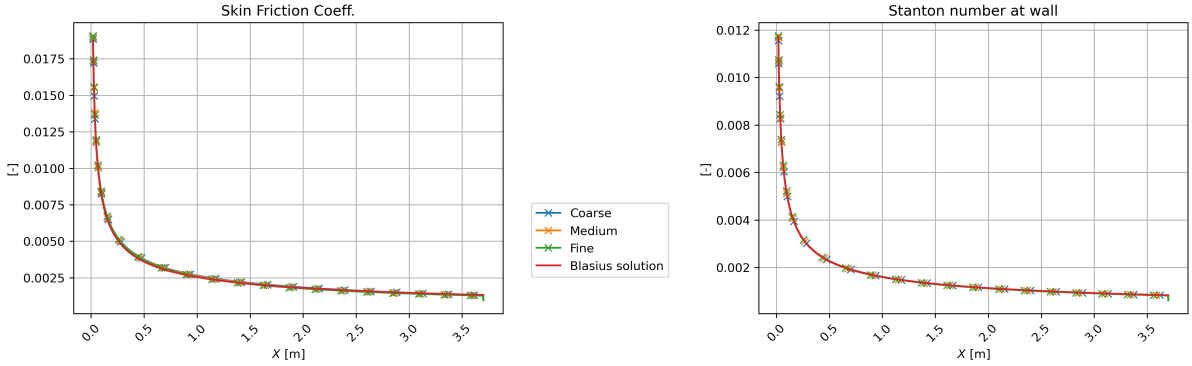
curve attenuation. The result values get more precise and accurate for both velocity and temperature dimensionless as the mesh's refinement increases, without causing a relevant effect on the real dimensional parameters.

Figure 2.7 displays the skin friction coefficient (C_f) and the Stanton number (St). Where St is a dimensionless number representing the ratio of heat transfer by forced convection to the heat capacity of the fluid, a suitable parameter to use in thermal simulation validation. After observing the Figures 2.4, 2.5, 2.6, and 2.7, due to the results' consistency and similarity to the theoretical plotted curves, this laminar CFD job has been considered accomplished and ready to proceed in turbulent flow scenarios.



(a) (b)

Figure 2.6: η values at the boundary layers (99% of the freestream) over the wall: **(a)** Streamwise velocity profile; **(b)** Temperature profile.



(a) (b)

Figure 2.7: Skin friction coefficient and Stanton number results of laminar flow: **(a)** Skin friction coefficient, **(b)** Stanton number.

2.3 Turbulent Flow

2.3.1 Flow Solver and Numerical Schemes

The strategy implemented for the turbulent simulations was to use the algorithm called semi-implicit method for pressure-linked equations consistent (SIMPLE-C) with a residual control value of 1×10^{-8} for all the variables or fields. For the specific-pressure field (P), the solver selected is called the geometric agglomerated algebraic multigrid solver along with a Gauss-Seidel smoother (GAMG), a relative tolerance of 0.1, and a tolerance of 1×10^{-8}. For the passive scalar field, it was used the preconditioned bi-conjugate gradient (PBiCGStab) solver with a simplified diagonal-based incomplete LU preconditioner (DILU); relative tolerance was set to 0, and the tolerance 1×10^{-9}. Lastly, for all the other variables (i.e., U, $\tilde{\nu}$, K, ω), the solver adopted was a smooth solver accompanied by a symmetric Gauss-Seidel smoother, and tolerances values equal to the P field. The relaxation factor for P was 0.3 meanwhile 0.7 for U, $\tilde{\nu}$, K, and ω.

As for the numerical schemes, the following were designated for the time, gradient, Laplacian, interpolation, surface-normal gradient, and wall distance calculation, respectively: steady state, Gauss linear, Gauss linear corrected, linear, corrected, and mesh wave method. The divergence schemes chosen were Gauss linear for the non-linear stress and the effective viscous stress; Gauss linear upwind for the divergence of the surface scalar field (ϕ) with the passive scalar; bounded Gauss linear upwind for ϕ with U and with $\tilde{\nu}$; while for the divergence of ϕ with K and ω it was used bounded Gauss limited linear 1.

All the solvers and numerical schemes determined for the work presented were already available in the open-source software OpenFOAM® version 7 as a standard option without any modification in the algorithms or implementation. For details on the approach of any method mentioned in this section, readers are referred to Moukalled et al. (2016) and Versteeg and Malalasekera (2007).

2.3.2 Boundary Layer Detection Based on Potential Flow

A potential flow is characterized as an inviscid and irrotational velocity field, described as the gradient of a scalar function. On the other hand, a boundary layer develops from strong viscous interactions within a small region, usually caused by the presence of a surface or vorticity source. The potential flow satisfies the Laplace equation as,

$$\nabla^2 \Phi = 0 \tag{2.1}$$

where Φ is the scalar function, the potential velocity field is defined as $U_p = \nabla\Phi$, with the flow being irrotational, i.e., $\nabla \times U_p = 0$. However, since for any arbitrary velocity flow field expressed using finite, floating-point arithmetic, the divergence is not strictly zero, OpenFOAM® implements a numerical scheme following Poisson's equation to enforce a solenoidal field and conditions required for the irrotationality of the flow. This issue is not exclusive to OpenFOAM® as it is an inherent limitation of finite-precision arithmetic using floating points. The scheme can be expressed as follows, where ϕ is the face flux of a finite volume as,

$$\nabla^2\Phi = \nabla \cdot \phi \tag{2.2}$$

As done in Baskaran et al. (1987), the edge of the boundary layer is determined based on the potential flow theory in curved geometries. It is the standard procedure used in the fluid dynamics theory since surface curvature locally induces a streamwise pressure gradient, making it challenging to identify the beginning of the inviscid zone above the turbulent boundary layer. Given that viscous interactions are not present in potential flows, the boundary layer can be described as the region where these viscous interactions can not be neglected (a somewhat simplistic description, but it suffices for the proposed methodology). This work hypothesizes that the edge of the boundary layer can be detected by searching for the region where the potential flow and the real viscous flow cross paths and exhibit similar behavior. According to Baskaran et al. (1987), the edge of the boundary layer thickness, δ, is defined as "that wall distance at which the dynamic pressure is 99% of the free-stream value". In the present work, is applied the following criteria for identifying the edge of the boundary layer:

$$U^2_{s,Viscous} \geq (99\%)U^2_{s,Potential} \quad \& \quad \left|\frac{\partial^2 U_s}{\partial n^2}\right|_{Potential} \geq \left|\frac{\partial^2 U_s}{\partial n^2}\right|_{Viscous} \tag{2.3}$$

Here, U_s represents the wall-parallel component of the fluid flow, and n is the wall-normal coordinate. It is important to mention that the fulfillment of both criteria according to Equation (2.3) ensures an acceptable identification of the viscous–inviscid interface or boundary layer edge, particularly in the recirculation flow zone. The second derivative of U_s in the wall-normal direction describes the local curvature of the velocity profile, which must be zero in the inviscid region. Figure 2.8 provides a pictorial representation of the intuition behind the approach employed in the present work. Note, in Figure 2.8(a), that the viscous RANS solution starting from the wall highly disagrees with the potential solution. However, as the distance from the wall grows, viscous RANS tends asymptotically to the potential solution until further, in the free-stream, the viscous RANS solution is consistently very similar to the potential solution. Moreover, with the cyan color curve of

station $s = 2500\ mm$ (at the separation bubble), note how the velocities first matched in a wall level without actually both solutions having the asymptotical behavior, confirming that the second criterion in Equation (2.3) is necessary. With the possibility of multiple wall levels with the solution's paths crossed, the issue is solved by looking at the U_s slopes. Figure 2.8(b) shows the wall-normal derivative of the streamwise velocities; note the asymptotical and overlapping behavior of the viscous solution towards the zero value and potential solution again. It can be intuitive that after some n distance, the absolute value of the wall-normal second derivative of viscous solution is going to be equal to the corresponding derivative of the potential solution; or at least, a smaller second derivative for the viscous solution compared to the potential solution due to the tendency to zero showed in both solutions. Readers are referred to Appendix B for a demonstration of the boundary layer thickness capturing algorithm implemented in the Python script *postProcessing.py*.

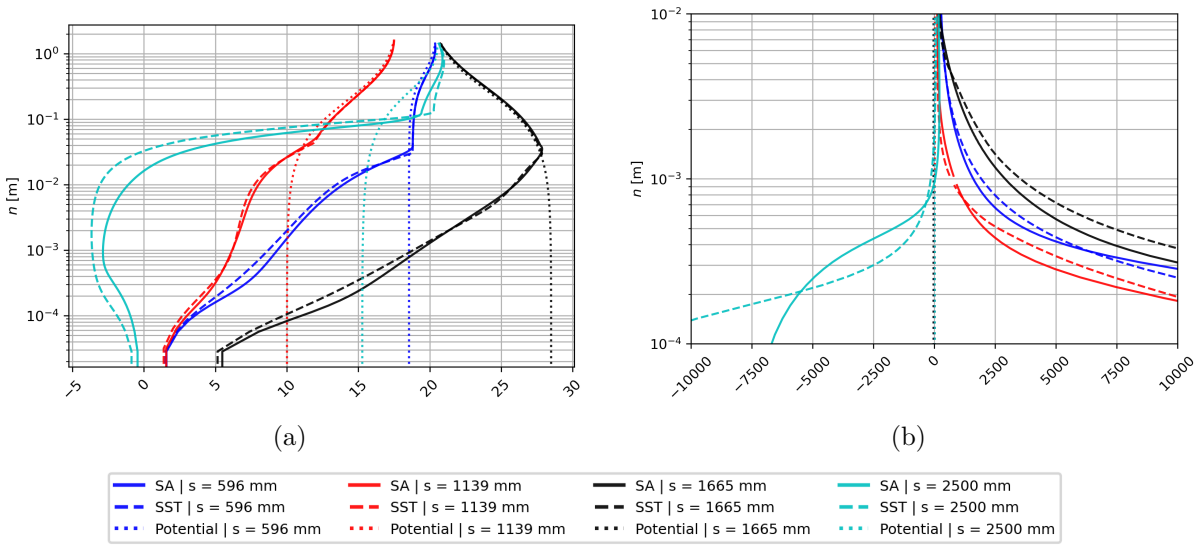


Figure 2.8: (**a**) Potential flow and RANS wall normal profiles, U_s in m/s; (**b**) wall-normal derivative of the streamwise velocity, $\frac{\partial U_s}{\partial n}$ in $1/s$.

2.3.3 Turbulent Inflow Generation

Before beginning with the curved hill cases, a scenario of the turbulent flow over a flat plate was considered using SST and SA models. For all cases presented in this work, the classical no-slip condition was prescribed at the wall surfaces for the velocity fields; meanwhile, an isothermal condition of 293.15 K was prescribed for the thermal field. The top-surface boundary condition was set via a zero-gradient of all flow parameters

(shear-less wall) and a zero-pressure gauge value at the outlet boundary. For the SA's conditions, $\tilde{\nu}$ and ν_t were specified as zero at the walls, while $\tilde{\nu} = 1.5\times\ 10^{-5}\ m^2/s$ and $\nu_t = 1.1\times\ 10^{-5}\ m^2/s$ for both constant inlet and domain's initial state. For the SST's conditions, the turbulent intensity was assumed as 1%, giving $K = 0.06m^2/s^2$ and $\omega = 4000\ 1/s$ to be applied as Dirichlet conditions at the inlet plane and walls, while $\nu_t = 1.5\times\ 10^{-5}\ m^2/s$ was employed to initialize numerical predictions.

In the flat plate or turbulence inflow generator scenario, the total domain's length was set to 3.85 m with an initial 0.15 m length as a slip-condition. Three meshes were designed, namely coarse, medium, and fine mesh, as described in Table 2.1. Furthermore, vertically end-to-start cell expansion ratios (γ_y) were 4140, 3000, and 1900, respectively. The vertical cell expansion ratio, γ_y, is defined as the dimensionless relationship between the end cell near the top surface to the start cell in the near-wall region, i.e., $\gamma = (\Delta y)_{\text{end}}/(\Delta y)_{\text{start}}$. In the inlet plane, freestream conditions were set, i.e., 20 m/s for the horizontal velocity and 323.15 K for the static temperature (passive scalar).

In the curved hill scenario, mean flow solutions from the flat plate were extracted and injected as velocity and temperature inlet profiles to overcome the unrealistic freestream inlet profiles and avoid a lengthy inlet developing section. In addition, this recycling strategy allows better control of the inlet boundary layer parameters as a solid requirement for validating numerical predictions against wind-tunnel experiments. As shown later in this section, these "recycled" profiles are validated. Like in the flat plate scenario, the curved hill had three meshes: coarse, medium, and fine mesh; all three meshes with height divided into two blocks for an efficient cell distribution control. The coarse mesh had 200 cells in the first 25% of vertical distance with $\gamma_y = 1000$ and the other 75% of height with 100 cells in uniform distribution to comply with an acceptable mesh aspect ratio. In the medium mesh, the nearest block to the wall was only 3.5% of total height with 200 cells and $\gamma_y = 386$, while the outer region was composed of 200 cells with $\gamma_y = 12$. Similarly, the fine mesh possessed the vertical 3.5% and 96.5% split with the same end-to-start cell-expansion ratio as the medium mesh, but both regions were populated with 400 cells. Horizontally, all three meshes were divided into five blocks; Table 2.2 shows the resolution and the horizontal cell distribution.

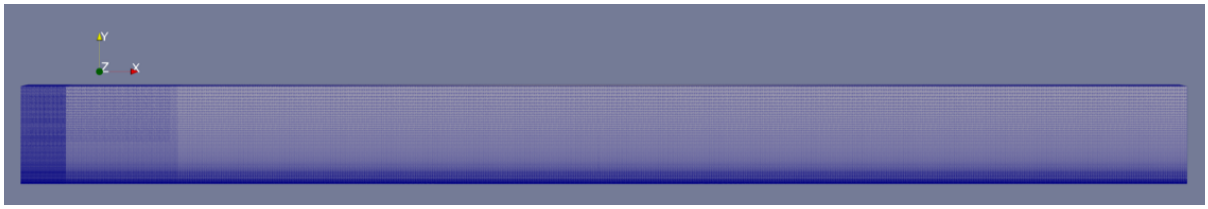
Table 2.1: Flat Plate Meshes' Cells Horizontal Distribution; from left to right.

	(*Horizontal cells count ; γ_x*)			(*horizontal* × *vertical*)
Id	**block 1**	**block 2**	**block 3**	**Total Cells**
Coarse	75 ; 1	75 ; 1	525 ; 1	675 × 100
Medium	100 ; 1	100 ; 1	700 ; 1	900 × 134
Fine	150 ; 1	150 ; 1	1050 ; 1	1350 × 201

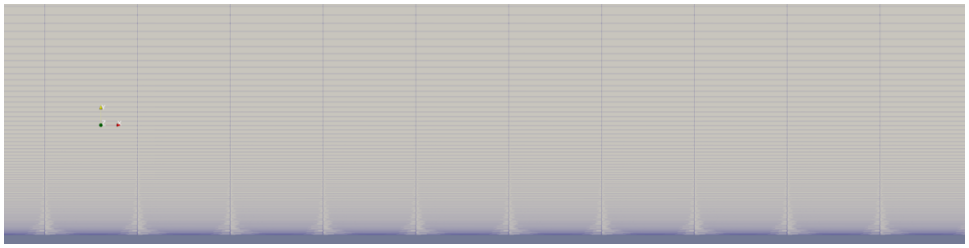
Table 2.2: Curved Hill Meshes' Cells Horizontal Distribution; from left to right.

	(*Horizontal cells count ; γ_x*)					(*horizontal* × *vertical*)
Id	**block 1**	**block 2**	**block 3**	**block 4**	**block 5**	**Total Cells**
Coarse	250 ; 0.1	75 ; 1	350 ; 1	75 ; 1	250 ; 10	1000 × 300
Medium	450 ; 0.25	150 ; 1	700 ; 1	150 ; 1	425 ; 4	1875 × 400
Fine	675 ; 0.25	225 ; 1	1050 ; 1	225 ; 1	638 ; 4	2813 × 600

Figures 2.9(a) and 2.10(a) display a full view of the fine meshes for the flat plate and curved hill domains, while the corresponding figures (b) are the near wall region close-ups, respectively. Figures 2.11 and 2.12 show schematics of the computational domains for the flat plate as well as for the curved hill. Figure 2.13(a,b) show the ZPG regions' near-wall resolution for the flat plate and curved hill, also respectively.

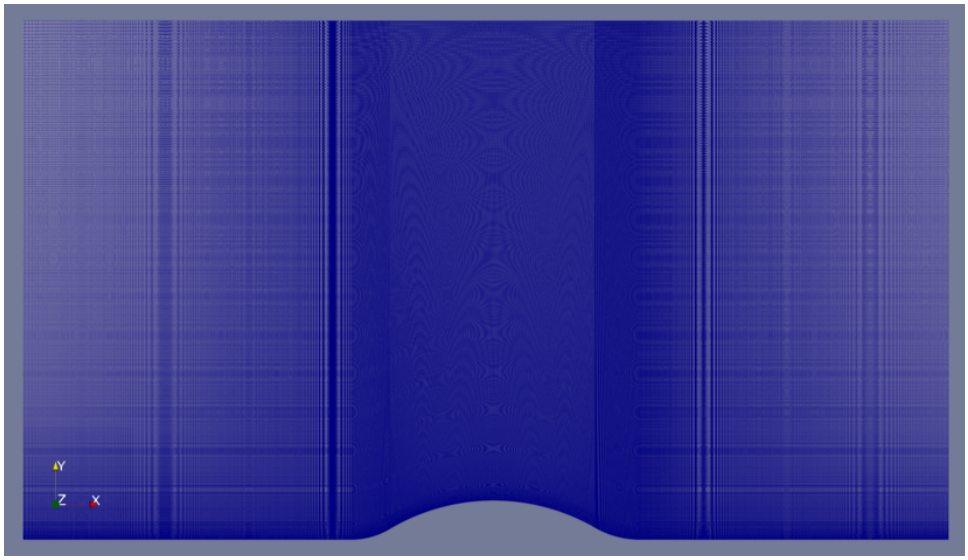


(a)

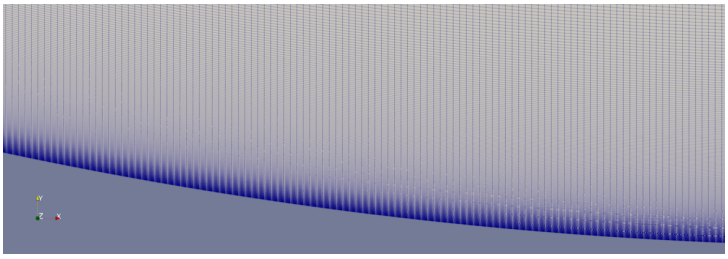


(b)

Figure 2.9: Schematic of the fine mesh configuration in the flat plate (turbulence precursor): (**a**) Full view; (**b**) Near wall region close-up.



(a)



(b)

Figure 2.10: Schematic of the fine mesh configuration in the curved hill: (**a**) Full view; (**b**) Near wall region close-up at the second concave surface.

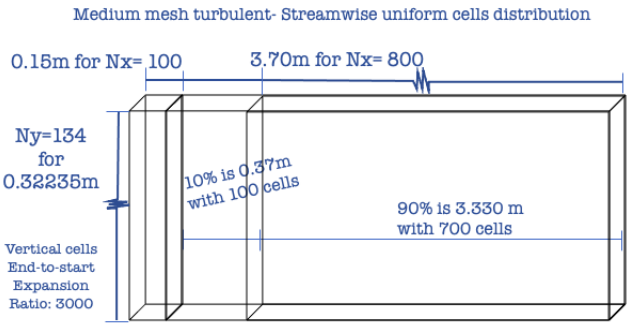


Figure 2.11: Example of dimensions and cell distribution for flat plate medium mesh.

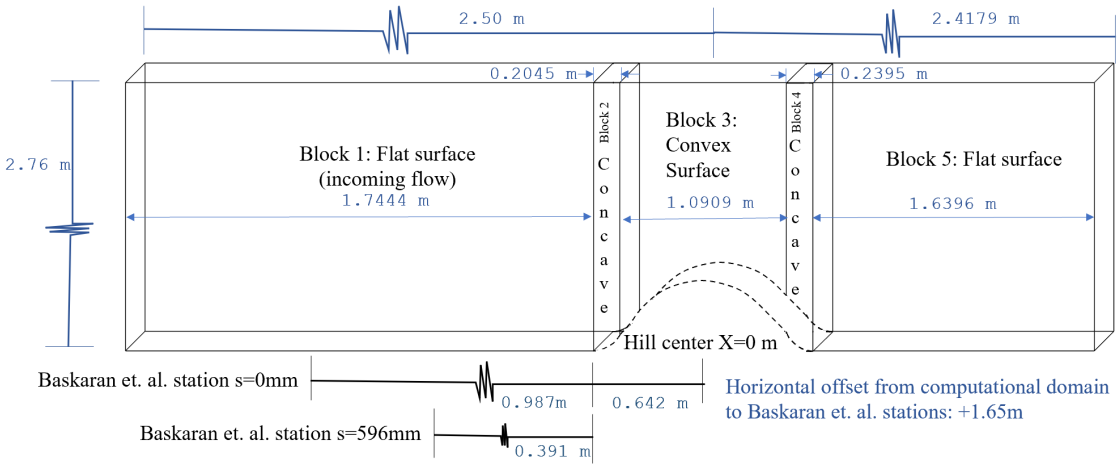
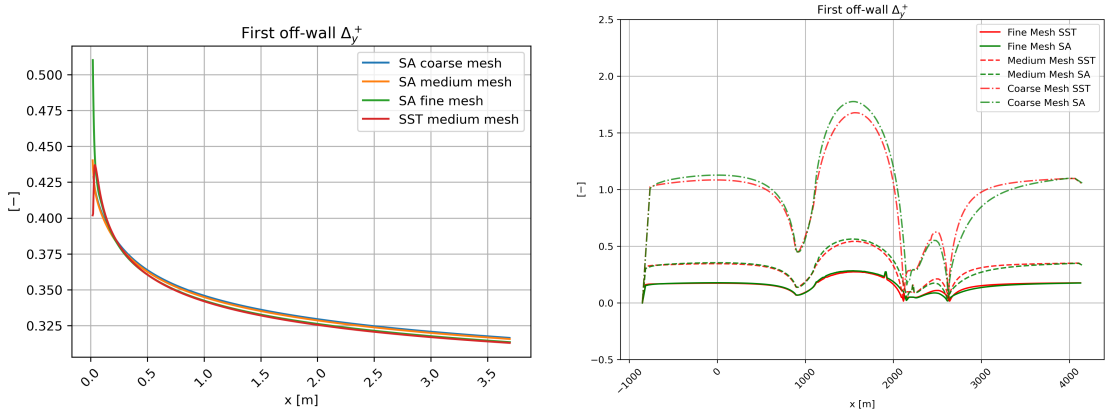


Figure 2.12: Example of dimensions of curved hill computational domain (not at scale) (Baskaran et al. 1987). From Paeres et al. (2022b); reprinted by permission of MDPI from the journal Energies.

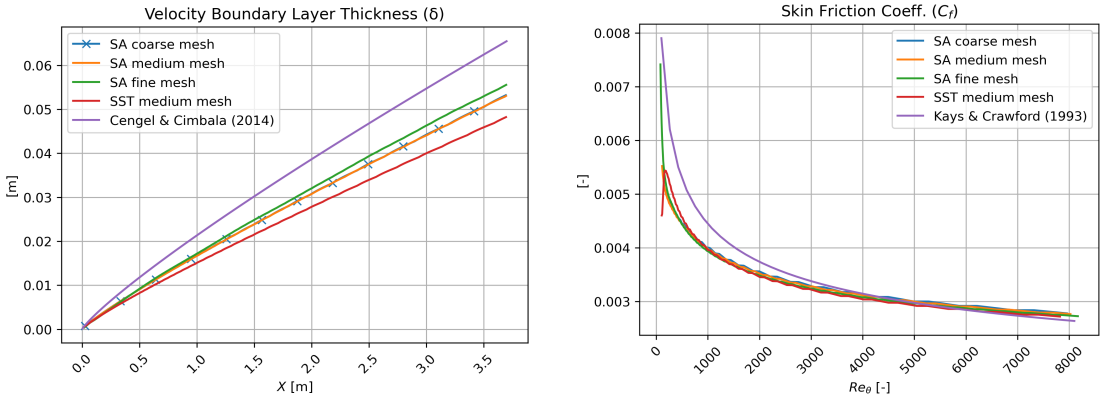


(a) (b)

Figure 2.13: Comparison of the dimensionless near-wall mesh resolution in: (**a**) The flat plate domain; (**b**) The curved hill domain.

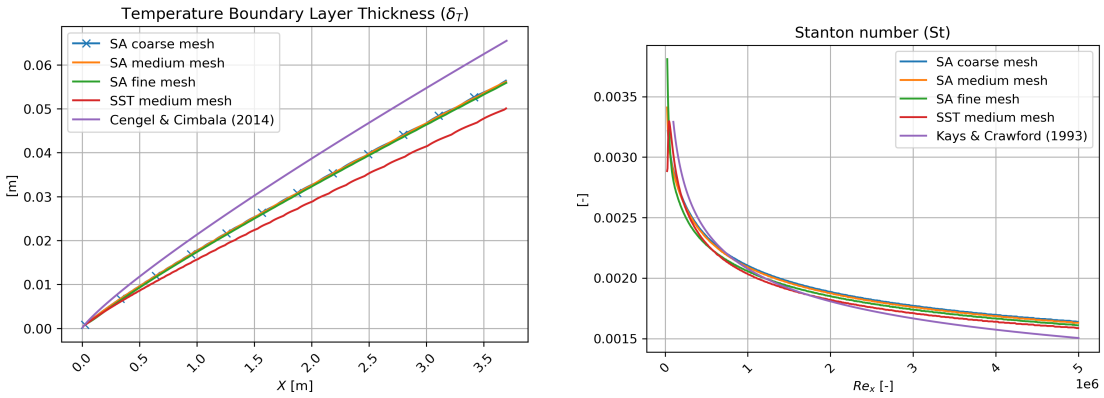
The flat plate simulations have shown excellent agreement with empirical correlations. The velocity and thermal profiles were used to inject a more realistic (and developed) boundary layer flow into the curved-hill domain inlet. Because SA cases were executed first, it was found that the results of coarse to fine mesh were sufficiently resolution independent; therefore, SST was only performed with coarse and medium resolutions. Figure 2.14(a) displays the boundary layer thickness and comparison with the 1/7-power

law given in Cengel and Cimbala (2014). Figure 2.14(b) presents the skin friction coefficient and the theoretical formulation proposed by Kays and Crawford (1993).



(a) (b)

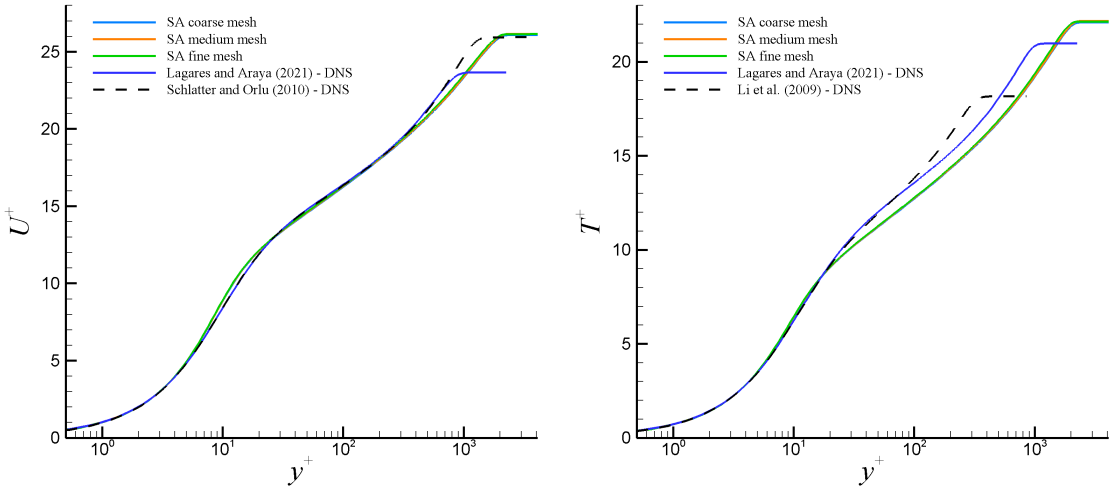
Figure 2.14: Flat plate solutions of: (**a**) Boundary layer thickness; (**b**) Skin friction coeff.



(a) (b)

Figure 2.15: Flat plate passive scalar solutions of: (**a**) temperature boundary layer thickness, and (**b**) Stanton number.

Regarding the passive scalar, the temperature boundary layer thickness is shown in Figure 2.15(a); as it can be seen, the results are just differentiated by the turbulent model used. Furthermore, to validate the temperature fields, the Stanton number was used in Figure 2.15(b) by comparison with the theoretical formulation given in Kays and Crawford (1993).



(a) (b)

Figure 2.16: Quality assessment of turbulent inflow profiles of: (**a**) Mean streamwise velocity, U^+; (**b**) Mean temperature, T^+, in wall units.

Figure 2.16 shows the extracted streamwise velocity and thermal profiles from the flat plate simulations for the three meshes and based on the SA turbulence model at a location ($X = 2.458\ m$) where the local momentum-thickness Reynolds number, Re_θ, is approximately 5659.7. Furthermore, in Figure 2.16(a), the mean streamwise velocity is depicted in wall units (i.e., $U^+ = U_s/u_\tau$ and $y^+ = yu_\tau/\nu$), where a consistent trend can be observed in all meshes. Here, $u_\tau = \sqrt{\tau_w/\rho}$ where τ_w is the wall shear stress and ρ is the fluid density. It is worth highlighting that y^+ and n^+ will be used interchangeably to denote wall-normal coordinates in this work. In addition, a very good agreement is seen with DNS data from Lagares and Araya (2021), Schlatter, P. and Orlu, R. (2010) at $Re_\theta = 2354.7$ and 4060, respectively. Some discrepancies can be detected around $y^+ \approx 10$ and in the wake region (i.e., beyond the log region). The latter could be attributed to some Reynolds number dependency. Figure 2.16(b) exhibits the mean temperature profile (or passive scalar) normalized in wall units, as well. Here, $T^+ = \Theta/\theta_\tau$, where $\Theta = (T - T_w)/(T_\infty - T_w)$ T being the dimensional static temperature, T_w is the wall temperature, and T_∞ is the freestream temperature. The friction temperature is defined as $\theta_\tau = q_w/(\rho c_p u_\tau)$, where q_w is the wall heat flux and c_p is the fluid specific heat. Moreover, an excellent collapse of the RANS results via the SA turbulence model is visualized in the linear viscous layer and buffer region (i.e., for $y^+ < 20$) as well as with DNS data from Lagares and Araya (2021), Li et al. (2009) at $Re_\theta = 2354.7$ and 800, respectively. In the log and wake region, significant differences can be observed between the RANS and DNS approaches. One can infer that the turbulent Prandtl number model employed in RANS simulations has performed quite well in the near wall region, whereas it has been able to

reproduce moderately well the thermal field in the outer part of the turbulent boundary layer. Despite the fact that all meshes have generated similar outcomes, the solutions of SA medium mesh were selected for inflow in the curved hill cases (i.e., streamwise and wall-normal components of the mean velocity and temperature). Readers are referred to Appendix C for a comprehensive grid sensitivity study.

2.4 Results and Discussion for The Curved Hill

Turning into the curved hill scenario, let us begin by setting our bases by contrasting our numerical predictions with experimental data from the paper "A turbulent flow over a curved hill Part 1" (Baskaran, Smits, and Joubert 1987). Figures 2.17–2.19 show contour results of kinematic pressure gauge (P/ρ), horizontal velocity (U_x) and static temperature (T), respectively; via the SA turbulence model and fine mesh. Zero values of the kinematic pressure gauge were assigned to the outflow plane (reference or atmospheric pressure). It is very important to note that in this work the surface streamline distance (s) was matched to the presented in Baskaran et al. (1987), also the wall-normal distance is represented with n. In the path of the shear-layer region (i.e., turbulent boundary layer), the following aspects can be mentioned. As the flow approaches the hill or obstacle, the pressure increases, the flow decelerates and the pressure gradient becomes strongly adverse. Zones with intense red color can be seen along the first concave region ($987\ mm < s < 1191.5\ mm$). The fluid velocity in the viscous and buffer layer decelerates, inducing a decrease in the skin friction coefficient, as will be shown later. However, no strong backflow or reverse flow is seen due to the moderate APG infringed. Whereas, the temperature generally shows a small gradient with just a small temperature drop in a very limited location at the peak of this pressure strong adverse gradient. In the geometry change from concave to convex, the pressure gradient switches to favorable (FPG), this is translated to streamwise velocity acceleration. This flow acceleration or FPG continues until the flow reaches the hill's top. Clearly, the flow decelerates downhill, with a significant recovery of the pressure coefficient (as seen in Figure 2.20(a)). The presence of this strong APG (the pressure increase equals the dynamic pressure, since the change in $C_P \approx -1$) ends up in flow separation, with a posterior flow reattachment due to the presence of zero-pressure gradient (ZPG) zone, again. Temperature shows no change during this section with the exception of the very near wall. At the start of this separation zone, the temperature begins to drop. At the end of the hill, the separation "bubble" is noticeable due to the presence of a quasi-isothermal zone (in blue). The high level of mixing inside the separation bubble balances the static temperature. This is consistent with observations in the thermal boundary layer downstream of crossflow jet

problems via DNS (Quinones 2020).

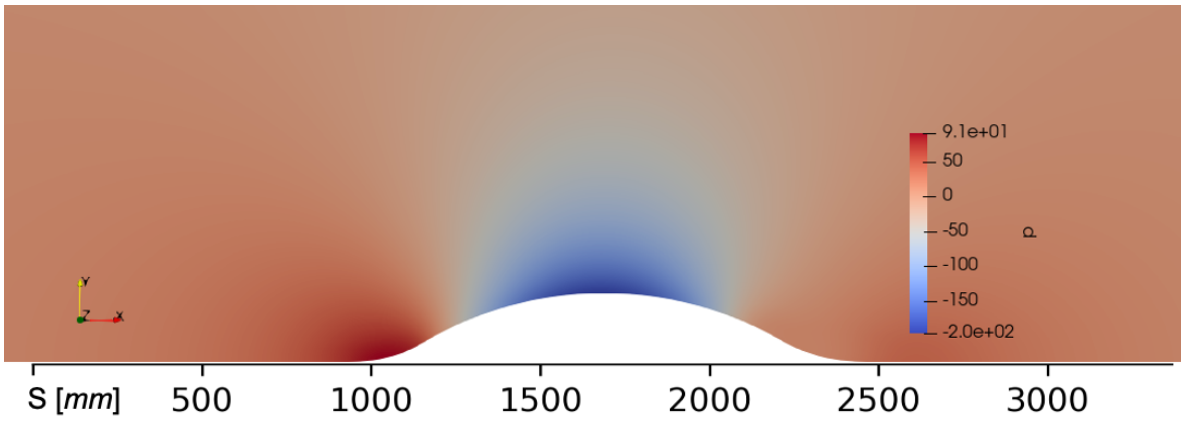


Figure 2.17: Contours of kinematic pressure gauge in m^2/s^2. The image has been zoomed in to highlight the curved hill and immediate surroundings.

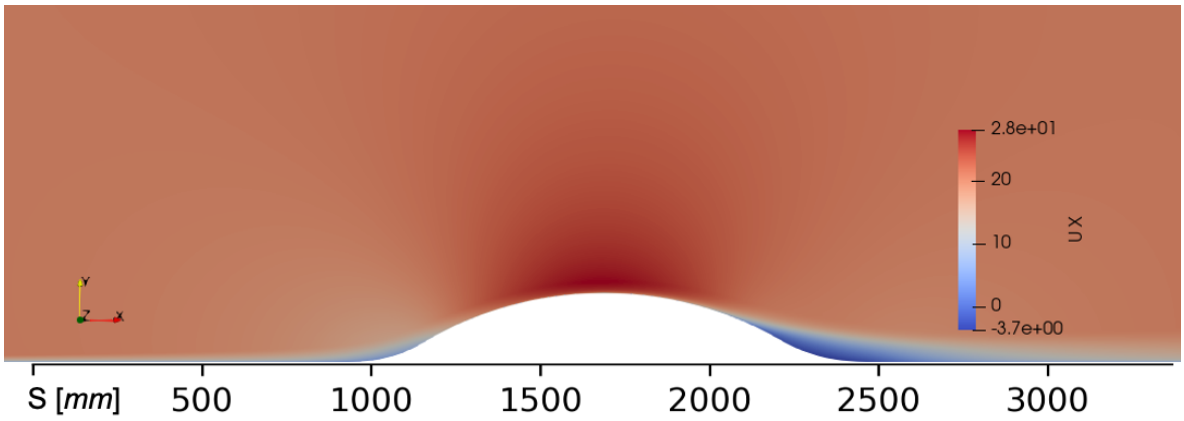


Figure 2.18: Contours of horizontal velocity in m/s. The image has been zoomed in to highlight the curved hill and immediate surroundings.

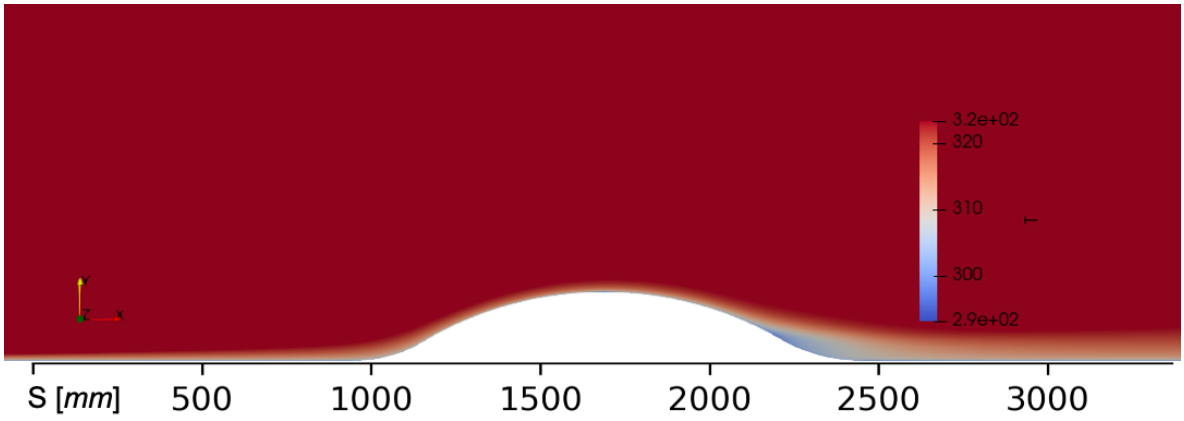
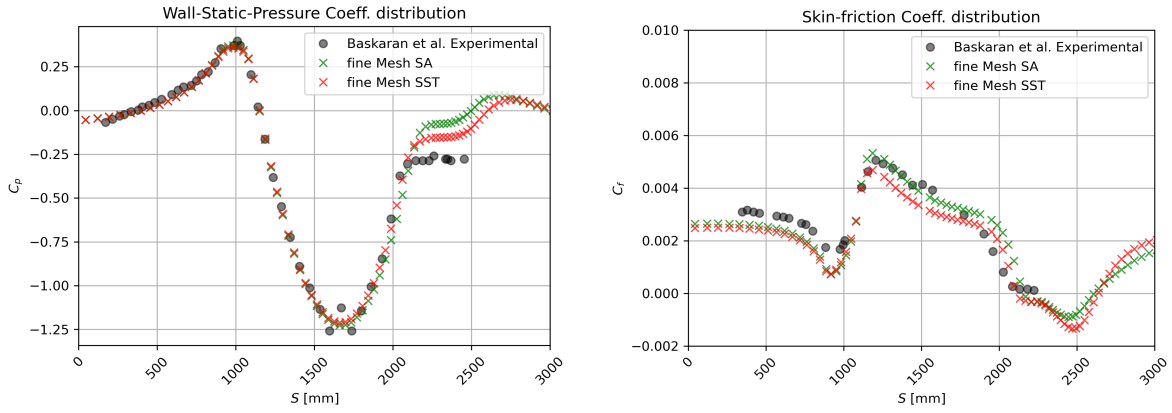


Figure 2.19: Contours of the static temperature in K. The image has been zoomed in to highlight the curved hill and immediate surroundings.

Figure 2.20(a) depicts the pressure coefficient along the computational domain. The pressure coefficient is defined as $C_P = (P_w - P_\infty)/q_\infty$. Here, P_w is the wall static pressure, P_∞ is the freestream static or reference pressure, and q_∞ is the freestream dynamic pressure. In general, a good agreement is observed with experimental data by Baskaran et al. (1987). As the flow approaches the obstacle or hill, it decelerates due to the presence of an increasing pressure or APG. The maximum C_P ($\approx$0.375) is located at the hill feet. Interestingly, good performance of both turbulence models in reproducing the wall pressure coefficient was observed in the vicinity of the hilltop, where the streamwise pressure gradient abruptly switches from FPG to APG, passing through a very short ZPG-zone. It is expected that boundary layer flow experiences a severe distortion in that zone with combined pressure gradients. Major discrepancies occur by the end of the strong APG zone (second half of the hill) where back or recirculating flow can be found. This is consistent with the deficient performance of RANS-eddy viscosity models in capturing boundary layer detachment. While both turbulence models have predicted constant wall static pressures in the separation bubble (ZPG zone), which is physically sound; however, smaller pressure gauges were obtained by SA and SST, e.g., $C_P \approx -0.0625$ and -0.125, respectively, as compared to the measured valued of -0.25. In Figure 2.20(b), the skin friction coefficient, C_f, is depicted. The skin friction coefficient is defined as follows, $C_f = \tau_w/q_\infty$, where τ_w is the wall shear stress. One can observe an opposite trend of C_f as compared with the pressure coefficient C_P. As the flow decelerates due to the presence of moderate APG nearby the hill feet (concave surface), it is seen a decreasing behavior of C_f just downstream of the ZPG region where almost constant skin friction coefficient values are seen, as expected. However, it never reaches negative values, indicating that the mean flow does not separate. The wall shear stress

then recovers as the flow starts to accelerate in the FPG region (convex surface). At roughly one-quarter of the curved hill (where the surface changes geometry from concave to convex) a meaningful increase of the wall shear stress and C_f is observed since the flow strongly accelerates (FPG), and approximately a 100% increase can be seen with respect to the incoming C_f under ZPG-flow conditions. Downhill, the pressure coefficient C_P recovers (presence of APG), inducing a reduction in C_f, to finally reach slightly negative values in the separation bubble ($s \approx 2100\ mm$). Obviously, the boundary layer flow should "pass-through" the laminar skin friction coefficient value before separating (i.e., $C_f < 0$). This may indicate the presence of quasi-laminarized flow within $2000\ mm < s < 2100\ mm$ and in the near wall region (Narasimha 1983) (extension of the viscous sub-layer). This supposition would be better addressed when discussing mean streamwise velocity and Reynolds shear stress profiles in the next pages. In summary, the SA and SST turbulence models have estimated similar and consistent values for C_f regarding the experimental data from Baskaran et al. (1987), perhaps the SA model has shown moderate supremacy, overall. The skin friction coefficient in the incoming ZPG zone is slightly under-predicted by both turbulence models (∼15% lower than in Baskaran et al. (1987)). As previously mentioned, the most "challenging" situation for turbulence models has undoubtedly been the hilltop and vicinity since the flow goes through acceleration and deceleration in a very short distance. Major differences were computed as roughly 35% in that zone. According to Baskaran et al. (1987), the location of the boundary layer detachment point was found to be situated at $s = 2095\ mm$ by extrapolation. The SA and SST models have predicted a separation point around $s \approx 2100\ mm$, in very close agreement with experiments.



(a) (b)

Figure 2.20: Coefficients on wall compared to experimental data from Baskaran et al. (1987): (**a**) Pressure coefficient; (**b**) Skin friction coefficient.

The streamwise variation of the Stanton number and the Reynolds analogy ratio (i.e., $S_t/(C_f/2)$) are shown in Figure 2.21. The Stanton number is defined as follows,

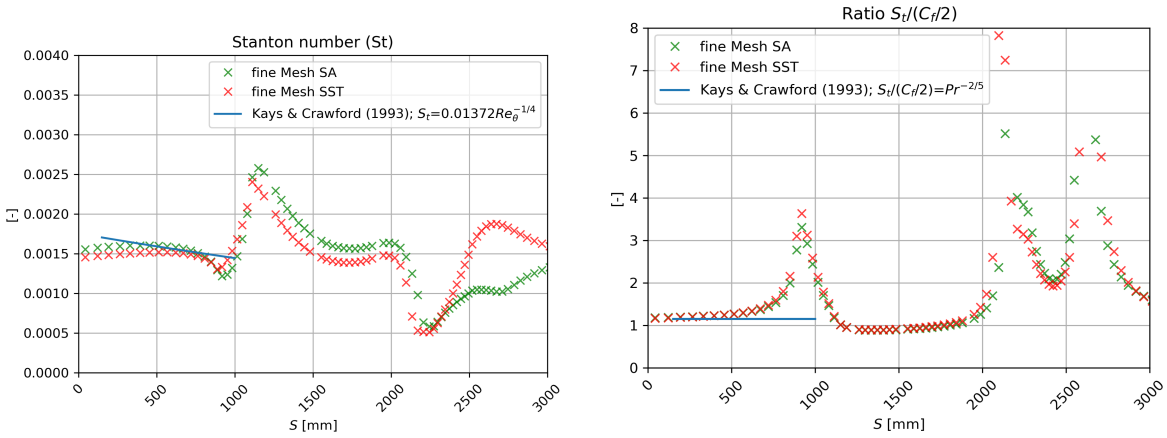
$$S_t = \frac{q_w}{\rho c_p U_\infty (T_\infty - T_w)} \tag{2.4}$$

where q_w is the wall heat flux defined as:

$$q_w = -\left(k\frac{\partial T}{\partial n}\right)_w \tag{2.5}$$

Here, k is the fluid thermal conductivity, c_p is the fluid's specific heat at constant pressure, and $\partial T/\partial n$ is the thermal gradient at the wall in the wall-normal direction. The Stanton number is a dimensionless number that relates the heat interchanged between the surface and the fluid to the thermal capacity of the fluid. From Figure 2.21(a), one can observe nearly constant S_t in the incoming flow (ZPG), which is typical in canonical or flat-plate boundary layers. A very good agreement was obtained with the empirical correlation by Kays and Crawford (1993), who proposed a variation of S_t as a function of the local momentum thickness Reynolds number, Re_θ, for ZPG turbulent flows (adapted to $Pr = 0.71$). The Stanton number (and heat transferred) peaks by $s \approx 1125\ mm$, where $(C_f)_{max}$ is located, demonstrating a high similarity between maximum viscous shear force and total heat transferred at the wall. This peak in the heat transfer (representing about an 80% increase regarding the incoming baseline S_t) is situated approximately by the end of the first concave bend. Downstream, the Stanton number decreases much faster than C_f does, suggesting a non-similarity between these two boundary layer parameters. The strong changes of streamwise pressure gradients in this zone, which are sources of dissimilarity between the momentum and thermal boundary layer transport (G. Araya and L. Castillo 2013), are the reasons for that behavior. Beyond $s = 1500\ mm$, a "plateau" is observed in S_t values, and an abrupt reduction of the heat transfer is achieved by $s \approx 2100\ mm$, caused by the presence of the flow recirculation zone. This separation bubble is characterized by a quasi-adiabatic process, since no heat transfer occurs between the surface and the fluid ($S_t \approx 0$). Furthermore, the Stanton number can also be related to the skin friction coefficient via the Reynolds analogy (similarity between the viscous drag to the heat interchanged). The $S_t/(C_f/2)$ ratio is introduced in Figure 2.21(b). An excellent agreement with the $Pr^{-2/5}$ empirical correlation by Kays and Crawford (1993) is seen in the ZPG zone. This ratio significantly departs from the unitary value by the hill feet (beginning of the concave bend, $s \approx 987$ mm). It is worth highlighting that $S_t/(C_f/2)$ remains very close to one in most of the curved hill for $s > 1100\ mm$, getting large negative values in the vicinity of the separation bubble due to the very small (and

negative) values of C_f. In essence, both turbulence models have generated very similar Stanton numbers.



(a) (b)

Figure 2.21: Streamwise variation of: (**a**) Stanton number; (**b**) The $S_t/(C_f/2)$ ratio.

To the best of our knowledge, the implemented approach, based on a potential flow-based scheme, has shown robustness and accuracy in the detection of boundary layer edge parameters as well as its integral values in comparison to the classical $99\%U_\infty$ criterion. This made the overall calculation of the parameters shown in Figures 2.23 and 2.22 consistent in the presence of strong pressure gradients, and subsequently, severe boundary layer distortion. The boundary layer thickness is slightly underestimated, although it follows the experimental results' behavior. Baskaran et al. (1987) have limited data in the separation bubble, thus, let us focus on the sections highlighted by them. The overestimated shape factor is likely due to an overestimation in the boundary layer's edge velocity incurred when comparing the potential and RANS flow fields. Nonetheless, the comparisons along the hill are extremely favorable to our approach compared to the experimental baseline. Overall, the SA model has superior performance when compared to the $K-\omega$ SST (Menter 1994). This is particularly noticeable within the portion $800\ mm < s < 1200\ mm$ seen in the integral parameters prediction. Both turbulence models significantly over-predict the maximum shape factor, H, located at $s \approx 1000$. At this point, the meaningful thickening of the turbulent boundary layer is consistent with the presence of strong APG and flow deceleration (note the C_P peak in Figure 2.20(a)). Consequently, the shape factor increases (up to $\sim$15% increases with respect to the incoming flow), and discrepancies in numerical results are within 25% regarding experimental values. Since the shape factor, H, is the ratio of the displacement thickness to

the momentum thickness, the previously mentioned discrepancies on H_{max} are caused by over-predictions on the displacement thickness, δ^*, as confirmed from Figure 2.22. In addition, the momentum thickness, θ, has been almost faithfully replicated by RANS as compared to experiments. This is consistent with the good agreement on the C_f variation (see Figure 2.20(b)) since the momentum thickness is proportional to the drag force over the surface. To understand the physical significance of δ^* and θ, those integral boundary layer parameters are proportional to the deficit of mass flow and momentum inside the boundary layer due to the viscous effect and no slip wall condition, but how it would be if the flow were inviscid. δ^* represents how much the boundary layer should be displaced outward to compensate for the mas flow reduction due to the presence of a solid wall, while θ is representative of the momentum loss; thus, proportional to the drag. For the creation of Figures 2.22 and 2.23, the potential flow-based scheme used Equations (2.6) and (2.7) to calculate displacement and momentum thickness parameters. Meanwhile, the parameter Re_θ used a reference velocity, U_{ref}, where after following the results of Baskaran et al. (1987), it was found to be $s = 596\ mm$, i.e., the reference station.

$$\int_0^\delta \left(1 - \frac{U_s}{U_{Potential}}\right) dn \tag{2.6}$$

$$\int_0^\delta \frac{U_s}{U_{Potential}} \left(1 - \frac{U_s}{U_{Potential}}\right) dn \tag{2.7}$$

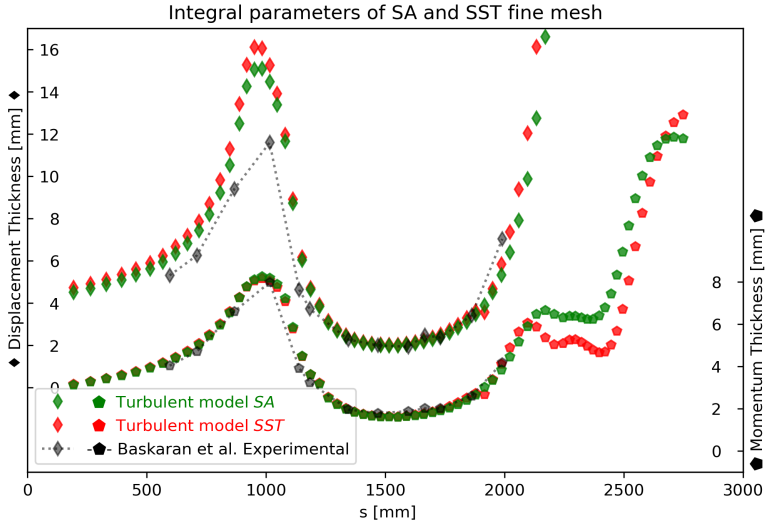
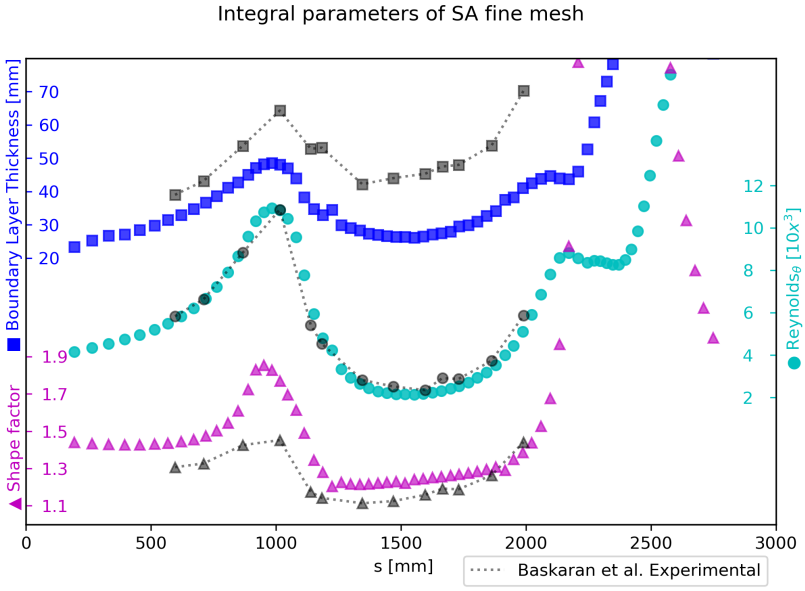
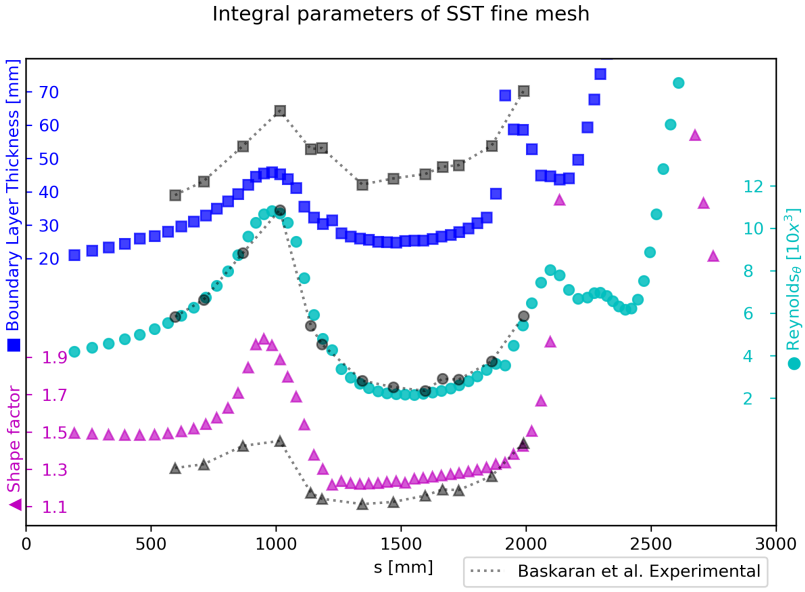


Figure 2.22: Displacement thickness and momentum thickness compared to experimental data from Baskaran et al. (1987).



(a)



(b)

Figure 2.23: Boundary layer thickness, shape factor, and momentum thickness Reynolds number compared to experimental data from Baskaran et al. (1987): (**a**) Spalart-Allmaras model; (**b**) $K-\omega$ SST model.

To assess the distortion of the momentum boundary layer due to the hill, some wall-normal inner-scaled profiles are presented at several streamwise stations in Figures 2.24 and 2.25. Comparison is performed with experiments by Baskaran et al. (1987) as well as against two DNS baselines by Schlatter, P. and Orlu, R. (2010) ($Re_\theta = 4060$) and Lagares and Araya (2021) ($Re_\theta = 2305$). For the ZPG zone in Figure 2.24(a), the inner scaled velocity profiles of both closure models collapse near perfectly with a very slight variation observed in the wake region where the $K-\omega$ SST variant predicts a slightly higher streamwise velocity, U_s. Both DNS databases exhibit a high level of consistency (almost overlap), and a long log region can be seen due to the high Reynolds numbers considered. The flow starts to decelerate by the first concave bend beginning ($s \approx 710\ mm$), which causes a decrease in the skin friction coefficient and in the friction velocity as well. This is the reason for the slight upward movement of the wake in Figure 2.24(b), which is over-predicted by the turbulence models. The effects of flow acceleration and FPG can be observed in Figures 2.24(c) and 2.24(d), inducing a "hump" in velocity profiles over the log region, also reported by Araya, Castillo, and Hussain (2015) in sink flows. The boundary layer suffers a significant distortion around the hilltop where a "plateau" in U_s can be observed between $n^+ = 300$ and 1000 from experimental data. In this FPG zone, the SA model predictions are in better agreement with experiments by Baskaran, Smits, and Joubert (1987). As discussed in Figure 2.20(a), the pressure coefficient recovers from $s \approx 1700\ mm$ and the streamwise pressure gradient becomes more adverse. Therefore, one can describe the APG influence as steeper slopes of the velocity profiles within the log-wake region. Finally, the sudden deceleration (strong APG) of the boundary layer produces a recirculation bubble above the wall, i.e., between 2 to 1000 wall units, as can be seen in Figure 2.25, where streamwise velocity profiles can be seen at three different stations. While the SA model predicts a slightly stronger reversal flow, the SST model predicts a larger (in terms of wall-normal distance) bubble. This recirculation zone is characterized by a low level of velocity fluctuations (this will be confirmed later in the Reynolds shear stress profiles) and back (negative) flow, where almost constant values of U_s can be observed (within 5 to 10 in wall units, according to the streamwise station). Beyond the separation zone in the wall-normal direction, the streamwise velocity exhibits very sharp increases towards the boundary layer edge, resembling Blasius velocity profiles. The substantial discrepancies observed in both turbulence models confirm the previous statement regarding the limitations of eddy viscosity models to accurately predict boundary layer detachment.

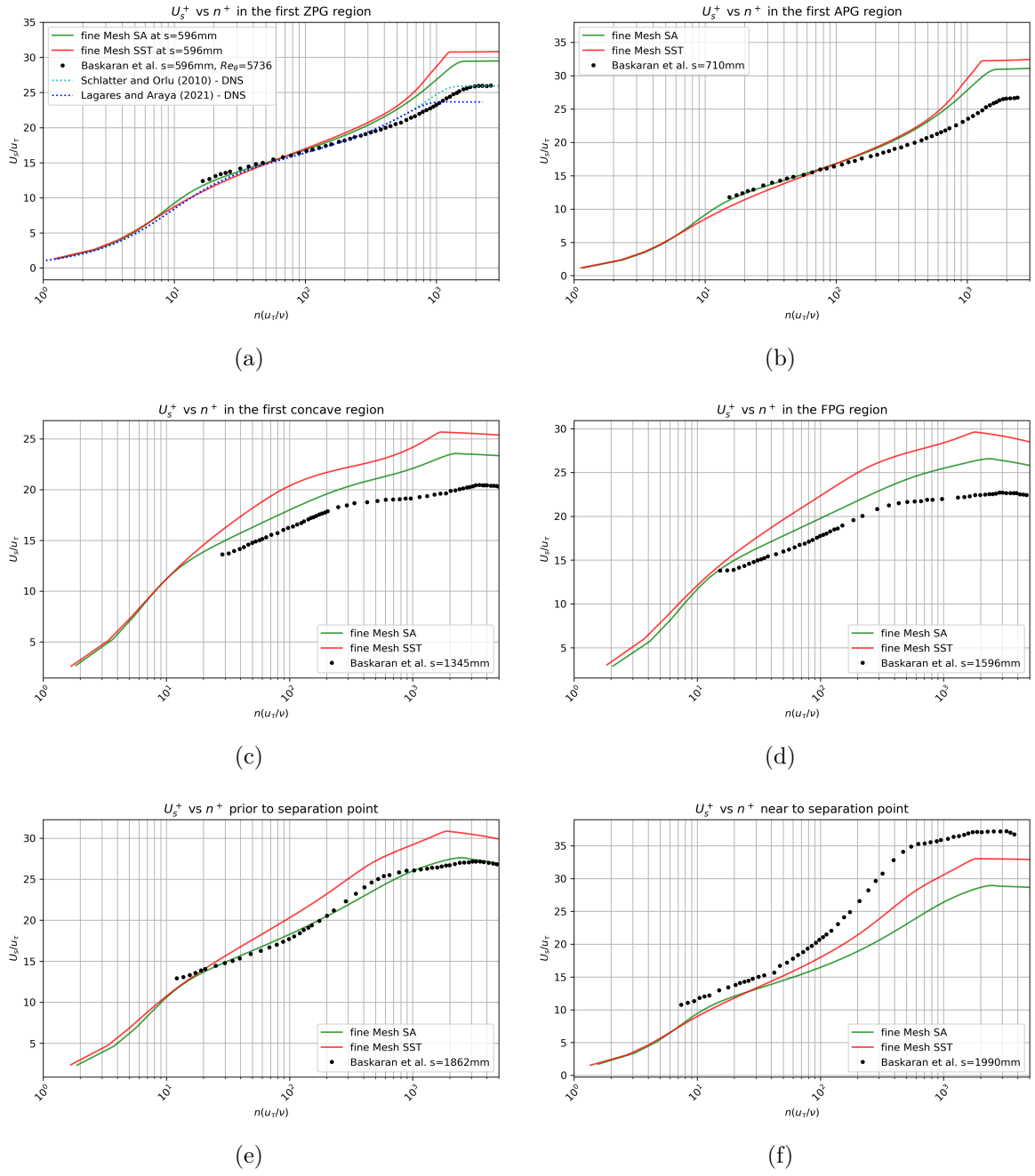


Figure 2.24: Streamwise velocity profiles in wall units at locations of: (**a**) s = 596 mm; (**b**) s = 710 mm; (**c**) s = 1345 mm; (**d**) s = 1596 mm; (**e**) s = 1862 mm; (**f**) s = 1990 mm.

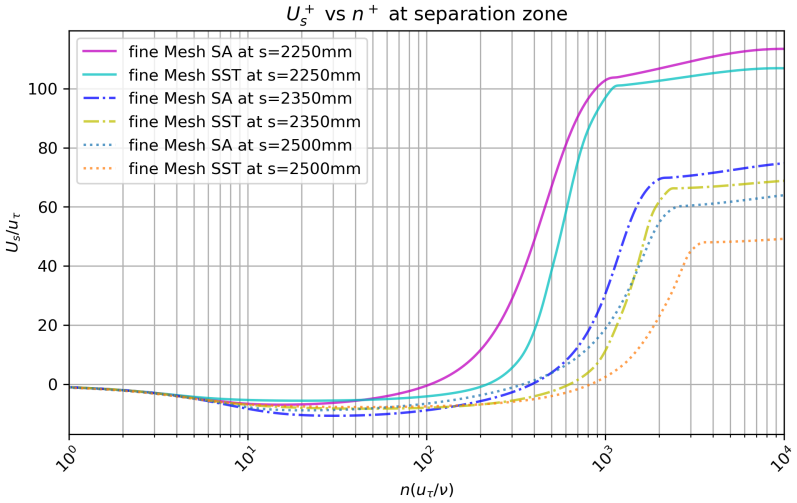


Figure 2.25: Velocity profiles in the flow separation bubble at $s = 2250\ mm$, $2350\ mm$, and $2500\ mm$.

The Boussinesq hypothesis was applied to estimate Reynolds shear stresses from the RANS output as per the equation:

$$< u'v' > \approx -\nu_T \left(\frac{\partial U_s}{\partial n} + \frac{\partial U_n}{\partial s} \right) \tag{2.8}$$

In Figure 2.26, profiles of Reynolds shear stresses are plotted in inner units and several streamwise stations by considering the friction velocity at the reference station ($s = 596$ mm or ZPG zone). This choice is based on the isolated assessment of the baseline (incoming) Reynolds shear stresses under combined streamwise pressure gradients caused by the curved hill. Moreover, we remove any scaling effect according to the local values of the friction velocity. At the ZPG station ($s = 596\ mm$), it can be seen that both turbulence models tend to capture the inner portion of the boundary layer with very good agreement with DNS from Lagares and Araya (2021) and Schlatter, P. and Orlu, R. (2010). The comparison breaks down in the outer region where both models predict larger values, perhaps, caused by the higher Reynolds numbers considered in RANS predictions. The APG effect at $s = 1139\ mm$ is manifested as a clear secondary peak on $< u'v' >$ around $n^+ \approx 800$. It can be infer that the flow is subject to a very strong deceleration or APG since the outer peak is larger (almost twice as large) than the inner peak, around $n^+ \approx 15$. The SST model predicts a more intense outer peak, addressing one of the original research questions of this study. These outer secondary peaks of $< u'v' >$ have also been reported by G. Araya and L. Castillo (2013) in DNS of turbulent spatially

developing boundary layers subject to strong streamwise APG. Moreover, outer streaks are enhanced by APG, which in turn cause local increases of streamwise velocity fluctuations and Reynolds shear stresses, according to DNS studies by Skote, Henningson, and Henkes (1998). Interestingly, a much stronger APG effect can be seen at $s = 1990\ mm$, just upstream of the separation bubble, given by the inclined shear layer or "plateau" in the zone $10 < n^+ < 200$. The larger the APG, the more inclined the shear layer (G. Araya and L. Castillo 2013). At the flow recirculation zone, i.e., at $s = 2500\ mm$, there is an appreciable attenuation of the Reynolds shear stresses in the near wall and buffer region ($n^+ < 100$).

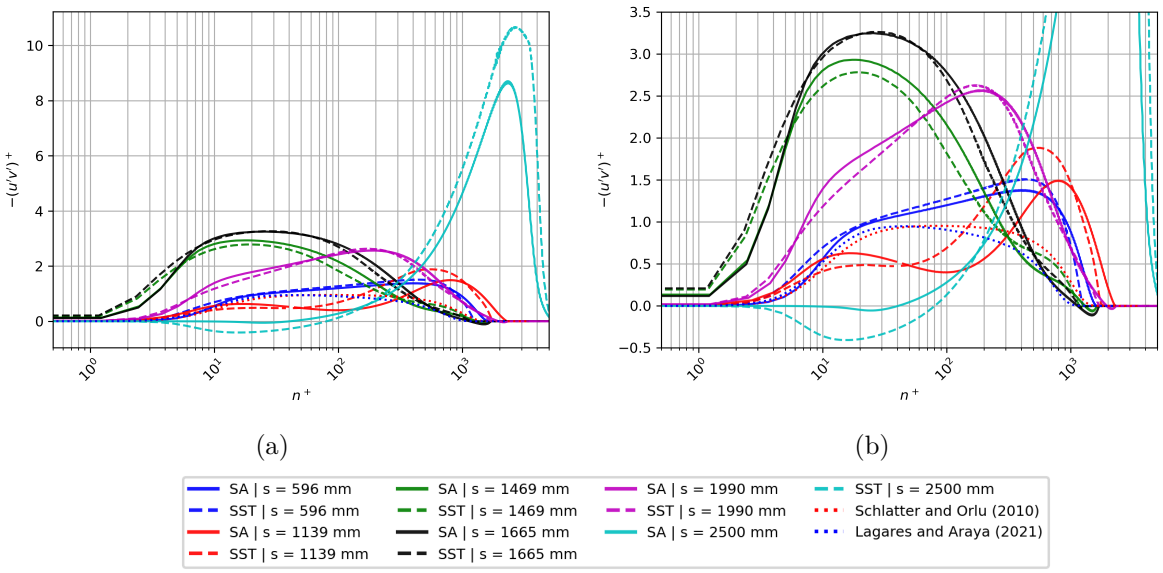


Figure 2.26: (**a**) Inner-scaled Reynolds shear stresses; (**b**) Zoomed view of the inner-scaled Reynolds shear stresses.

Furthermore, the SST model predicts large positive values of the cross-correlation $< u'v' >$ inside the bubble (not seen in the SA model), which is consistent with previous DNS studies in flow separation (Skote, Henningson, and Henkes 1998; Quinones 2020). Clearly, the very low values of the Reynolds shear stresses suggest that the flow is quasi-laminarized or on the verge of relaminarization (Narasimha 1979) (viscous sublayer extended). On the other hand, the very large values of $< u'v' >$ in the outer region ($1000 < n^+ < 3000$) plus the non-negligible wall-normal gradients of the streamwise velocity indicate the presence of significant turbulence production (i.e., $< u'v' > \partial U_s/\partial n$) well above the separation bubble, and thus, the flow is highly turbulent in that zone, as will be shown in the next figure. The published data by Baskaran et al. (1987) do not contain

much information beyond the early portions of the separation bubble.

$$\mathcal{P} \equiv -u_i' u_j' \frac{\partial U_i}{\partial x_j} \approx - < u'v' > \left(\frac{\partial U_s}{\partial n} + \frac{\partial U_n}{\partial s} \right) \tag{2.9}$$

The turbulent kinetic energy (K) production, $\mathcal{P}$, inside the boundary layer can be evaluated by computing the term with the highest contribution to K and mean-flow kinetic energy equations. The time-averaged velocity gradients act against the Reynolds stresses, removing kinetic energy from the mean flow and transferring it to the fluctuating velocity field (Pope 2000). The turbulent kinetic energy is defined as in Equation (1.21); whereas the Boussinesq hypothesis leads to the following definition of $\mathcal{P}$ in Equation (2.9), where the Reynolds shear stress definition in Equation (2.8) is employed. Figure 2.27 shows the principal term of K production in wall units at different streamwise stations, as done with the RANS-modeled Reynolds shear stresses. It was observed that the term $\partial U_n/\partial s$ in the mean flow gradient was negligible, even in zones with large wall curvatures. For the incoming flow (ZPG zone) at $s = 596\ mm$, the SA turbulence model reproduces properly the turbulence production in the turbulent boundary layer, as contrasted to DNS from Lagares and Araya (2021) and Schlatter, P. and Orlu, R. (2010) at lower Reynolds numbers. Peak values are approximately 0.25 to 0.29 in the buffer layer at $n^+ \approx 10$. While an increase in the term $\mathcal{P}^+$ would suggest a mean flow deceleration; whereas, an enhancement of the fluctuating component of the velocity field and Reynolds shear stresses. Previous flow physics descriptions can be clearly seen at stations $s = 1469\ mm$ and $1665\ mm$. Down the hilltop, turbulence production begins to recover the inflow features (attenuation process) as the Reynolds shear stresses decrease in the viscous sublayer and buffer region. In addition, flow deceleration by APG tends to destabilize the boundary layer, inducing turbulence intensification in the outer portion. Based on DNS studies by Skote et al. (1998), the outer streaks are intensified by strong APG and can be related to local increases in turbulence production and $<u'v'>$ (outer peaks). As seen in Figure 2.27, SA and SST models predict outer peaks of turbulence production around $n^+ \approx$ 1500–2500 at $s = 2500\ mm$, where the separation bubble is thicker. Furthermore, the production of K inside the bubble is almost negligible, suggesting that the flow is locally quasi-laminar.

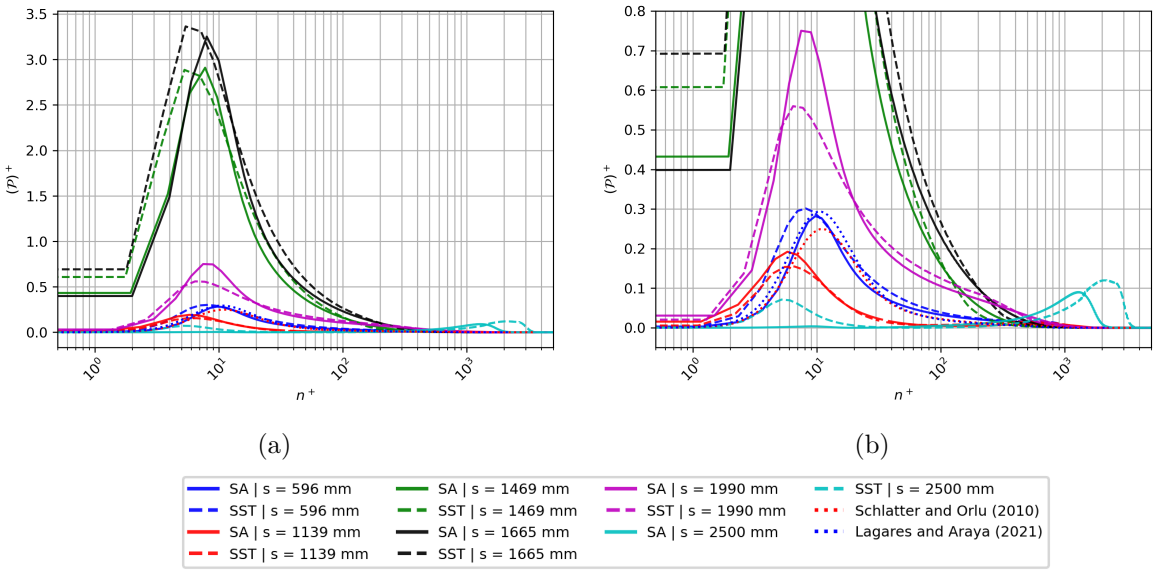


Figure 2.27: (**a**) Inner-scaled K Production; (**b**) Zoomed view of the inner-scaled K Production.

Observing the temperature as a passive scalar, from Figure 2.28, the separation bubble seems to elongate the thermal "plateau", although the distortion effects are less "violent" than those seen in the momentum boundary layer. As expected, the predictions at the separation bubble seem to disagree more than on any other station between turbulence models. Interestingly, we can now visualize two distinctive logarithmic regions of the thermal profile: one log slope inside the bubble and the other (steeper) outside the separation zone up to the thermal boundary layer edge. This log behavior may open up the opportunity for future better turbulence modeling of passive scalar transport in flow separation. Furthermore, a deeper analysis must be performed in this sense. In general, the thermal boundary layer profiles presented in Figure 2.29 follow similar tendencies (i.e., Reynolds analogy) to those presented for the momentum boundary layer. This is expected since the temperature is modeled as a passive scalar. Particularly, a high level of similarity has been observed in ZPG zones since streamwise pressure gradient is a source of dissimilarity between momentum and thermal fields. For instance, nearby the hilltop, strong FPG effects were described by the presence of "humps" in velocity profiles over the log region. However, thermal profiles look very different at $s = 1345\ mm$ and 1596 mm, indicating Reynolds analogy breakdown. Actually, a significant portion of the thermal boundary could be represented by a logarithmic curve fitting, given by the observed "linear" behavior when plotted on a semi-log scale.

It is worth highlighting that the 2D results presented in this work assume spanwise homogeneity. The main expected deviations when considering a full 3D domain and unsteady simulations can be summarized as follows: (i) The appearance of Görtler-like vortices due to the strong concave curvatures ($\delta/R \approx -0.13$ to -0.15, where δ is the local boundary layer thickness and R is the local radius of curvature) present in the complex geometry and (ii) Flow separation bubble. Our previous experience on supersonic turbulent boundary layers subject to strong concave surfaces via DNS (Lagares et al. 2019) dictated the existence of Görtler-like vortices caused by centrifugal forces, which in turn, enhanced spanwise flow fluctuations. However, no spanwise inhomogeneity has been observed in time-averaged flow statistics of turbulent flows. On the other hand, flow separation at the second concave surface should be ruled by unsteadiness and three-dimensionality effects. Even when the assumption of spanwise homogeneity may not be perfect, the results have shown reasonable agreement with wind-tunnel experiments.

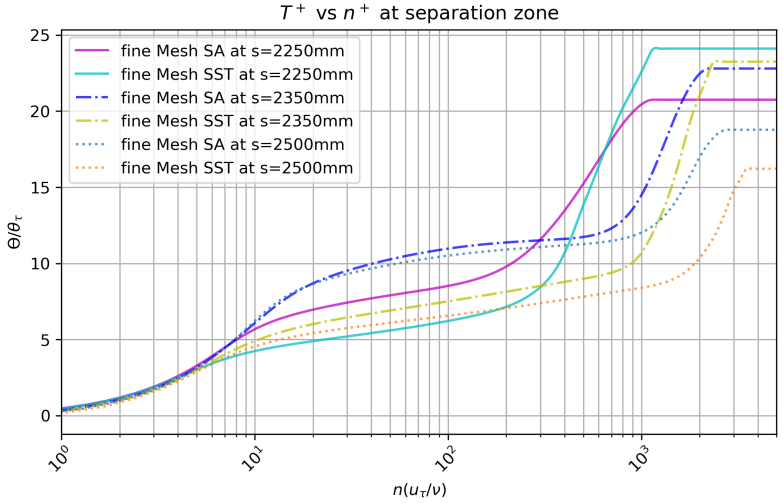


Figure 2.28: Thermal profiles in the flow separation bubble at $s = 2250\ mm$, $2350\ mm$, and $2500\ mm$.

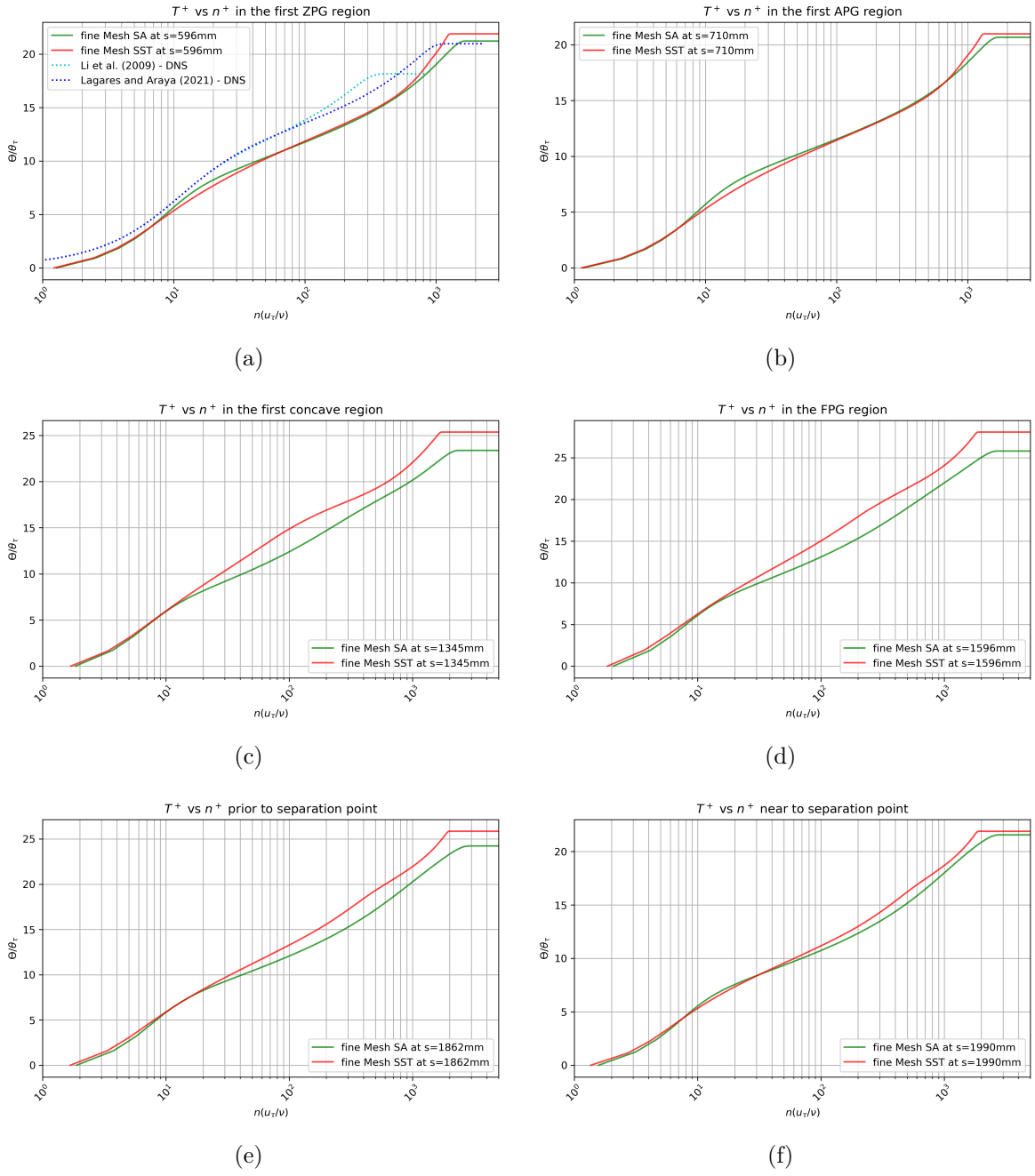
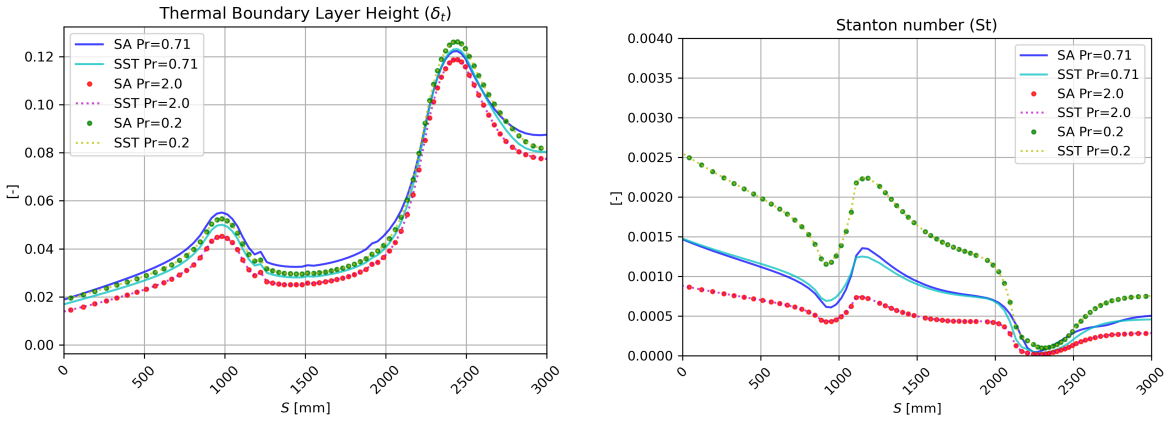


Figure 2.29: Thermal profiles in wall units at locations of: (**a**) s = 596 mm; (**b**) s = 710 mm; (**c**) s = 1345 mm; (**d**) s = 1596 mm; (**e**) s = 1862 mm; (**f**) s = 1990 mm.

2.4.1 Evaluation of Several Passive Scalars

In this section, the assessment of various molecular Prandtl numbers (e.g., $Pr = 0.2$, 0.71, and 2.00) is carried out in the transport phenomena of passive scalars. Accordingly, three turbulent Prandtl numbers, Pr_t, are prescribed based on empirical correlation proposed by (Li, Schlatter, Brandt, and Henningson 2009) as follows: $Pr_t = Pr^{0.4}$, which results in $Pr_t = 0.525$, 0.872, 1.319 for $Pr = 0.20$, 0.71, 2.00, respectively. The purpose of defining turbulent Prandtl numbers is to model the turbulent passive-scalar diffusivity and the turbulent passive-scalar flux in the transport equation. Figure 2.30(a) depicts the streamwise variation of the passive-scalar boundary layer thickness. Clearly, as the molecular Pr increases, the thermal boundary layer slightly shrinks. Similarly, the Stanton number (proportional to the wall heat flux and inversely proportional to Pr) is the smallest at the maximum Pr of 2, as seen in Fig. 2.30(b). As expected, the influence of the molecular Pr on the thermal boundary layer height in turbulent flow is not as critical as in laminar boundary layers, since wall-normal turbulent mixing plays a key role as compared to molecular diffusion. On the contrary, the effect of the Pr on the St is evident due to the implicit presence of the temperature gradient. Figure 2.31 shows profiles of the passive-scalar or temperature at six streamwise stations in the complex domain. In general, very similar results are obtained via the SA and SST turbulence model. Furthermore, profiles at the largest Prandtl number exhibit steeper slopes, whereas, profiles at $Pr = 0.2$ depict a gradual variation in the wall-normal direction. Moreover, in Figure 2.32, profiles of passive scalars are shown at three different sections of the separation bubble: start, middle, and end. Practically, all profiles collapse inside the bubble (up to $n^+ \approx 100$) no matter what the molecular Pr is. Outside the bubble, there is a tendency for profiles to be steeper as the Pr number increases.



(a) (b)

Figure 2.30: Streamwise variation of **(a)** the passive-scalar boundary layer thickness, and, **(b)** Stanton number.

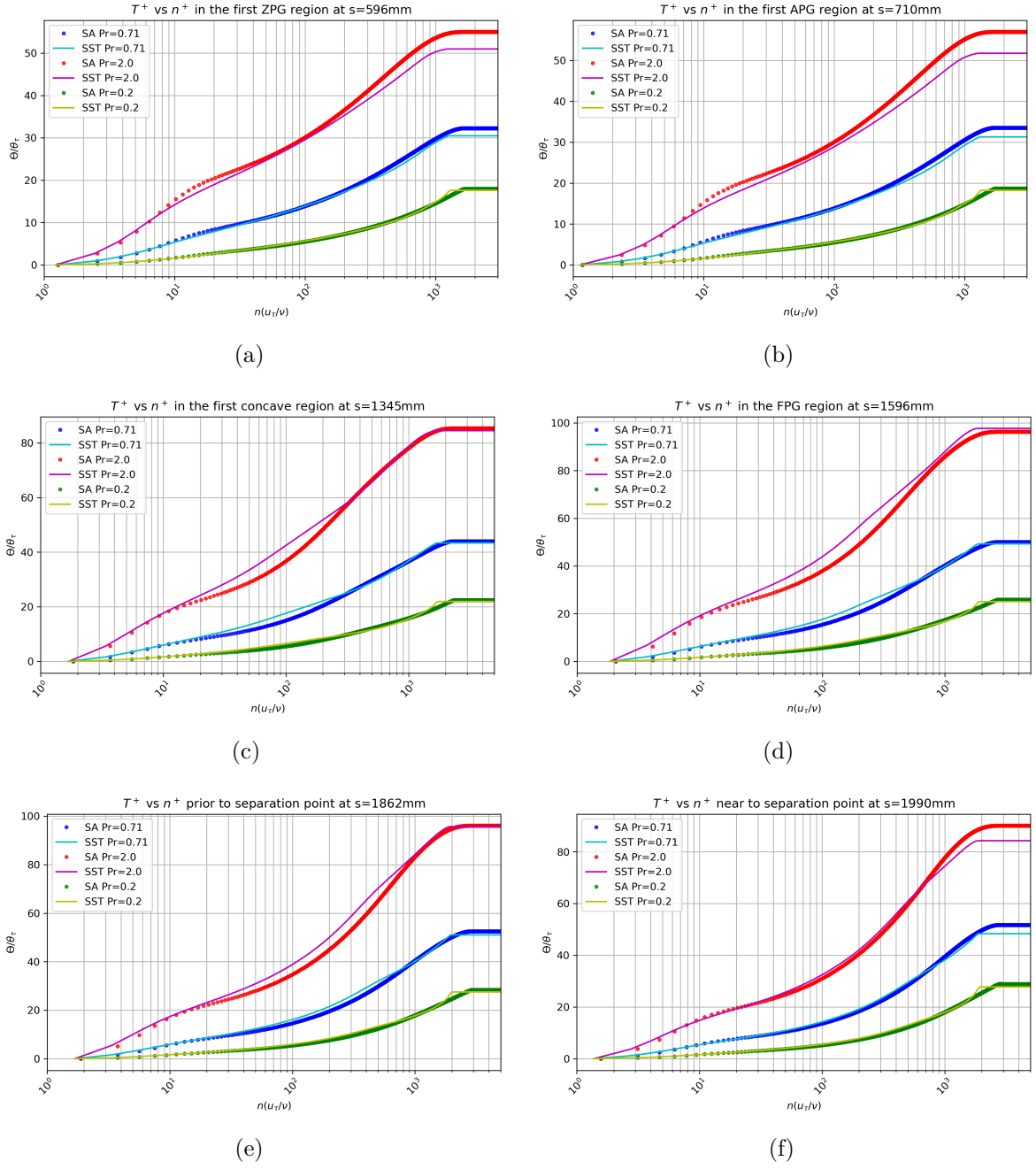
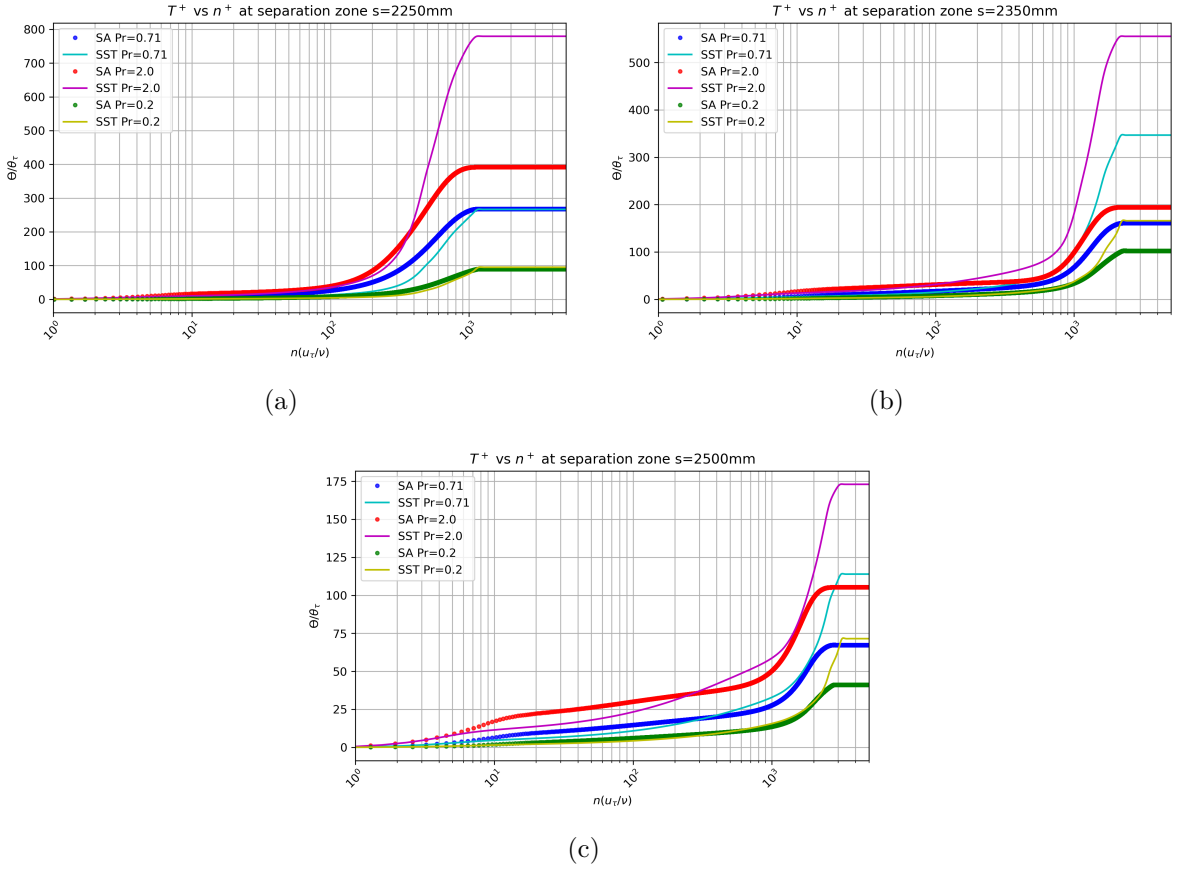


Figure 2.31: Passive-scalar profiles at several streamwise stations (attached flow zone).



(a) (b)

(c)

Figure 2.32: Passive-scalar profiles at several streamwise stations (recirculation flow zone).

2.5 Conclusions of the Curved Hill's Assessment

This chapter presented a numerical study of a turbulent boundary layer subject to a curved hill. The study was limited to a RANS simulation plus two eddy-viscosity turbulence models (i.e., SST and SA). The domain geometry was reproduced following work by Baskaran, Smits, and Joubert (1987). The inlet velocity components and temperature were recycled and injected from a precursor ZPG simulation to avoid a longer developing section and better control (and match with experiments) the incoming reference boundary layer parameters. By leveraging a boundary layer identification scheme based on a potential flow field, it was shown to be in very good agreement with the experimental data by Baskaran et al. (1987). This approach led to better identification of the turbulent boundary edge to compute the integral parameters and boundary layer thickness.

Furthermore, the proposed methodology has been resilient and robust in the presence of strong pressure gradients and significant boundary layer distortion.

Overall, the significant conclusions can be summarized as follows:

- The SA model had a better agreement with the experimental data in those zones where the turbulent boundary layer remained attached.
- The SST model, on the contrary, has depicted slightly superior agreement with experiments in the separation bubble (in terms of C_P and C_f) as well as in U_s profiles just upstream of the bubble, capturing more evidently the positively correlated characteristics of u' and v' or positive $< u'v' >$.
- Both models have detected outer peaks of $< u'v' >$, turbulence production, and the inclined shear layer caused by strong APG.
- The effect of the detachment is more notable on the velocity profile than on the thermal profile (with significant differences between turbulent model predictions).
- Strong streamline curvature-driven pressure gradients cause a noticeable Reynolds analogy breakdown.
- Prandtl number clearly affects more the temperature gradient implicit in the Stanton number than the height of the thermal boundary layer.
- However, more data are needed to objectively judge their overall accuracy in the separation bubble.

Chapter 3
Scientific Visualization of Fluid Flows

This chapter aims to provide fundamental knowledge on the visualization of fluid flows by using ParaView and more recent cutting-edge technologies, such as virtual and augmented reality (VR/AR). At the same time, it has been developed to serve as a guide for prospective users. In the previous Chapter 2, the viscous flows and passive scalars' behaviors over the curved hill were scrutinized in a CFD approach (Paeres, Lagares, and Araya 2022a) (Paeres, Lagares, and Araya 2022b). In this chapter, scientific visualization is performed for those cases and additional numerical data with high spatial and temporal resolution. Results will take advantage of both Virtual Wind Tunnel (Paeres, Santiago, Lagares, Rivera, Craig, and Araya 2021) and FlowVisXR app (Paeres, Lagares, Santiago, Craig, Jansen, and Araya 2020) (Paeres, Lagares, and Araya 2021), tools developed for this research. At the end of this chapter, readers are expected to understand extended reality (XR) visualization at the fundamental level, enough to explore the broad options and workarounds currently possible for vast applications and those soon to be discovered with the continuing XR technology rise.

3.1 CFD Data Post-processing for XR Visualization

In CFD, post-processing is a set of custom steps for a specific investigation performed after the simulations are completed. Generally, the results of CFD simulation contain basic fields like velocity, pressure, and temperature in their solutions. To properly execute an assessment, the data needs to be post-processed. Mentioning all the possible steps for flow visualization post-processing would be absurd; it is more convenient to show examples directly involved with the assessment of the presented work and add other simple examples using Big Data files. The most straightforward visualization is presenting the whole simulation's domain with one primary field, for example, the temperature, the velocity's magnitude, or only one of its components. A slightly more advanced step would be to extract an iso-surface or the entire colored domain using a custom colormap or contour to the range of values of the displayed field. An iso-surface is a set of surfaces with a constant value that has been specified. At the same time, an iso-contour would be

a set of iso-surfaces with several ranges of values. These ranges can be differentiated by applying different colors to each iso-surface. From the author's experience, an efficient approach for extracting an iso-surface from 3D volumetric data is the Marching Cubes Algorithm. This algorithm can be applied on scripting platforms that read and manipulate the data of the files that contain the CFD solutions (e.g., MatLab and Python). 3D volumetric data refers to a three-dimensional array structured in the form (x,y,z), where each element in this array contains the value of the field of interest. The element's indices are related to its spatial position in the domain of the CFD case.

In this chapter, the intention is to present step by step one technique to visualize fluid flows conveniently for the user, especially if he or she is unfamiliar with scientific visualization. It is worth mentioning that before developing the following methodology, an attempt to export iso-surfaces in Wavefront OBJ (.obj) format with the Marching Cubes Lewiner version (Lewiner, Lopes, Vieira, and Tavares 2003) was made in Python. The choice to explicitly implement the Marching Cubes Lewiner algorithm was discarded because the most straightforward implementation was to extract iso-surface to only rectangular 3D volumes. For non-rectangular domains, more calibration and complex techniques were required for the extraction of iso-surface for each specific CFD case. Because the goal is scientific visualization, the virtual objects must have true shapes & constructions, and not fictional designs. Thus, the real appearance's validation occurs in the CFD simulations process, where the results are from real equations and models.

In the search for ways to visualize CFD results without many complications and for the methodology to be capable of being automated, the option of the ParaView visualization toolkit arose. *ParaView* is a free and open-source multiple-platform software for scientific visualization developed by Kitware Inc., Sandia and Los Alamos National Labs. Another open-source tool highly used for 3D computational graphics (i.e., 3D-printed models, animated films, visual effects, motion graphics, interactive 3D applications, virtual reality, and more.) is called Blender. However, Blender moves away from scientific visualization and leans more towards object design, modeling, color rendering, and animation, a profession apart from a CFD analyst. It is more convenient to use ParaView since it has been gaining ground in academia over the years. The tools focus on scientific visualization instead of just 3D animation. Here, an essential ParaView tool will be used to automate the methodology facilitating the scripting in Python. From ParaView 4.2 version and on, there is a function called *Trace*, which records every modification made in the ParaView user interface (UI) and generates a Python script containing all the actions in chronological order.

Regarding flow animation and the term FPS (frames per second), there are *active FPS*

and *natural or genuine FPS*. The *active FPS* is used in the visualization process (i.e., prescribed in Unity for display rate). Meanwhile, the *natural FPS* is based on the simulation's true time steps and sampling. Using the genuine FPS as the active FPS causes the animation to present the flow's real velocity; this natural FPS should be the max limit for the active FPS. Prescribing lower FPS values results in slow motions, which usually helps to observe turbulent structures. It is important to acknowledge other FPS limitations, like the device's display max FPS and the eye's max frame rate perception.

3.1.1 Manual and Single File Demo

ParaView has a wide variety of file formats it can read. On every version update, they tend to include more compatible formats. The guide presented here is based on ParaView 5.10.1. In the CFD science field, simulation results can usually be saved in .vts, .vtk, or other extensions of the Visualization Toolkit data type family. Figure 3.1 shows as an example the "Open" button pressed (which is also in the "File" tab). The .vtk file to be opened is specified after searching for it in its directory. Suppose the file is not shown as available in the list; probably, in that case, it is necessary to specify the data type extension of the working file.

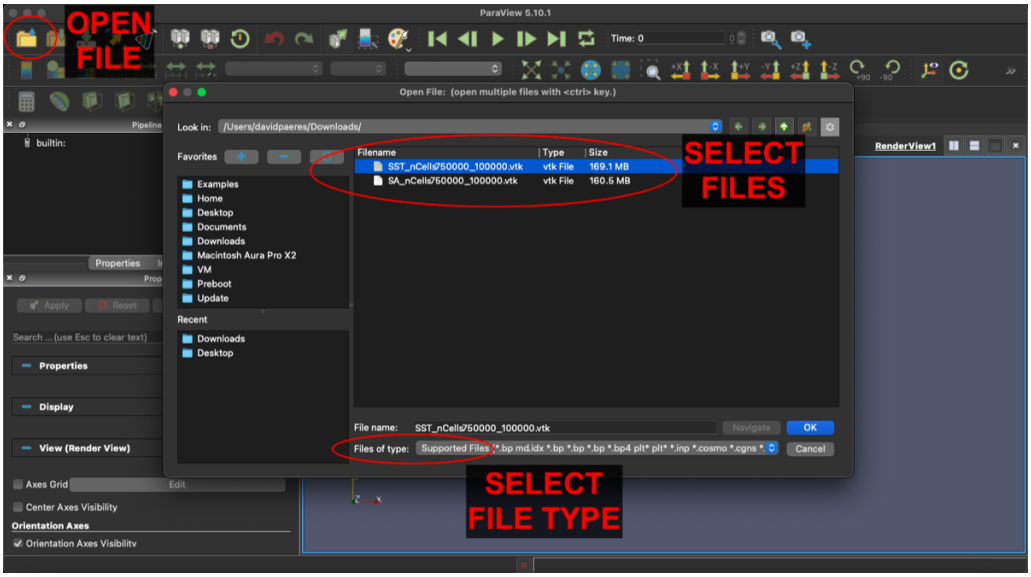


Figure 3.1: Manually importing files in ParaView 5.10.1.

In figure 3.2 are shown the three drop lists needed to visualize the solutions. At the top left, the field or parameter to be displayed is specified. ParaView can interpolate values between cells, so it might seem that there are repeated fields but what happens

is that for the same parameter, it can have *Cell Data* values which are the raw values of the cells' center and can also have *Point Data* values which are the interpolated values at the points that create the cells (Cell points). The second drop list is to specify the component or the magnitude chosen from the vector field. The third drop list is to specify the rendering mode. However, the presented examples will always use *Surface*. It is essential to mention that sometimes the changes are not automatically applied. For that, the *Apply* button will update the changes made in the visualization.

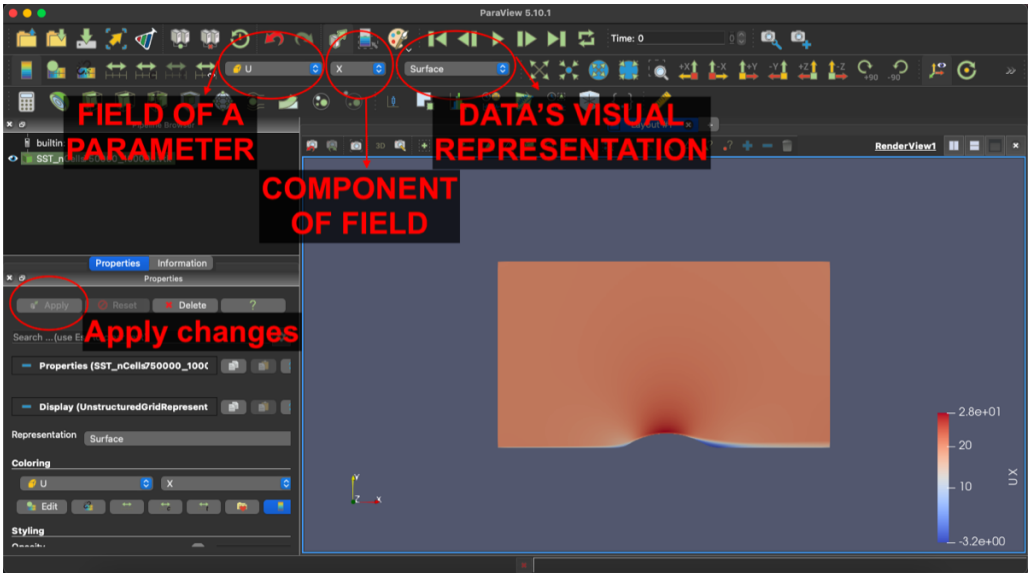


Figure 3.2: Manually selecting a field to visualize in ParaView 5.10.1.

Data manipulations and post-processing calculations have probably been carried out before opening the data in ParaView. If that is not the case, here will be shown to calculate streamlines from the velocity vector and apply a contour to it from the x component of the velocity. After selecting the file in the *Pipeline browser*, with the field and component of interest, press the *Stream Tracer* button (which can also be found in the *Filters/Alphabetical* tab). After that is done, Figure 3.3 shows the new *Pipeline Browser*'s component called *StreamTracer1*. Note that below it is specified to which vector the streamlines are calculated. Further down is the type of seed, which in this case was changed to *Point Cloud*. After scrolling down in the *Properties* window, Figure 3.4 shows how the location of the seed is changed to the coordinate (x=0.8, y=0.075, z=0.025) and radius = 0.5 to focus on the flow's separation zone. ParaView has no concept of units. See how the *Number of Points* is increased to 1000 to have more streamlines included. Also, the visualization is coloring the surface with a contour of the x component of the velocity U.

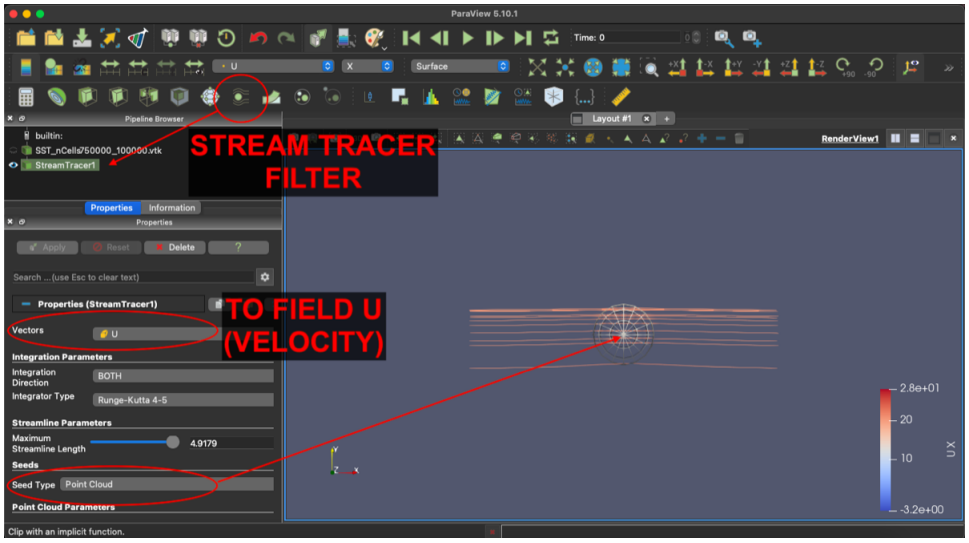


Figure 3.3: Manually applying Stream Tracer filter in ParaView 5.10.1.

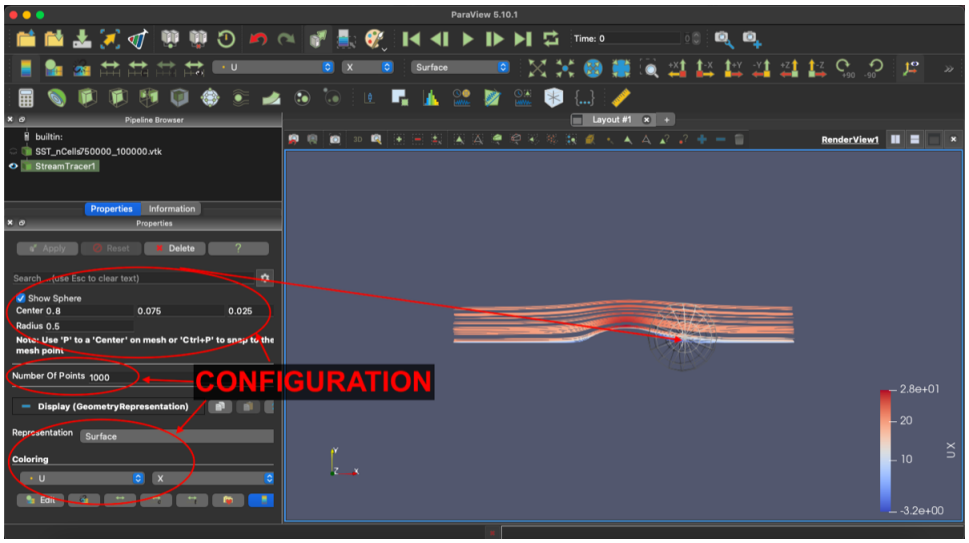


Figure 3.4: Manually configuring Stream Tracer filter in ParaView 5.10.1.

Although everything seems neat, adding volume to each streamline is better for achieving virtual objects that look genuinely 3D. This can be done with a filter called *Tube*. Note it this must be applied to the data set *StreamTracer1* of the *Pipeline Browser*. Figure 3.5 shows the application of the *Tube* filter with a radius of 0.05 and how the recirculating flow streamlines are visible right in the flow separation zone.

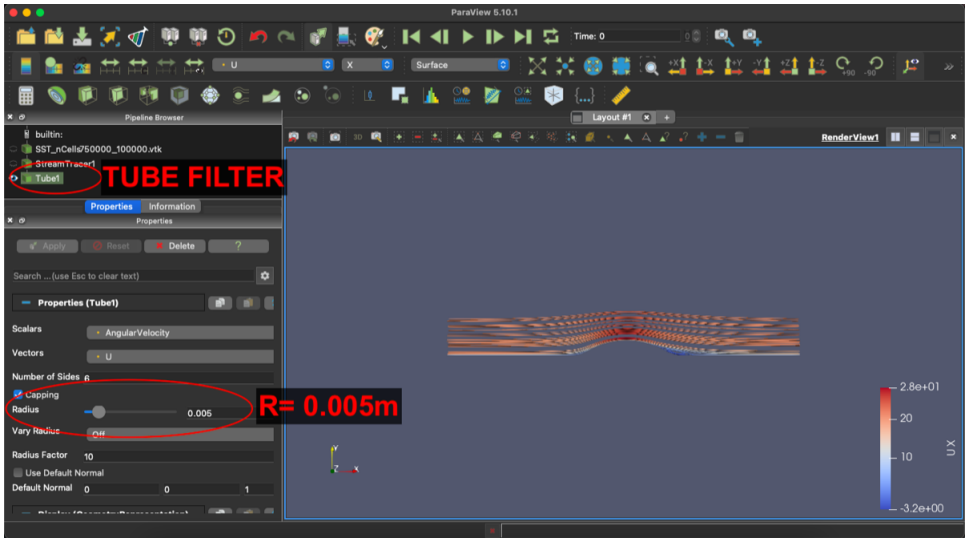


Figure 3.5: Manually applying Tube filter in ParaView 5.10.1.

After developing the 3D visualization approach in ParaView, the next step consists of exporting it. Since the ParaView 5.7 version, there is an option to export files in GLTF format, which will be used as an intermediate format to convert it to USDZ further. One should select *Export Scene* located in the *File* tab and specify the file type **.GLTF Files (*.gltf)* include a new file name. As shown in Figure 3.6, check all boxes in *Export Options* and *Save* it; this will export the whole scenario that is displayed and rendered in ParaView as GLTF format.

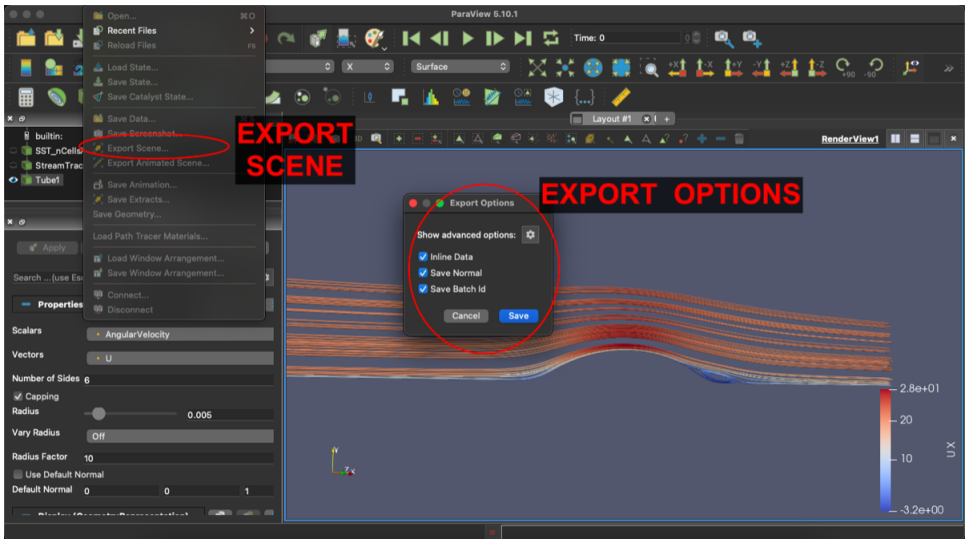


Figure 3.6: Manually exporting scene as GLTF in ParaView 5.10.1.

3.1.2 Automated and Multiple Files Demo

So far, it has been instructed on how to read a single file, apply a filter, and export it in GLTF format. It is possible and highly convenient to perform the same procedure automatically for multiple files employing the *Trace* tool. The intention is to obtain a Python script containing all the actions' commands to automate the process, including reading multiple files. This example uses eleven files, each representing a different time-step; these files correspond to the work of Lagares, Paeres, and Araya (2021), where are high-fidelity numerical results of supersonic spatially-developing turbulent boundary layers (SDTBL) subject to strong concave and concave curvatures and Mach = 2.86. There, a DNS simulation used a time-step of $1E-4$ seconds, resulting in a 10,000 Hz or FPS. However, because the sampling was every ten frames, the natural FPS is 1,000Hz. It is only needed to export two files from the whole file group in the *Trace* process and then edit the Python script for looping the export command for all the desired files. Figure 3.7 shows the first step, which is to turn on the tracing tool in the *Tools/Start Trace* tab. Also, the configuration includes tracing *all properties*, *Supplemental Proxies*, and *Show Incremental*. The *Skip Rendering* option must not be activated for accurate scene creation. After pressing the *Ok* button, as soon as ParaView recognizes the first action, a new window will appear with the Python script in progress. Several VTS files will be read in ParaView for this new example simultaneously. A very convenient ParaView feature is the ability to recognize file groups having the same characters at the beginning of their names while reading them. Figure 3.8 serves as an example; note how ParaView recognizes the selected file as a group.

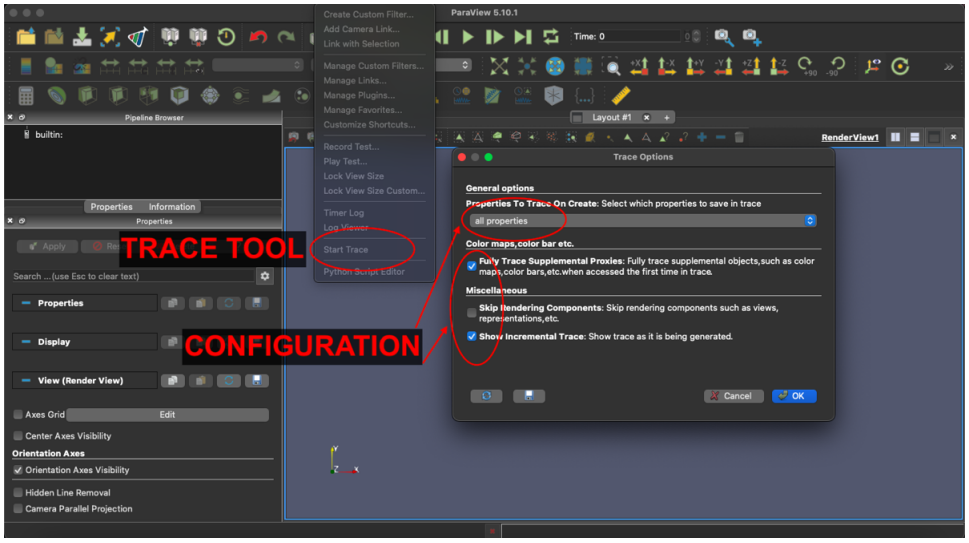


Figure 3.7: Activating Tracing tool in ParaView 5.10.1.

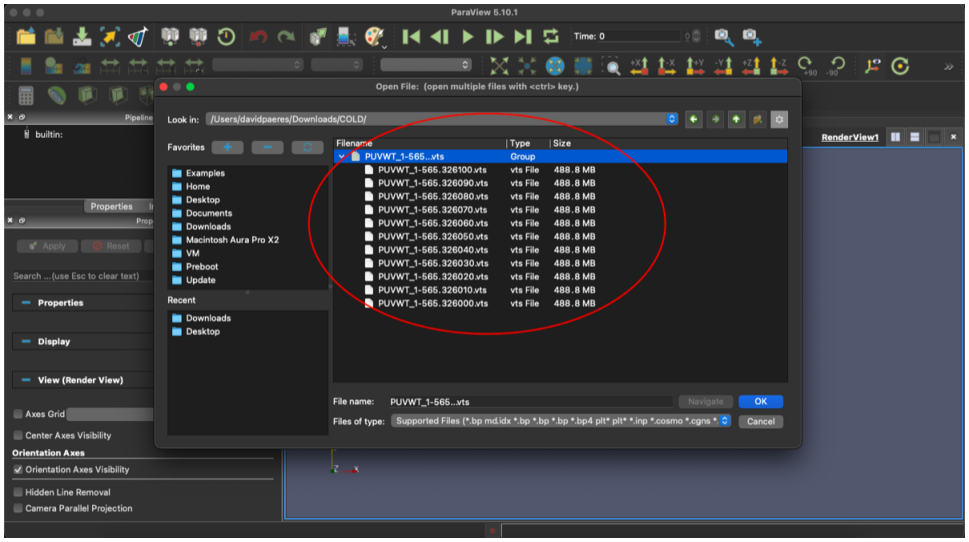


Figure 3.8: Opening a group of files in ParaView 5.10.1.

After opening files with the tracing tool on, the window of the Python script should be visible. The example here is to extract an iso-surface from the Q-Criterion parameter (Hunt et al. 1988) and color it with the temperature field. Figure 3.9 shows how a single iso-surface was extracted with the value of Q-Criterion = 2500 and colored by the temperature field by applying the filter *Contour*.

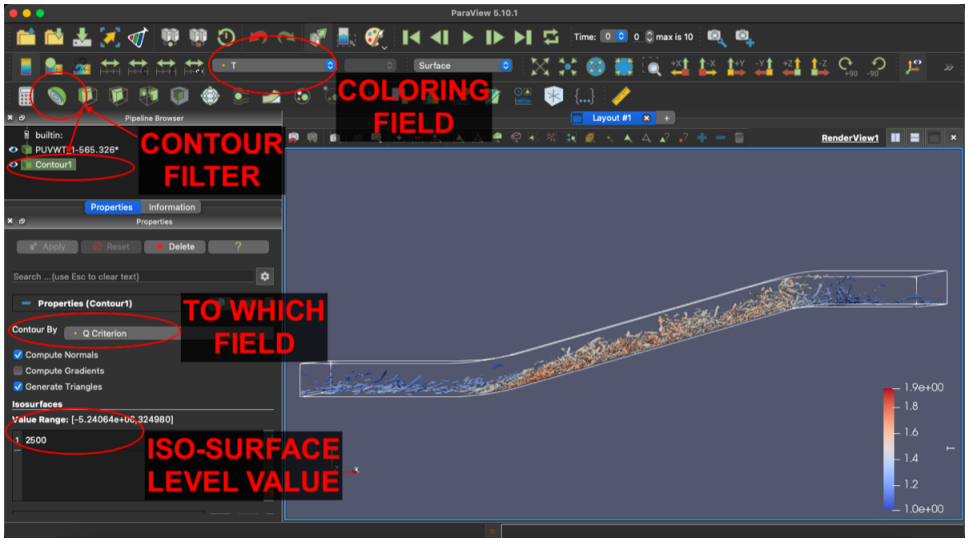


Figure 3.9: Applying *Contour* filter to Q-criterion field and coloring with temperature field in ParaView 5.10.1.

With the scene ready, it will now be exported as GLTF format following the same steps shown for Figure 3.6. Figure 3.10 shows the scene corresponding to time-step 0 being exported with the name *Qcriterion2500_colorTemp0.gltf*. Figure 3.11 shows the button for the next time-step (in this case is 1) and also how this new scene is exported with the corresponding name *Qcriterion2500_colorTemp1.gltf*.

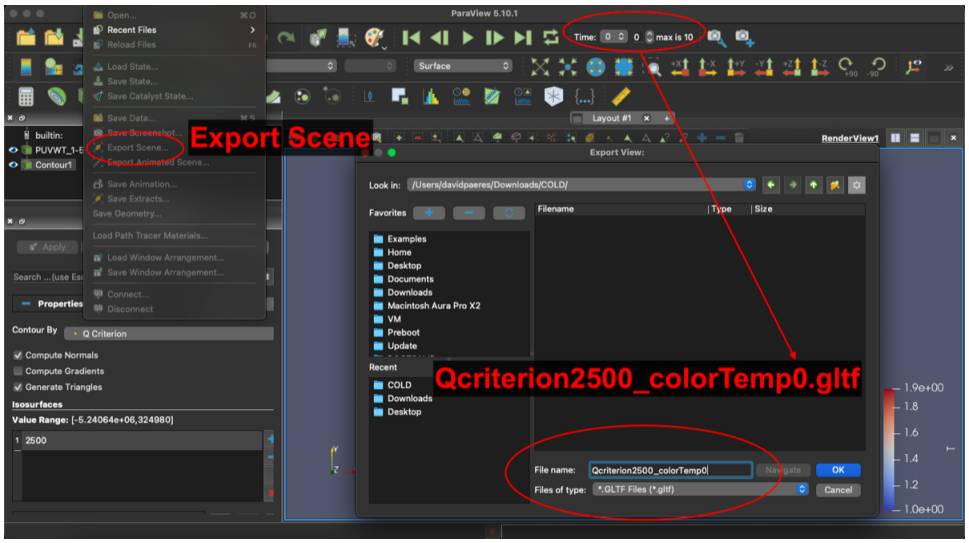


Figure 3.10: Exporting scene of the *Contour* filter for time step 0 as GLTF.

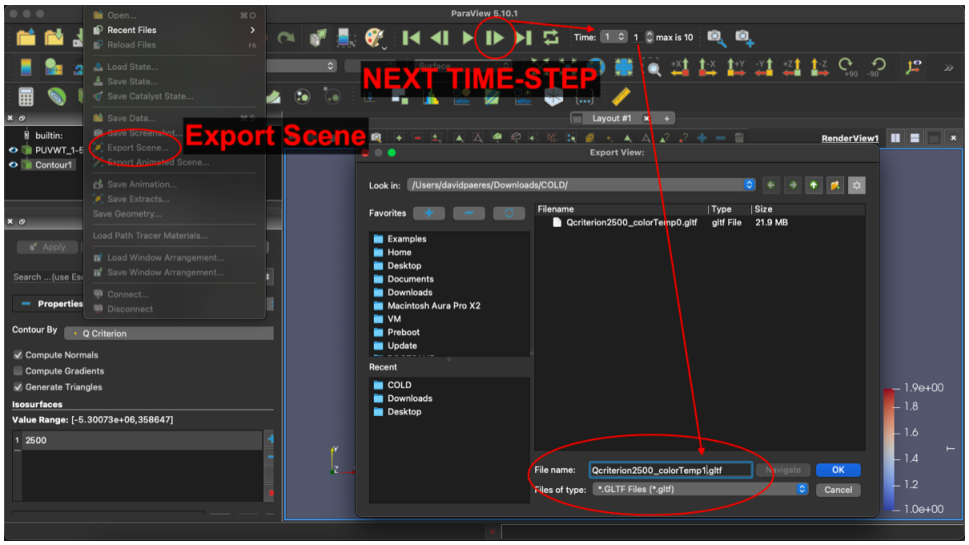


Figure 3.11: Exporting scene of the *Contour* filter for time step 1 as GLTF.

As two time-step scenes have already been exported, the Python script has essentially

the commands wanted. The tracing is stopped, which is achieved with the *Stop Trace* button located in the *Tools* tab. Figure 3.12 shows how the Python script is saved from its corresponding window after the tracing is stopped.

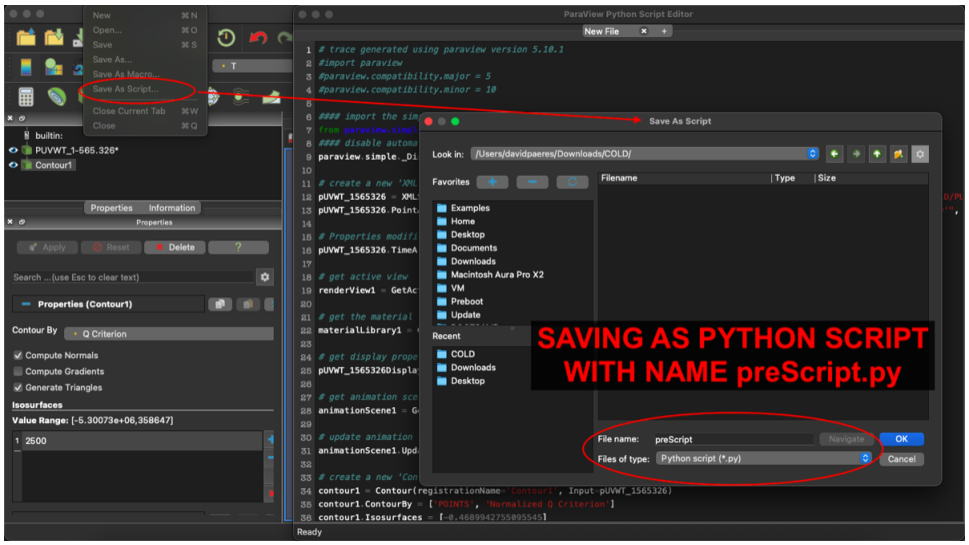


Figure 3.12: Saving tracing commands as a Python script.

By opening the Python script *preScript.py*, there are several essential code lines to identify. Figure 3.13 shows lines 12 and 13 as marked, where line 12 indicates which files are to be opened in ParaView, specified as an array of strings called *FileName*. Line 13 specifies the fields to be read from the VTS files. Line 12 can be edited to avoid manually typing a long list of names, while line 13 does not require any change.

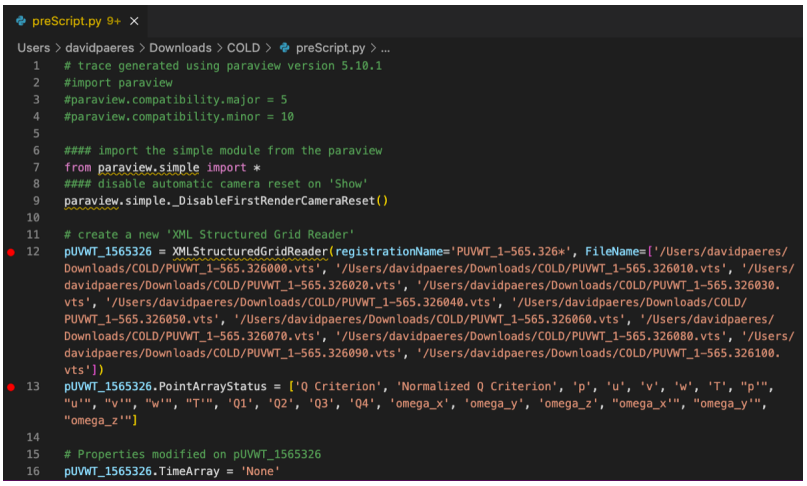


Figure 3.13: First command lines of the Python script: *preScript.py*.

Figure 3.14 shows lines 33 through 45, where the *Contour* filter starts to be applied. Note that the "Input" in line 34 is the variable *pUVWT_1565326* previously defined in line 12. Figure 3.15 presents lines 87 and 99, which specify that the iso-surface should be from the Q-Criterion field and for a value of 2500, respectively.

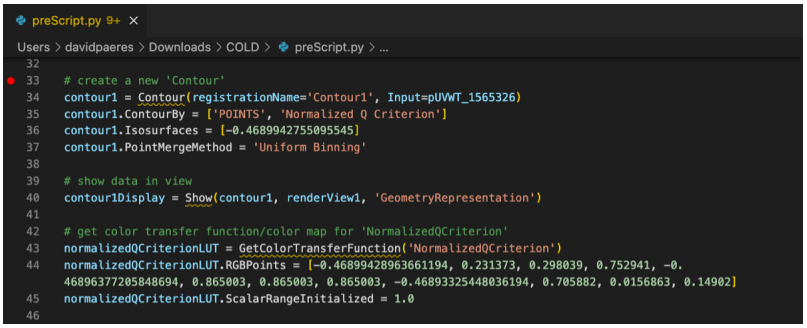


Figure 3.14: Command lines where a default *Contour* filter is applied in *preScript.py*.

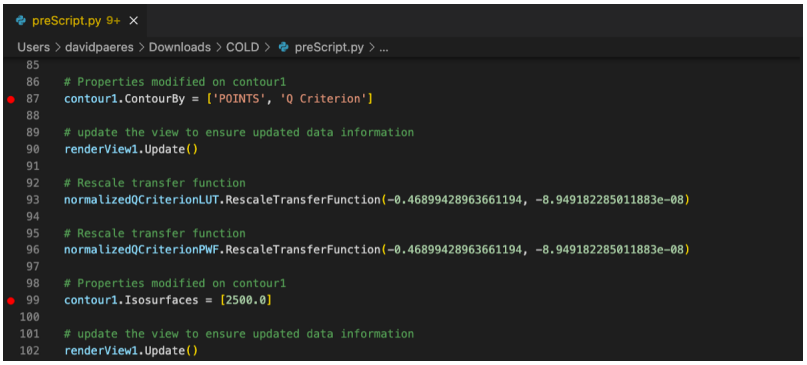


Figure 3.15: Setting the correct *Contour*'s field and value in *preScript.py*.

Figure 3.16 shows lines 111, 123, and 128, specifying that the extracted iso-surface should be colored using the temperature field. Although lines 87, 99, 111, 123, and 128 have been marked as important, it is recommended not to edit them since other supplementary lines accompany them. If other commands are desired to be added, it would be more appropriate to repeat the *Trace* process in ParaView, but with the new steps to be implemented. This advice is recommended until the user acquires extensive knowledge of ParaView's Python scripting and understands each command line well.

```
preScript.py 9+ ×
Users > davidpaeres > Downloads > COLD > preScript.py > ...
108  normalizedQCriterionPWF.RescaleTransferFunction(-0.46899428963661194, 0.000477100722491741 2)
109
110  # set scalar coloring
111  ColorBy(contour1Display, ('POINTS', 'T'))
112
113  # Hide the scalar bar for this color map if no visible data is colored by it.
114  HideScalarBarIfNotNeeded(normalizedQCriterionLUT, renderView1)
115
116  # rescale color and/or opacity maps used to include current data range
117  contour1Display.RescaleTransferFunctionToDataRange(True, False)
118
119  # show color bar/color legend
120  contour1Display.SetScalarBarVisibility(renderView1, True)
121
122  # get color transfer function/color map for 'T'
123  tLUT = GetColorTransferFunction('T')
124  tLUT.RGBPoints = [1.0065757036209106, 0.231373, 0.298039, 0.752941, 1.4728603959083557, 0.865003, 0.865003,
     0.865003, 1.9391450881958008, 0.705882, 0.0156863, 0.14902]
125  tLUT.ScalarRangeInitialized = 1.0
126
127  # get opacity transfer function/opacity map for 'T'
128  tPWF = GetOpacityTransferFunction('T')
129  tPWF.Points = [1.0065757036209106, 0.0, 0.5, 0.0, 1.9391450881958008, 1.0, 0.5, 0.0]
130  tPWF.ScalarRangeInitialized = 1
```

Figure 3.16: Setting the correct *Contour*'s coloring field in *preScript.py*.

After completion of *preScript.py*, Figure 3.17 shows how in line 133, the scene is exported for time-step 0 with its custom name. It also shows how in line 137, it is passed to the next time-step (i.e., time step 1) and that in line 140, the new scene is also exported with another custom name. Lines 133 to 143 can be edited to run in a loop generating all the wanted GLTF files. From line 144 onwards, the lines can be deleted since they are irrelevant to the presented methodology.

```
preScript.py 9+ ×
Users > davidpaeres > Downloads > COLD > preScript.py > ...
131
132  # export view
133  ExportView('/Users/davidpaeres/Downloads/COLD/Qcriterion2500_colorTemp0.gltf', view=renderView1,
     InlineData=1,
134      SaveNormal=1,
135      SaveBatchId=1)
136
137  animationScene1.GoToNext()
138
139  # export view
140  ExportView('/Users/davidpaeres/Downloads/COLD/Qcriterion2500_colorTemp1.gltf', view=renderView1,
     InlineData=1,
141      SaveNormal=1,
142      SaveBatchId=1)
143
144  #================================================================
145  # addendum: following script captures some of the application
146  # state to faithfully reproduce the visualization during playback
147  #================================================================
148
149  # get layout
150  layout1 = GetLayout()
151
152  #--------------------------------
153  # saving layout sizes for layouts
```

Figure 3.17: Command lines exporting two scenes as GLTF in *preScript.py*.

As mentioned for Figure 3.13, creating input variables to edit the *preScript.py* can control which files are processed and exported as GLTF in order. Figure 3.18 shows the inclusion of 7 code lines (plus two empty) at the beginning of the script. Line 1 is to import the

glob library that helps to read the list of files in an alphabetically organized way. Line 2 is for writing the working directory path and is used to build the absolute paths of each input and output file. Line 3 identifies the VTS files' directory relative to the working directory path. Line 4 is the name's form of the input files, the wildcard character (*) is used to include all the files with names containing the same characters before the wildcard character. Line 5 defines the value of the iso-surface that will be extracted from the Q-criterion field. Line 6 is the analog of line 3 but for the GLTF files' directory, with the path relative to the working directory given in line 2. The GLTF files' directory must already be created before running the Python script. Lastly, line 7 is where an array of strings is built containing all the names of the VTS files in alphabetical order to be further processed one by one. Notice how in the new line 21, to *registrationName* argument is given the *vts_groupName* variable. Also, to *FileName* argument is given the *vts_FileNames* variable, avoiding the need to explicitly write all the names with the absolute path of each VTS file.

preScript.py 9+

Users > davidpaeres > Downloads > COLD > preScript.py > ...

```
1  import glob
2  workingPath = '/Users/davidpaeres/Downloads/'
3  vts_directory = 'COLD/'
4  vts_groupName = 'PUVWT_1-565.326*'
5  isoValue= 2500.0
6  gltf_directory= 'Qcriterion2500_colorTemp/'
7  vts_FileNames = sorted(glob.glob(f'{workingPath}{vts_directory}{vts_groupName}'))
8
9
10 # trace generated using paraview version 5.10.1
11 #import paraview
12 #paraview.compatibility.major = 5
13 #paraview.compatibility.minor = 10
14
15 #### import the simple module from the paraview
16 from paraview.simple import *
17 #### disable automatic camera reset on 'Show'
18 paraview.simple._DisableFirstRenderCameraReset()
19
20 # create a new 'XML Structured Grid Reader'
21 pUVWT_1565326 = XMLStructuredGridReader(registrationName= vts_groupName , FileName=vts_FileNames)
22 pUVWT_1565326.PointArrayStatus = ['Q Criterion', 'Normalized Q Criterion', 'p', 'u', 'v', 'w', 'T', "p'",
   "u'", "v'", "w'", "T'", 'Q1', 'Q2', 'Q3', 'Q4', 'omega_x', 'omega_y', 'omega_z', "omega_x'", "omega_y'",
   "omega_z'"]
```

Figure 3.18: New lines added for the input files in *preScript.py*.

Figure 3.19 shows the new line 108, where previously the value 2500 was explicitly held, but now it is changed with the variable *isoValue*, prescribed in line 5 of Figure 3.18; allowing the reuse of the same Python script for iso-surface extracting with other iso-values.

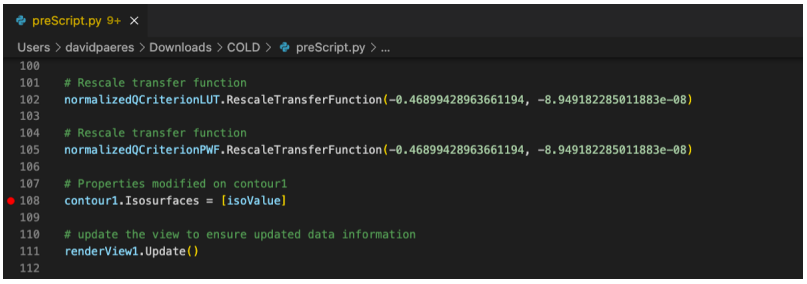


Figure 3.19: Making modular the iso-value used in *preScript.py*.

To conclude the Python script edition, see Figure 3.20. Notice how on line 145, a *for* loop is created to export each file in the *vts_FileNames* array. In line 146, a new variable called *gltf_fileName* is created for each file at a time, where the GLTF files' directory replaces the string's part corresponding to the VTS files' directory. In line 147, the variable *gltf_fileName* has the *.vts* extension replaced with *.gltf*. Having set the name and full path of the current GLTF files pending to be exported, line 150 does the export. Then, line 152 commands ParaView to move to the next timestep in the scene, and from there, the loop starts again in line 145 until all VTS files contained in the *vts_FileNames* array are processed.

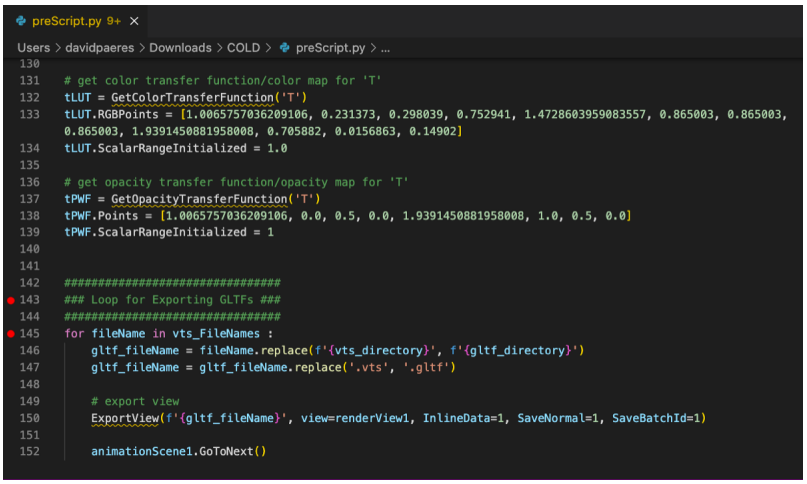


Figure 3.20: Command lines with the exporting loop for every time step in *preScript.py*.

After upgrading the *preScript.py*, it should be run from ParaView's Python shell, which includes its required libraries. Figure 3.21 shows where the *Python Shell* window can be activated and the *Run Script* button used to execute the script. Also, the screenshot was captured after all time steps were exported; the maximum time step is ten since the

count starts from zero. Figure 3.22 shows the result after extracting all the iso-surfaces. Note that the VTS files contained several parameter fields with the entire CFD domain. In contrast, each GLTF file is only an iso-surface of approximately 22 MB in memory storage size. For the final version of *preScript.py*, the reader is referred to Appendix B.

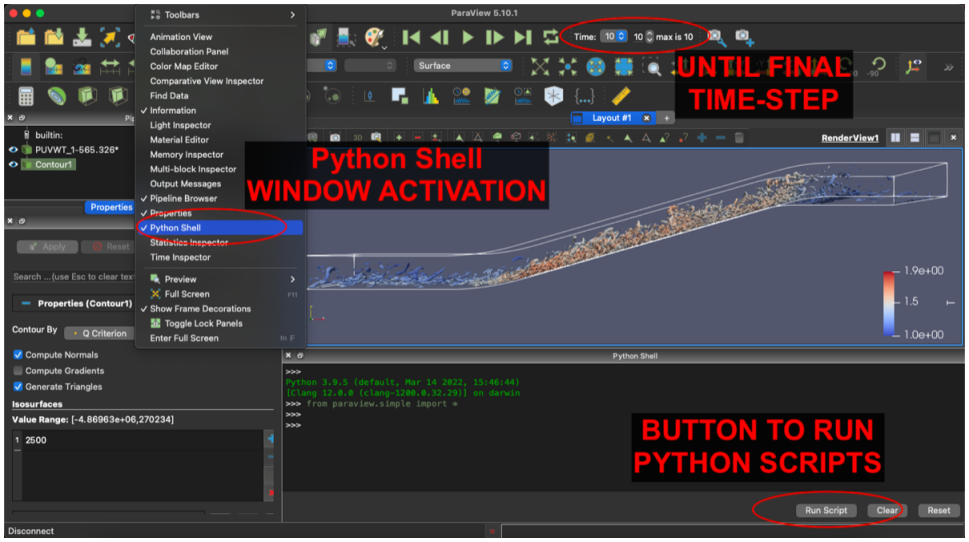


Figure 3.21: Running *preScript.py* in ParaView 5.10.1.

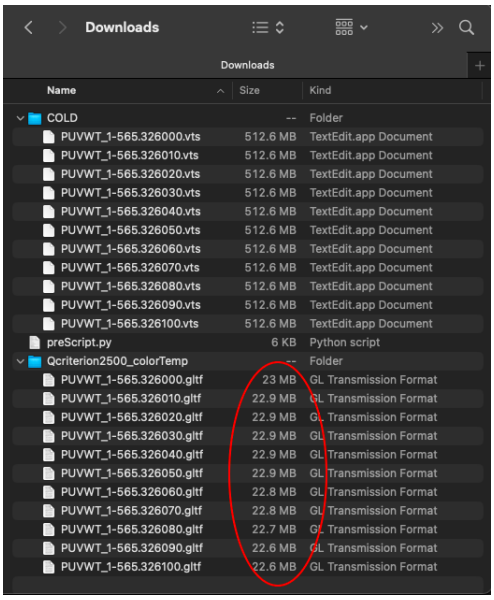


Figure 3.22: Resulting list of files from VTS to GLTF.

3.2 Augmented Reality with USDZ

The steps for AR flow visualization depend on the final goal and the interactive degree of virtual objects in the real world. Suppose the idea is only to invoke virtual objects in the real world via iOS devices; in that case, the USDZ file format is optimal because Apple devices have built-in apps capable of reading the USDZ files. On the other hand, if the device to use is not iOS (e.g., Android and HoloLens), a compatible app will be needed to import the USDZ files. Therefore, a new application was created for this work using the Unity game engine platform. The benefits of designing the apps with Unity 3D are extending the AR applicability to Android and HoloLens devices and increasing opportunities to develop interactive manipulations, such as image recognition, which will be shown and discussed in Section 3.4. Regardless of the final goal with AR, in this methodology, it is necessary to transform the GLTF files to USDZ format. It will be presented how to do it from a macOS.

3.2.1 Manual and Single File Demo

Apple has several tools available to convert files from GLTS to USDZ on their page: https://developer.apple.com/augmented-reality/tools/. This site mentions four software as downloading options. However, for simplicity, this guide will be only interested in *Reality Converter* and *USDZ Tools*. Reality Converter is the most intuitive option to convert, edit and view customized USDZ. This software allows file importing in OBJ, GLTF, and FBX formats by just dragging and dropping to convert them to USDZ. Figure 3.24 shows the Reality Converter interface. Notice how, after importing one GLTF file from the example of the previous section, it is colored yellow. The yellow coloring happens because ParaView cannot export GLTF files preserving the colors used in the visualization. However, these files conserve the gradient or distribution of the coloring applied. ParaView, by default, applied whatever *Base color* is specified. Since the intention is that the contour represents the temperature field, by applying the following Figure 3.23, the color map of the USDZ is now a gradient between blue and red. Figure 3.25 shows how to export the USDZ files with a simple drag and drop of images to add the desired color map.

Figure 3.23: Picture of blue-to-red color gradient used for the USDZ file creation.

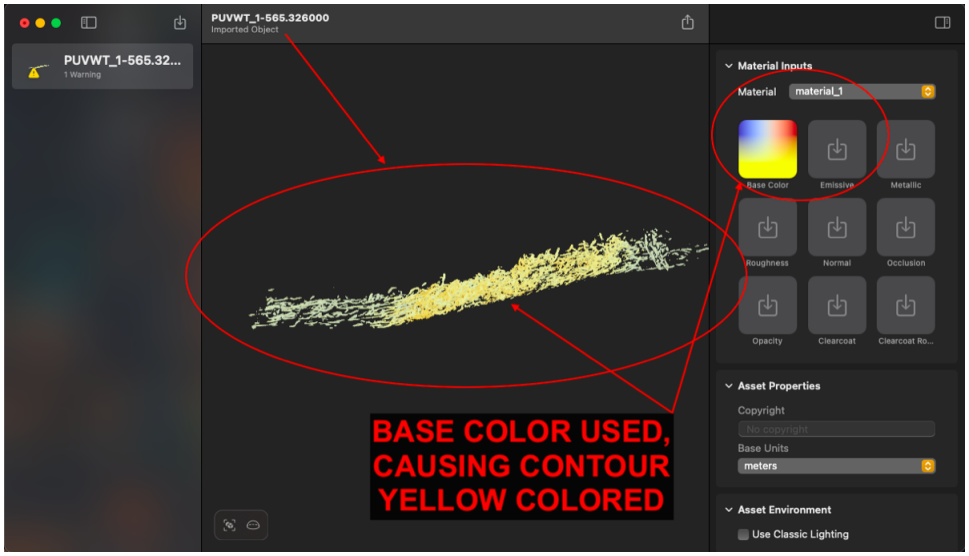


Figure 3.24: Reality Converter software interface with one GLTF file imported.

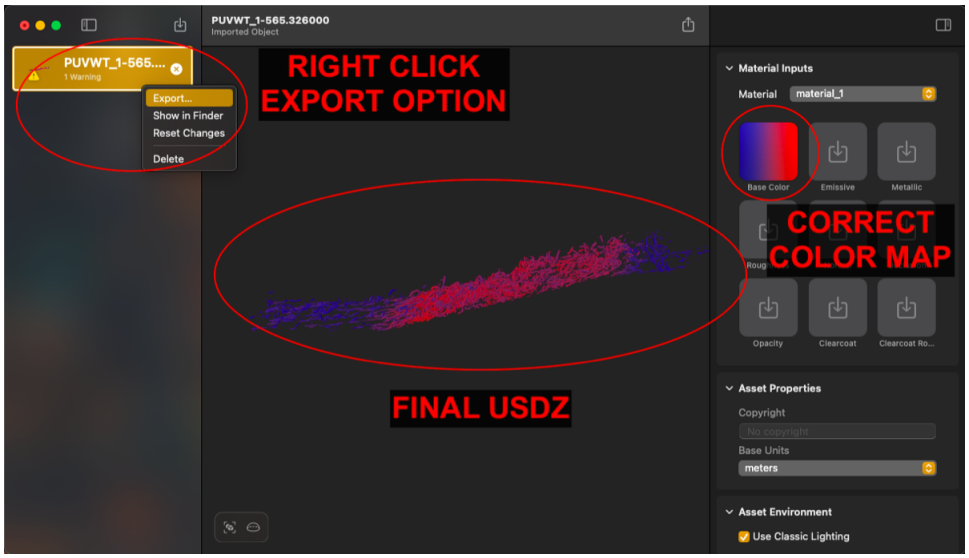


Figure 3.25: Exporting USDZ using Reality Converter.

3.2.2 Automated and Multiple Files Demo

Reality Converter is very intuitive due to its GUI (Graphical User Interface), but converting many files to USDZ, is not the most viable option. For multiple files, it is better to use USDZ Tools (also available on Apple's website: https://developer.apple.com/augmented-reality/tools/). USDZ Tools or USD Python Tools is

a Python-based pre-compilation with more tools than Reality Converter for modifying USDZ files. This means converting multiple GLTF files to USDZ can be automated with a single Python script. The software installation is simple but has a minor issue since the pre-compilation was done with Python 3.7.9, requiring this specific version of Python to be installed for compatibility in the libraries. To launch the tool, go to the installed directory called *usdpython* and double-click the file called *USD.command*. It opens a new terminal window for command lines with the proper environment. It is recommended to check if the USDZ Tools' terminal window is running with Python 3.7.9 version; there are usually bash or zsh profiles that automatically load other versions of Python, and it may be necessary to temporarily remove these commands from the profiles while using USDZ Tools.

Figure 3.26 presents the Python script *export_usdz.py*, which converts all the GLTF files created in the previous subsection to USDZ. In line 2, the OS library is imported to execute commands in the terminal from within the Python script. This short script has the same structure as *preScript.py* with the automated *for* loops; the newness of *export_usdz.py* are lines 8 and 18. Line 8 is for declaring the name of the picture that will be applied as coloration; in this example, this is the same one that was dragged and dropped in Figure 3.25. Moreover, line 18 corresponds to the command line going to be executed in the terminal window *USD.command*, which converts the GLTF files to USDZ using a specific picture file for colorization. The command in line 18 follows the structure of: *usdzconvert <inputFile.gltf> -diffuseColor <colorMapPicture.png> <outputFile.usdz>*, where the name of the input files contain their absolute path, and the output file's extension can be any of the USD family (e.i., .usd , .usda , .usdz).

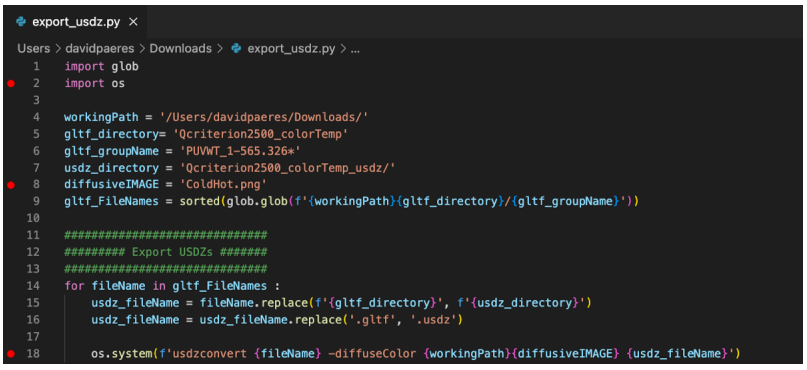


Figure 3.26: Example of Python script used to convert GLTF files into USDZ in an automated way.

Figure 3.27 shows two windows; on the left is the terminal created by "USD.command,"

and on the right are all the files used in the GLTF to USDZ conversion. It is important to note that the version of Python loaded in the terminal window is verified. The terminal's working directory is changed to the address where "export_usdz.py" is. Finally, the Python script is executed from a command line. Also, it is worth noting the storage-size reduction by approximately 50%; this is because the USDZ files are binary data files.

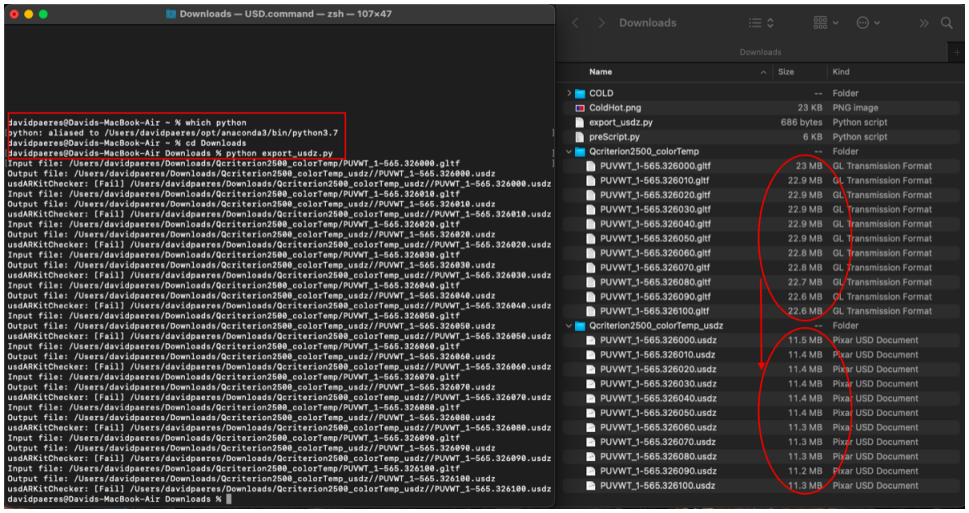


Figure 3.27: Terminal window of USDZ Tools with the recently created USDZ files.

Figure 3.28 shows as an example the final result of an AR object invoked and placed over a table; this visualization was done through iOS devices. The AR object was created following the methodology explained in this work.

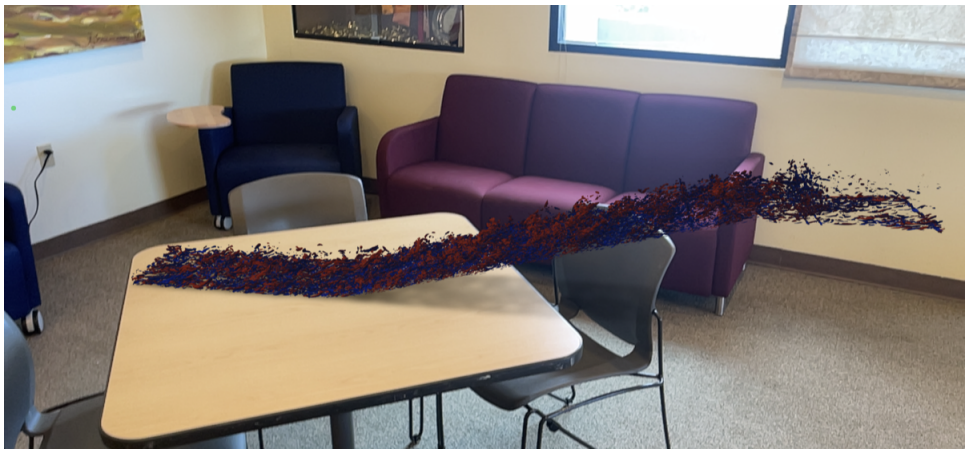


Figure 3.28: AR object visualized through an iOS device.

3.3 Unity Game Engine for XR

As said before, if the plan is to perform XR visualization with Android smartphones or XR Headsets (e.g., Microsoft HoloLens and HTC VIVE), we need to use platforms like Unity. In essence, the method is to create an application for the specific device, including the virtual objects desired to be visualized in the app. Unity is a game engine, which means it is for creating video games and can currently be installed on macOS, Windows OS, and Linux. The detail is that the game engine is generally more compatible and convenient to use on a computer with an operative system (OS) related to the target device. For example, for Android smartphones and Microsoft HoloLens is better to use a Windows computer. Unity from a macOS system can build Android apps. However, to build iOS apps, it is required to use Unity on a macOS system since it needs Apple's software Xcode.

The author recommends using an Apple computer with good storage, memory, and graphical specifications. With the help of Apple Boot Camp, it is straightforward to create a Windows partition, having both macOS and Windows OS available on the same computer. If the computer is limited in storage, the Unity projects can be saved on an external hard drive. This enables the projects' portability to other computers with the same Unity version installed. One advantage of using the platform Unity is the free licenses for personal use, meaning the methodology presented here is still free, presuming the hardware devices have already been bought. Also, designing the apps with Unity extends the applicability to other smartphones and XR devices. It enables opportunities to develop more interactive manipulations, such as image recognition.

The Unity installation begins from the official web page: `http://unity.com/developer-tools` with the option *Get started* to create a Unity account and obtain a free Personal License. The next step is downloading Unity Hub from the web page: `http://unity.com/download`. After the simple installation of Unity Hub, the next step is to sign in using the Unity account previously created and activate the license on Unity Hub. The final step is downloading a Unity editor from Unity Hub with the version and modules needed for the specific project.

3.3.1 FlowVisXR for the Microsoft HoloLens 1st Gen.

FlowVisXR is an application created for AR visualization as part of the presented work. This app has already been used in iOS devices and Microsoft HoloLens 1st generation (henceforth HoloLens1), as can be seen in Paeres et al. (2020) and Paeres et al. (2021). This sub-section will show how to replicate the FlowVisXR app with

the HoloLens1 as the target device.

Starting from Unity Hub without any Unity editor installed, the first step is to install Unity 2019.4.40f1 LTS with the modules: (i) Android Build Support, (ii) iOS Build Support, (iii) Universal Windows Platform Build Support, and (iv) Windows Build Support (IL2CPP). The last two modules mentioned (iii and iv) are required to build HoloLens1 apps. The module of Visual Studio 19 is not necessary since we are going to use Visual Studio 22. Figure 3.29 displays the Unity editor installed with its modules for FlowVisXR app creation.

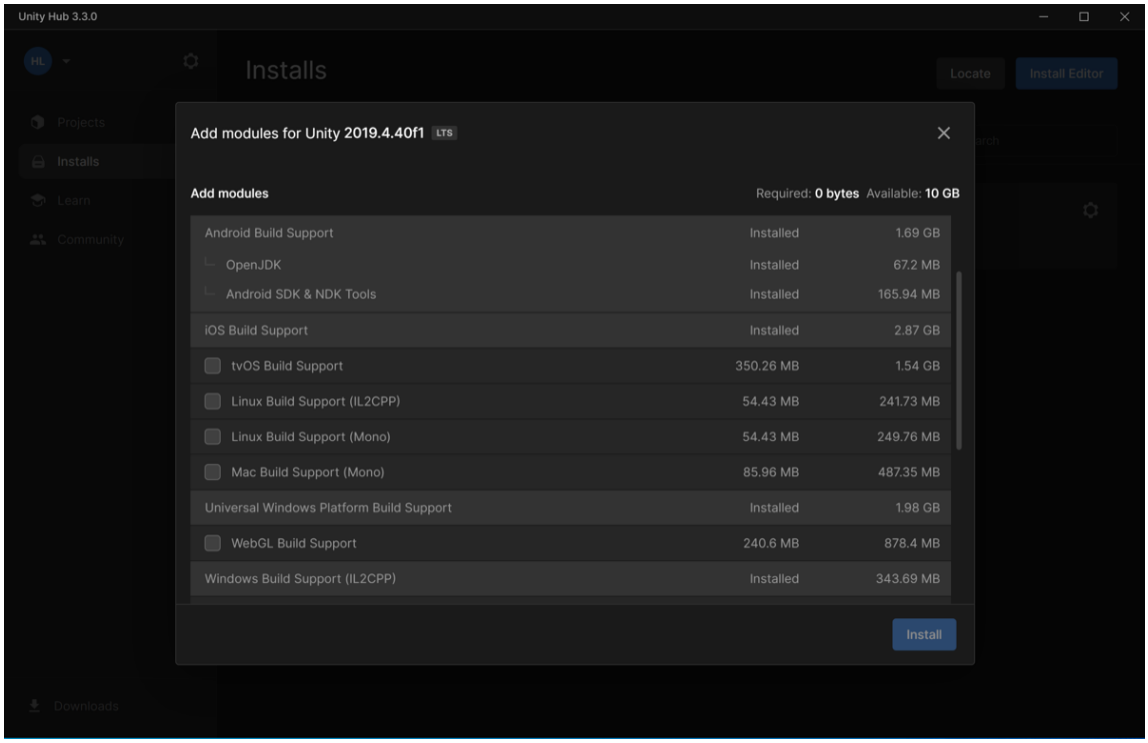


Figure 3.29: Unity 2019.4.40f1 LTS editor installed with its modules for FlowVisXR app.

FlowVisXR on Windows OS needs five extra tools and adjustments before starting the Unity project. These are: (i) Visual Studio, (ii) Git, (iii) Windows 10 SDK, (iv) Microsoft Reality ToolKit (MRTK), and (v) Developer mode activation in both computer system and target device.

Visual Studio 2022

Visual Studio 2022 can be obtained from their official web page: http://visualstudio.microsoft.com/downloads/, where there is a free license called Comunity. After the download, it is just to run *VisualStudioSetup.exe*, which offers to install a specific Visual Studio version with extra workloads and components. The workloads to include are: (i) Python development, (ii) .NET Multi-platform App UI development, (iii) .NET desktop development, (iv) Desktop development with C++, (v) Universal Windows Platform development, (vi) Mobile development with C++, (vii) Game development with Unity, and (viii) Game development with C++. Also, from the *Individual Components* tab, you will have to scroll down to find *SDKs, libraries, and frameworks* category and include *USB Device Connectivity* and *Windows 10 SDK 10.0.20348.0* (the most updated version) to press *Install* finally. Figure 3.30 displays an example of Visual Studio 2022 installation before including the Individual Components in the selection.

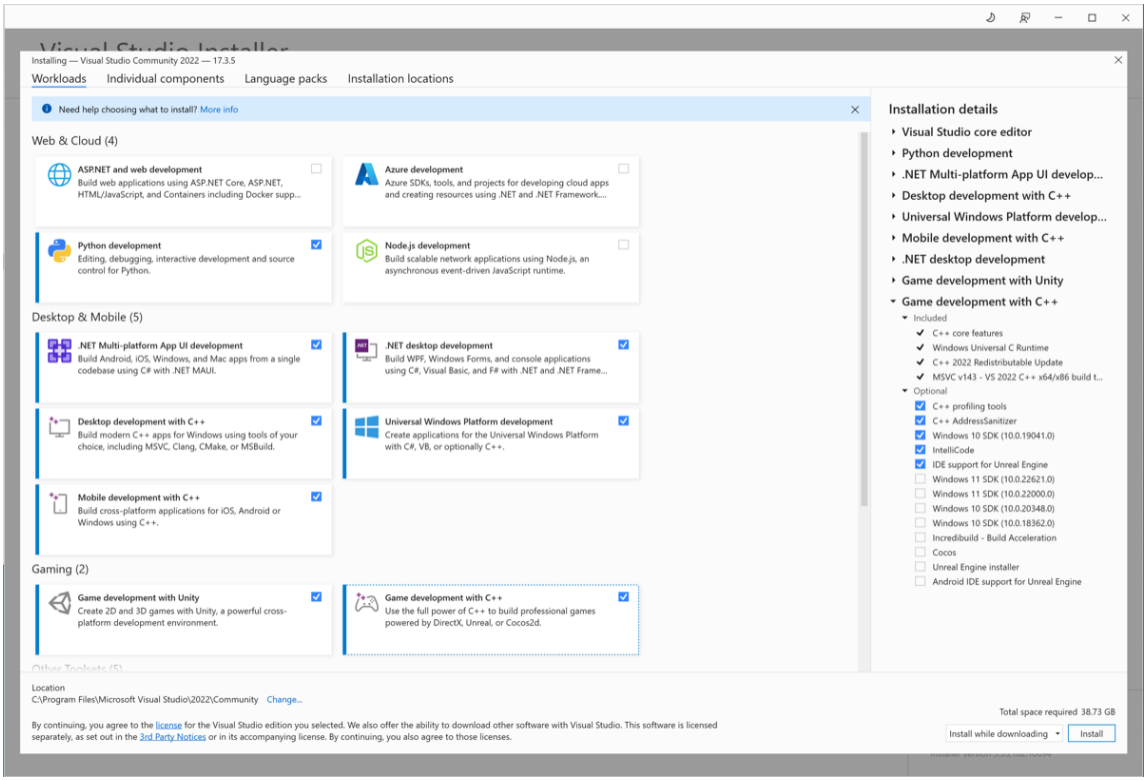


Figure 3.30: Visual Studio 2022 installation before including the Individual Components in the selection.

Git

One can obtain Git from their official web page: http://git-scm.com/downloads. At the moment of this research, the most updated version is 2.73.3, which has been installed using all default settings except for the *default editor by Git*, where Nano was selected. After Git is installed, it is required to confirm that it has been added to the system's Environment Variables. To do this, open *System's Settings* (right-click on the Windows Start icon) and the Environment Variables window. Confirm there is a variable named *Git*, and the value directs to the executable file located in Git's installation folder. Usually, the installation address is in *C:\Program Files\Git\cmd*. If the Git variable is not there, create a new one.

Windows 10 SDK

Windows 10 SDK can be obtained from the web page: http://developer.microsoft.com/en-US/windows/downloads/windows-sdk. The installation is very straightforward and does not require further details.

Microsoft Reality ToolKit

Microsoft Reality ToolKit (MRTK) can be obtained from their official GitHub web page: http://github.com/Microsoft/MixedRealityToolKit-Unity/releases. Because we will be using HoloLens1, the specific version we need is v2.4.0 which can be found by scrolling down through the releases list. After finding the correct release version, open the drop-list named *Assets* and download the following Unity packages:
(i) Microsoft.MixedReality.Toolkit.Unity.Foundation.2.4.0.unitypackage,
(ii) Microsoft.MixedReality.Toolkit.Unity.Extensions.2.4.0.unitypackage,
(iii) Microsoft.MixedReality.Toolkit.Unity.Examples.2.4.0.unitypackage, and
(iv) Microsoft.MixedReality.Toolkit.Unity.Tools.2.4.0.unitypackage.

Developer Mode Activation

To activate Developer Mode in Windows OS, go to *Settings/Update & Security*. In the category called *For developers*, activate the button ON. It has below a text saying *Install apps from any source, including loose files.* The steps to activate the Developer Mode in HoloLens1 are practically the same since it uses Windows Holographic OS.

For Android devices, the steps to activate Developer Mode are different for each smartphone. Nevertheless, generally is in the device's settings and *About this phone*. There,

tap multiple times on the text that either says *Built Number* or *System Version* until a *You are a developer* pops out. Then, a new tab should be available in settings called *Developer Options*. Enter there and activate *USB debugging* and *Install via USB*.

To build iOS apps, Unity has to be running on macOS since the app built uses Apple's Xcode. The steps to activate Developer Mode on iOS devices varies between versions. For example, for the latest iOS 16, go to *Settings*, select *Privacy & Security*, scroll down to Developer Mode, and activate it. Similarly, Apple has a developer account option where you can obtain a free personal license to install apps on your iOS devices.

FlowVisXR's Unity Project

Begin by creating a Unity project from Unity Hub using editor version 2019.4.40f1 and the 3D Core template, as shown in Figure 3.31. The project's name is *FlowVisXR* and is saved in an external hard drive with *Unity Projects* as the location's folder.

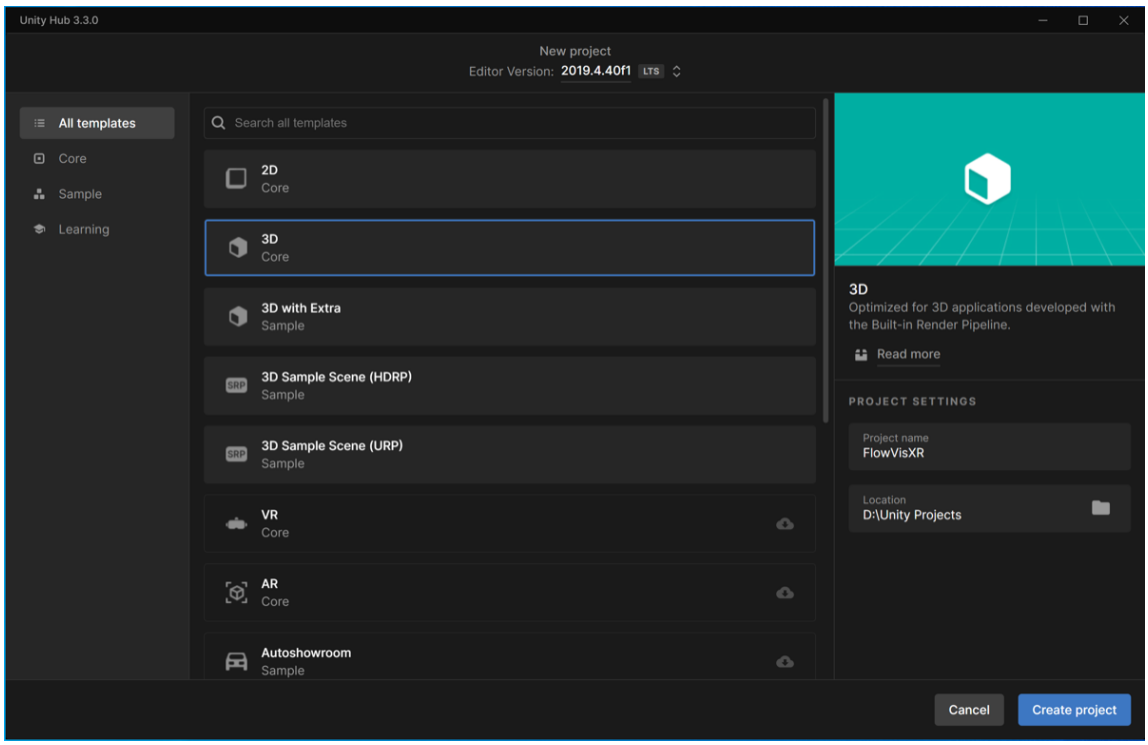


Figure 3.31: FlowVisXR app's Unity project creation.

After having the new project opened, import the MRTK Unity packages. Select *Import Package > Custom Package* from the *Assets* tab and begin importing the four packages

with the *Foundation* package, followed by the *Extensions* package. After importing all four into the Unity project, it should open a new window of the *MRTK Project Configurator*. Accept and apply all the suggested default settings. Next, from *Assets* panel, look up for *MRTK* > *Examples* > *Demos* > *HandTracking* > *Scenes* and double-click to *HandInteractionExamples.unity*. Accept the suggestions for importing *TMP Essentials*, which will open a new scene on Unity, similar to what is shown in Figure 3.32.

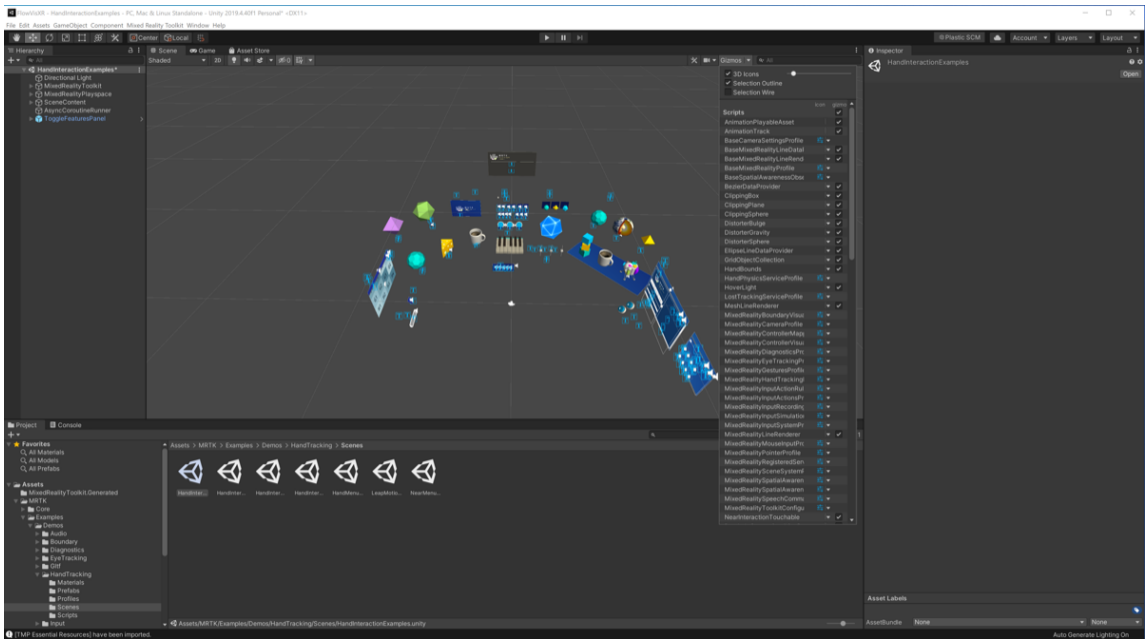


Figure 3.32: MRTK's HandTracking example scene.

Next, go to the *File* tab and open the *Build Settings* window. Because this guide is specifically for HoloLens1, the set-up to apply is: Target Device = HoloLens, Architecture = x86, Visual Studio Version = Visual Studio 2022, Build and Run = USB Device, and Build Configuration = Release. Finally, select *Universal Windows Platform* in the platform category and click on *Switch Platform*. Figure 3.33 shows these settings applied, and other suggestions from *MRTK Project Configurator* are also accepted. Now, open *Project Settings* windows from the *Player Settings* button or *Edit* tab. In the *Player Settings* category, fill the Company Name, Product Name, and Version. Select *Universal Windows Platform* and open the drop-down list of *XR Settings*. Change Depth Format to 16-bit depth and make sure *Virtual Reality Supported* is checked as shown in 3.34.

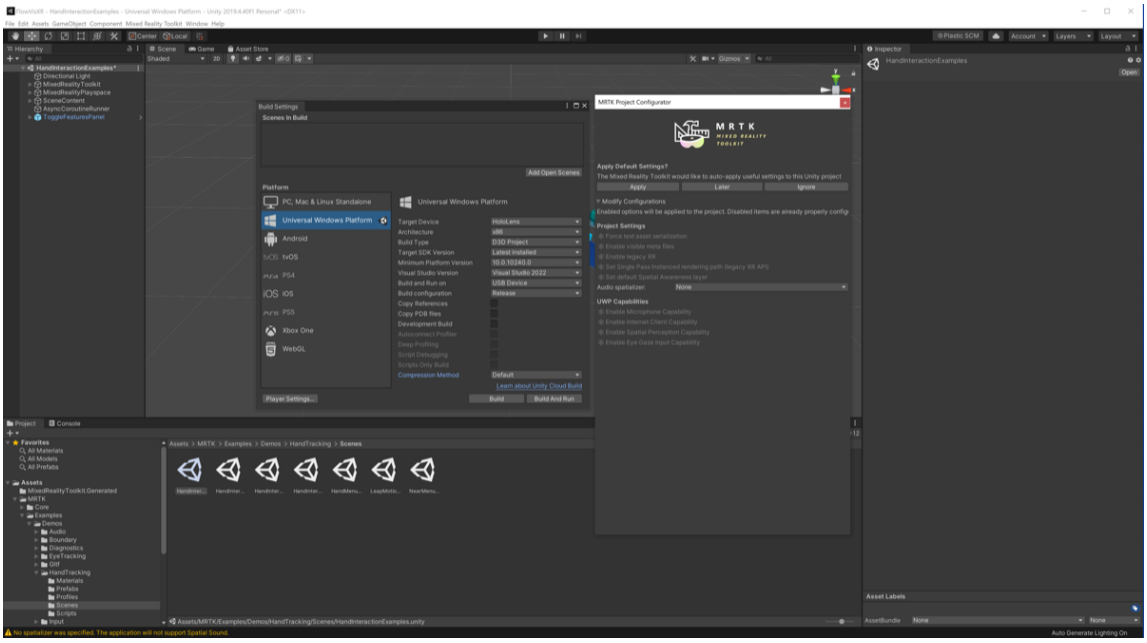


Figure 3.33: HoloLens with MRTK build settings.

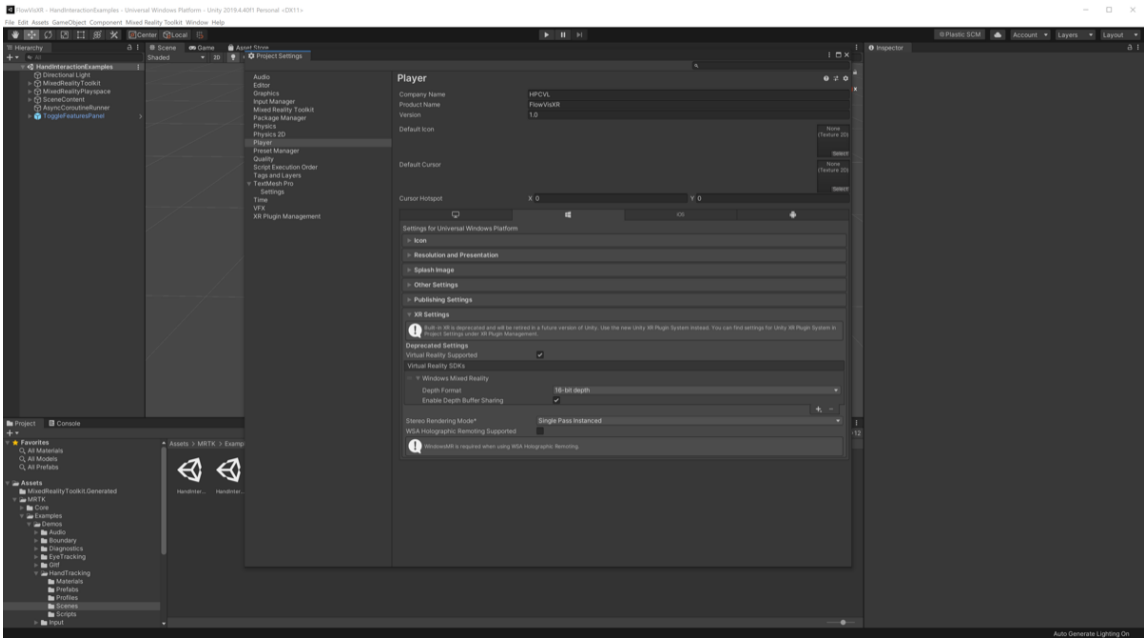


Figure 3.34: HoloLens with Player settings.

At this point, the Unity project is compatible with the HoloLens 1, and it is in a scene full of *gameObjects* in the *Hierarchy* panel, which are examples for getting to know what

components and scripts are needed in fundamental interactions of MRTK.

The methodology continues from the USDZ AR files to import them as virtual objects into the Unity project. To do this, we need to include an experimental package called *USD*. In Unity 2019.4*, the *USD* package is imported from the *Package Manager* located in the *Windows* tab. By clicking the plus (+) symbol in the *Package Manager* window, there is an option to *Add from git URL*, write the following *com.unity.formats.usd@3.0.0-exp.2* and import the package. There should be a new tab for *USD*; click on the option *Import as prefab*. The USD file to import is one of those USDZ files created as examples in the previous sections. Therefore, we edited the search to *All files (*.*)* and located the file named *PUVWT_1_565_326000.usdz*. After importing the USDZ file, it is automatically converted into a Unity prefab. Generally, the prefab is saved in the *Assets* directory; the minor issue is that the USDZ's color and texture are not correctly imported in the importing process. We need to re-import the USDZ files with unique options to fix this. Double-click the prefab in the *Assets* panel, which will cause edit mode on the prefab. In the *Inspector* panel with the prefab's properties, change the *Materials* option to *Import Preview Surface*, then click on *Refresh values from USD*. Figure 3.35 displays an example of re-importing USD files' material options. Now, it is time to create a Unity material. Right-click inside the *Assets* panel and choose *Create > Material*. Figure 3.36 shows the new material's settings. Leaving the name as *New Material*, change the Shader to *USD/StandardVertexColor*. The pending step is to drag and drop the photo as texture into the *Albedo (RGB)* box. Here, the image used is the same as on previous occasions, Figure 3.23. Adding the ColdHot.png file into Unity can be done by dragging and dropping it into the *Assets* panel. Having the New Material ready, drop it over the flow's mesh in the scene or over the *mesh1* gameObject in the *Hierarchy* panel. Finish the prefab edition by clicking on the back arrow of the *Hierarchy* panel.

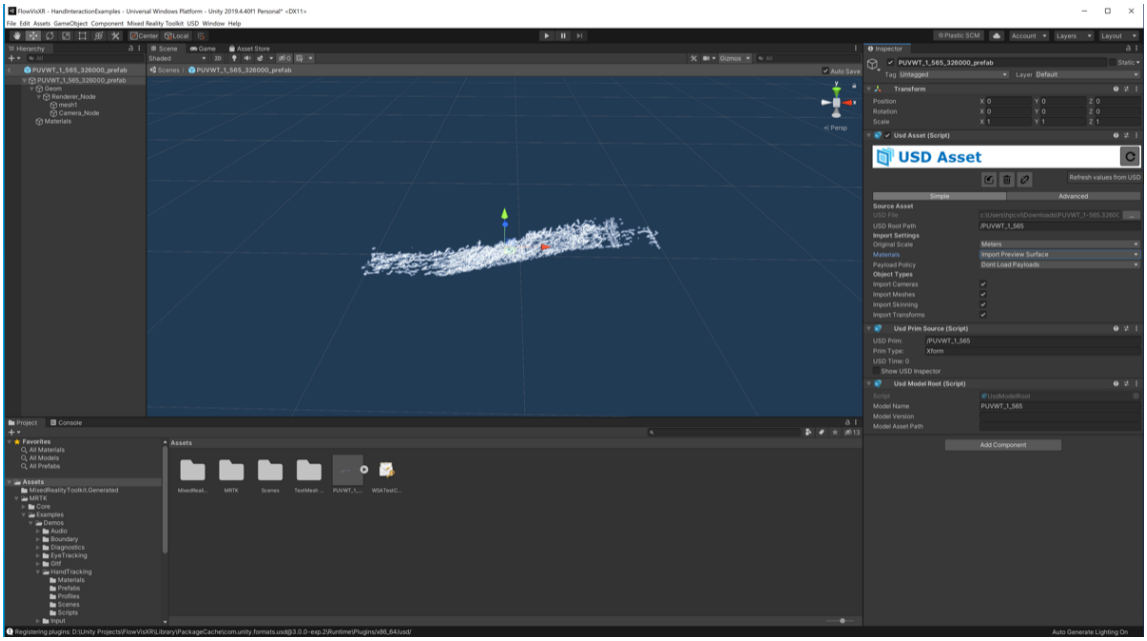


Figure 3.35: Example of re-importing USD files' material options.

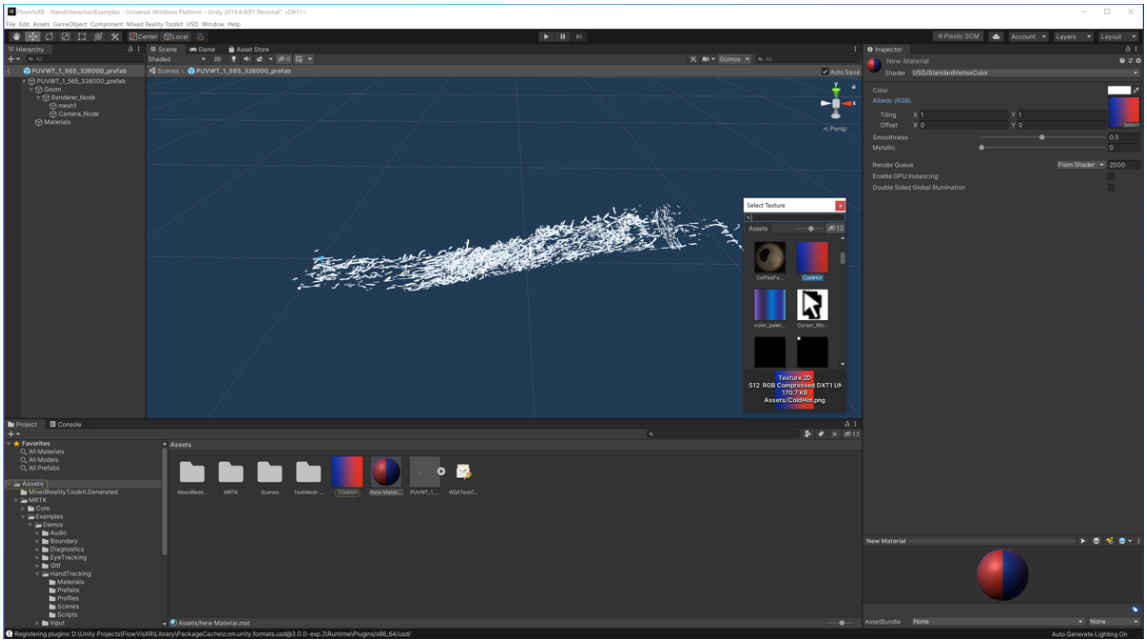


Figure 3.36: New Material's properties for the HoloLens USDZ files.

Now is the time to clean the Unity scene from all the *gameObjects* examples brought by MRTK and add our custom prefab with the AR manipulation ability. From the *Hierarchy*

panel, open the drop-list of *SceneContent*. Figure 3.37 shows all the *SceneContent*'s *gameObjects* children whom we will delete. After deleting them, add the USDZ prefab by dragging and dropping it over *SceneContent* to include it in the scene as a *gameObject* child. After the prefab is added to the scene, click on it to add a few components with the *Inspector* properties. The first component to add is *BoundingBox*. Drag and drop the custom prefab into *Target Object* because it is the *gameObject* to which we want to add the AR manipulation ability. Then, for the same component: (i) change the Behaviour Activation to Activate by pointer, (ii) Deactivate Show Wireframe, and (iii) Activate Proximity Effect Activate. The final three components to add are *Object Manipulator*, *NearIntectionGrabbable*, *CursorContextObjectManipulator*.

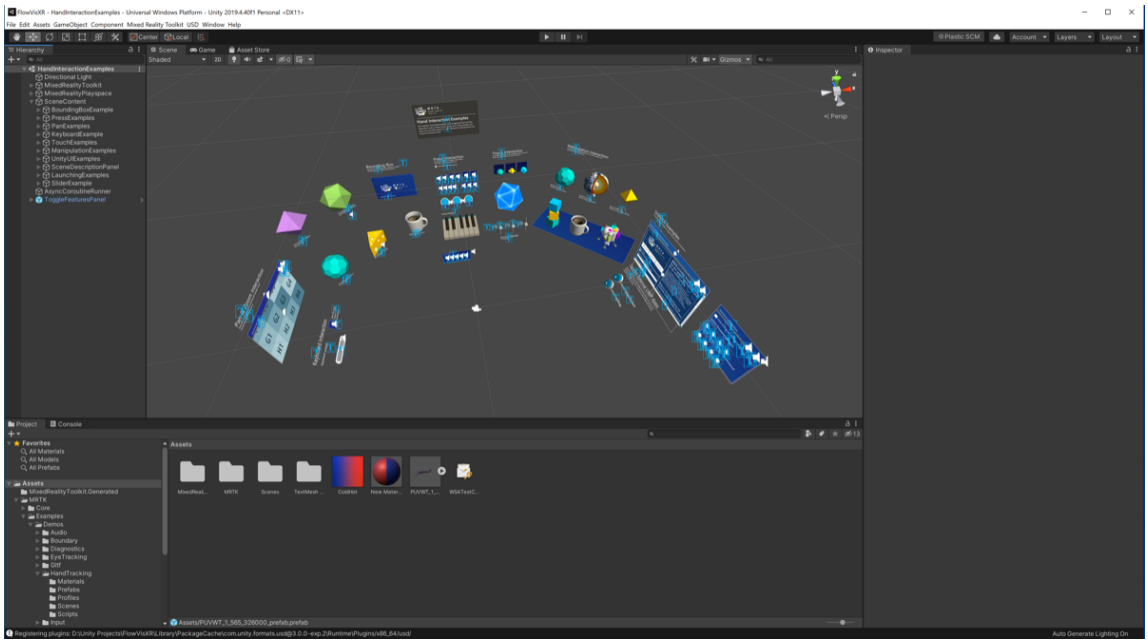


Figure 3.37: GameObjects to delete from SceneContent in FlowVisXR.

The FlowVisXR Unity project is completed, now is the time to build the app. Go to *File > Build Settings* and press the *Build* button, assuming that we have already set up the properties. Unity will ask for a directory address to save a few folders and a file with the extension of **.sln*. The **.sln* file is the instructions (or solution) to build the app written as a C++ Project. Figure 3.38 shows the Build Settings used for the HoloLens1 and the solution file.

Lastly, the **.sln* file should be open with Visual Studio 2022. It will understand that the **.sln* file is a Unity solution. It might suggest installing any missing tools to build the app properly. If that is the case, accept the suggestions, save and restart Unity, and

build the app's solution again in a clean directory. After the **.sln* file opened in Visual Studio, install the app by connecting the HoloLens1 via USB to the computer and press the start button with the settings of *Debug*, *x86*, and *Device*. Supposing it is the first time connecting the HoloLens to the computer, Visual Studio will ask to pair the device using a PIN. To get this code, in the HoloLens, go to *Settings/Update & Security/ For developers*. There is an option for pairing; if pressed, it will generate the PIN needed. Figure 3.39 shows Visual Studio asking for the pairing PIN and the correct settings before pressing play. Figure 3.40 presents a HoloLens1 screen capture while running the FlowVisXR app in front of the computer used to build the app. For more details with MRTK access: https://youtube.com/playlist?list=PLQMQNmwN3FvzWQ1Hyb4XRnVncvCmcU8YY.

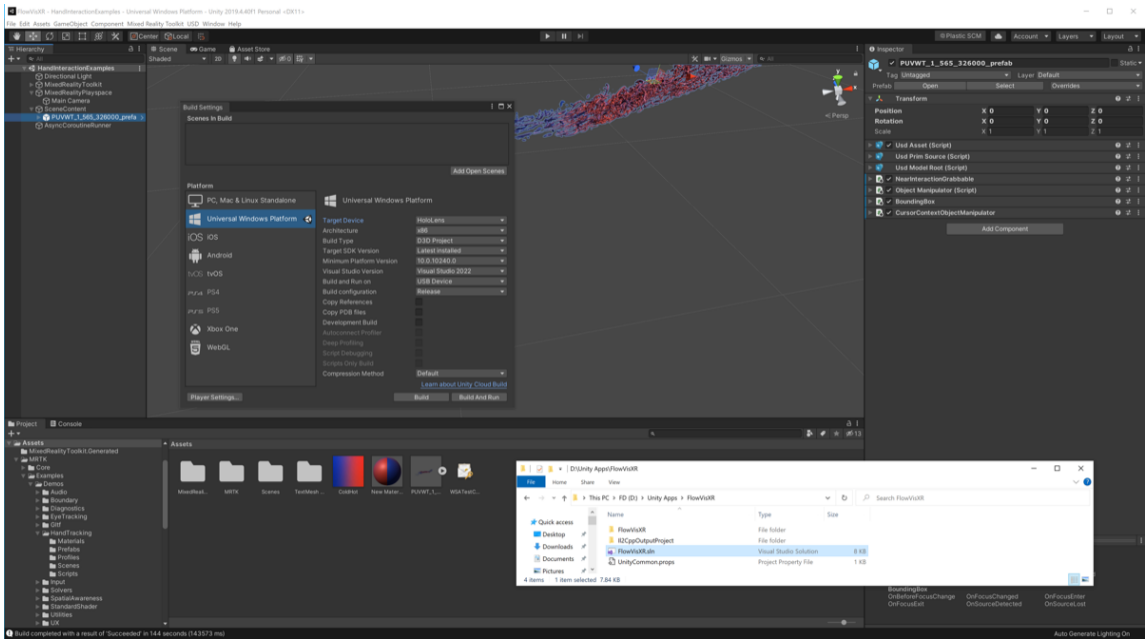


Figure 3.38: FlowVisXR Unity app build.

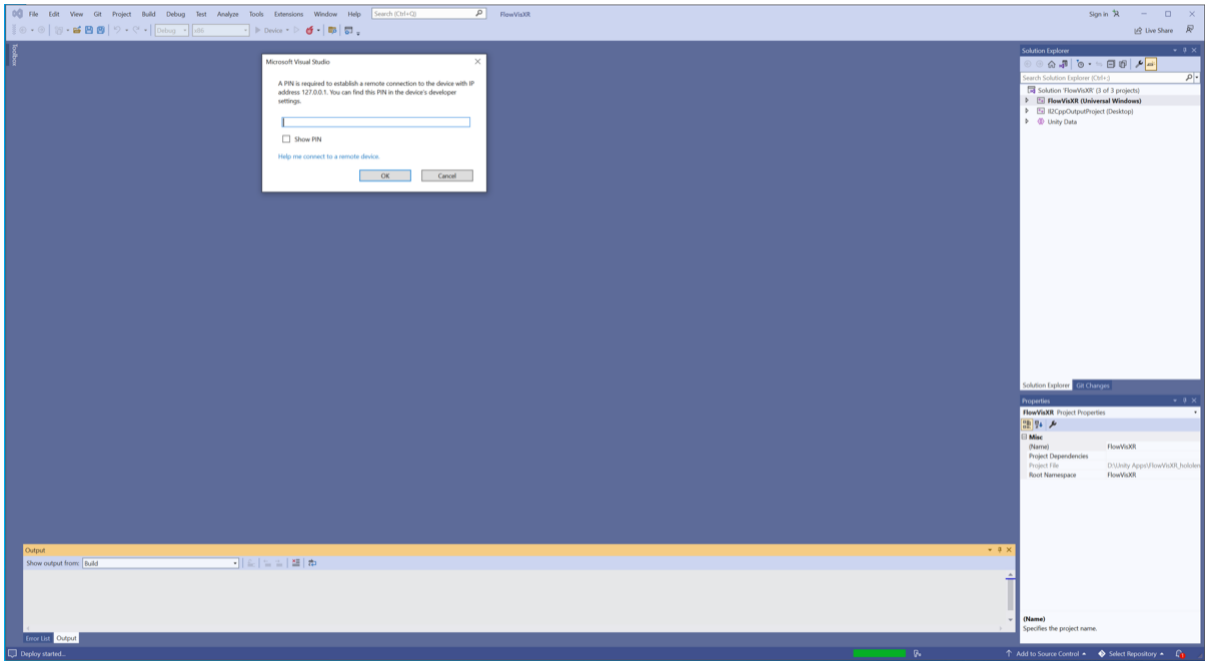


Figure 3.39: Visual Studio 2022 installing FlowVisXR on the HoloLens.

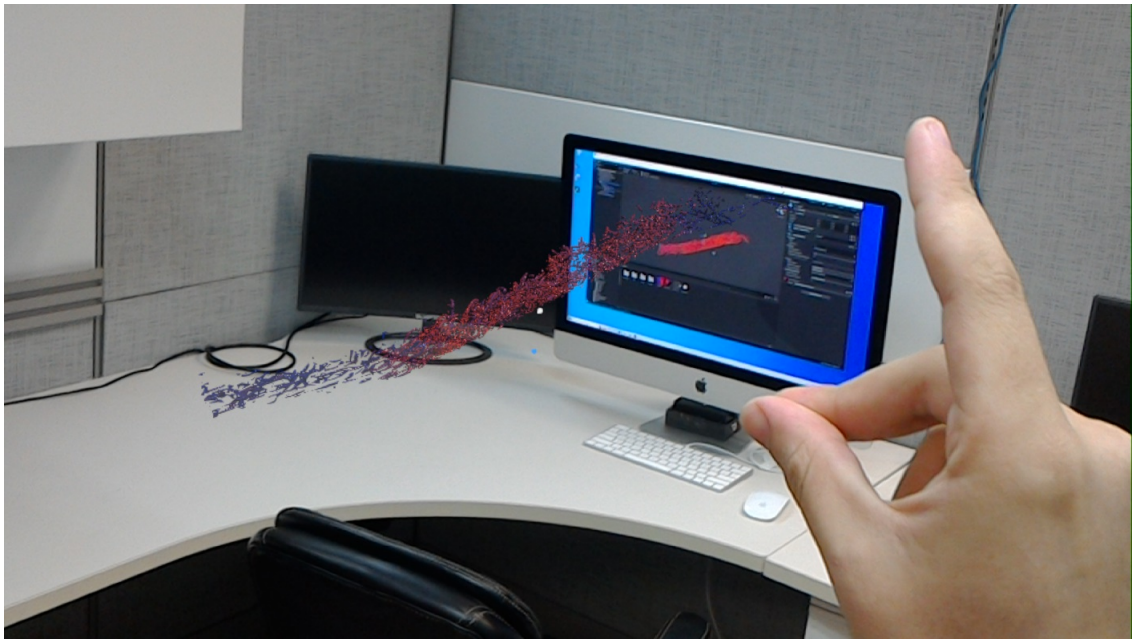


Figure 3.40: Screen capture of HoloLens running FlowVisXR.

3.3.2 Virtual Wind Tunnel for the HTC VIVE

Virtual Wind Tunnel (VWT) is an application created for VR visualization as part of the presented work. This app has been used with the HTC VIVE, as seen in Paeres et al. (2020) and Paeres et al. (2021).

VWT's Unity Project

Starting from Unity Hub, the first step is installing Unity 2021.3.11f1 without needing additional modules (assuming you already have Visual Studio) and creating a Unity project with a 3D (core) template. Given that the present methodology will cause a high computational load with the files transfers and render as part of the virtual environment, the Unity project is recommended to be saved locally on the computer for optimal hard drive and graphic board performance.

For connectivity between VR devices (e.g., HTC VIVE) and Unity, a potent tool is the SteamVR from the Steam platform. Steam software can be downloaded from the official web page: http://store.steampowered.com/about/. After having the Steam platform installed and opened, lookup for the SteamVR tool and install it, as is shown in 3.41.

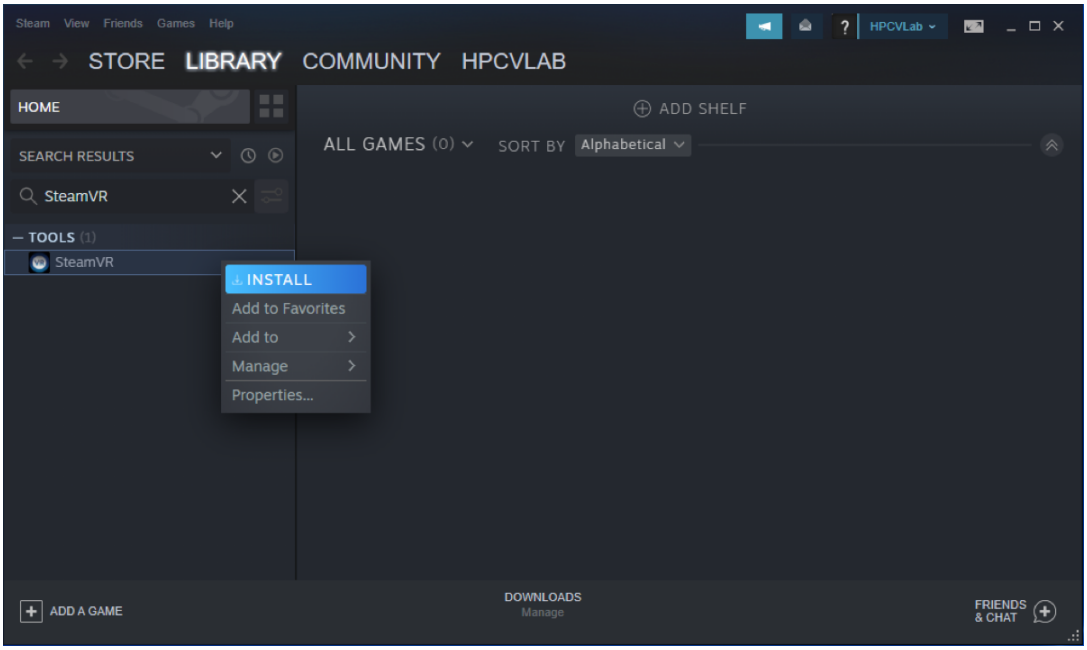


Figure 3.41: Installing SteamVR tool on the Steam platform.

SteamVR has a plugin registered as a commercial asset in Unity. Sign in with the Unity

account on the following web page: http://assetstore.unity.com, and purchase the SteamVR Plugin asset, which is free. Connect the HTC VIVE hardware to the computer, leave the SteamVR tool running and then go back to the Unity project. From the *Window* tab, open the *Package Manager*. In the *Package Manager* window, edit the packages list filter to *My Assets* and import everything from the SteamVR Plugin. Figure 3.42 shows the SteamVR package being imported and the SteamVR running on Steam. After importing all and because the HTC VIVE is connected and recognized by SteamVR, Unity will pop up a new window with the title *Valve...Settings Window*, suggesting some changes which, after accepting them, restart the Unity editor.

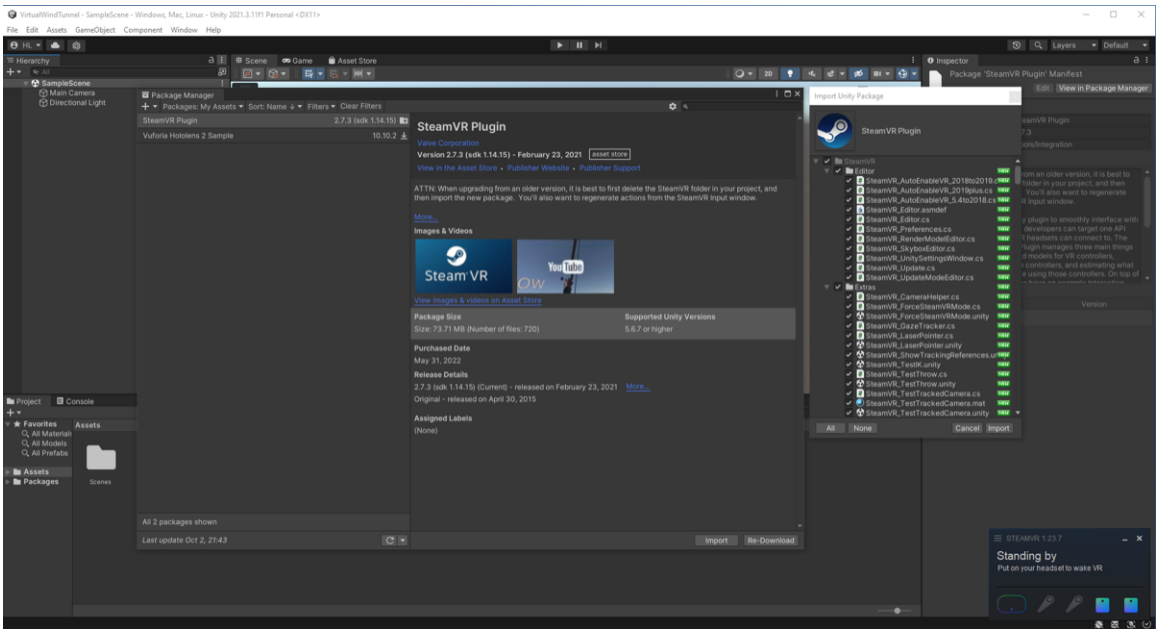


Figure 3.42: SteamVR Plugin imported into Unity project.

From the *Hierarchy* panel, delete the *Main Camera* gameObject. Then in the *Project* panel, look up the *Player* prefab and add it to the scene by using drag and drop over the *Hierarchy* panel. Figure 3.43 shows the Player prefab added in the scene. Then from the Edit tab, open *Project Settings*, and in the *XR Plug-in Management* category, activate *OpenVR Loader* as the provider, following the example of Figure 3.44.

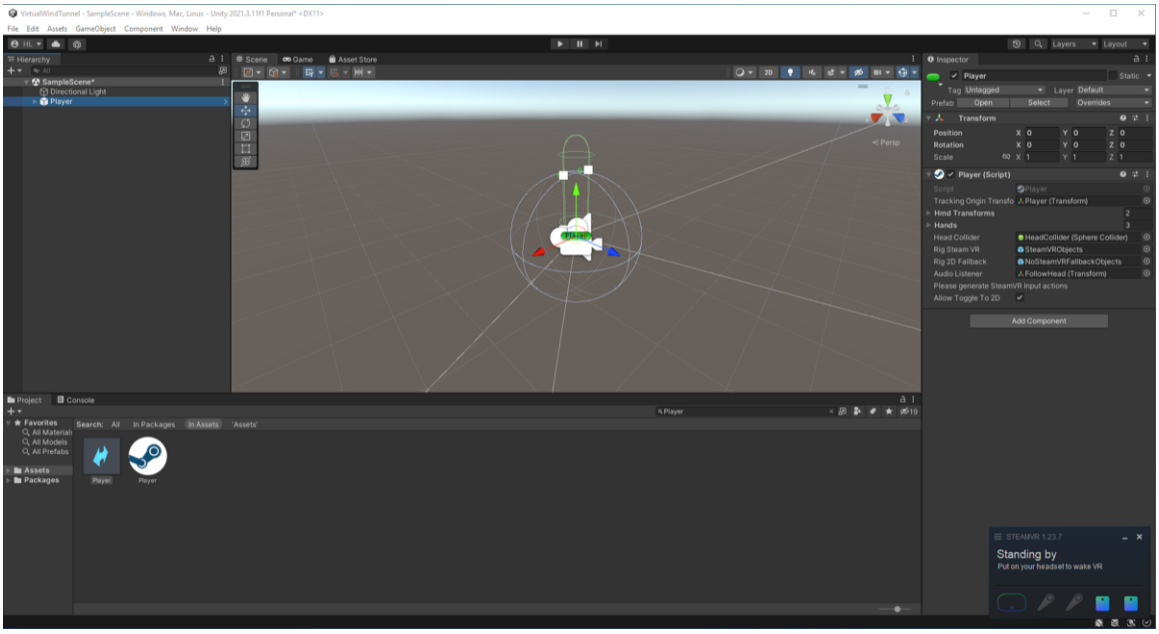


Figure 3.43: Main Camera gameObject deleted while the Player prefab is included.

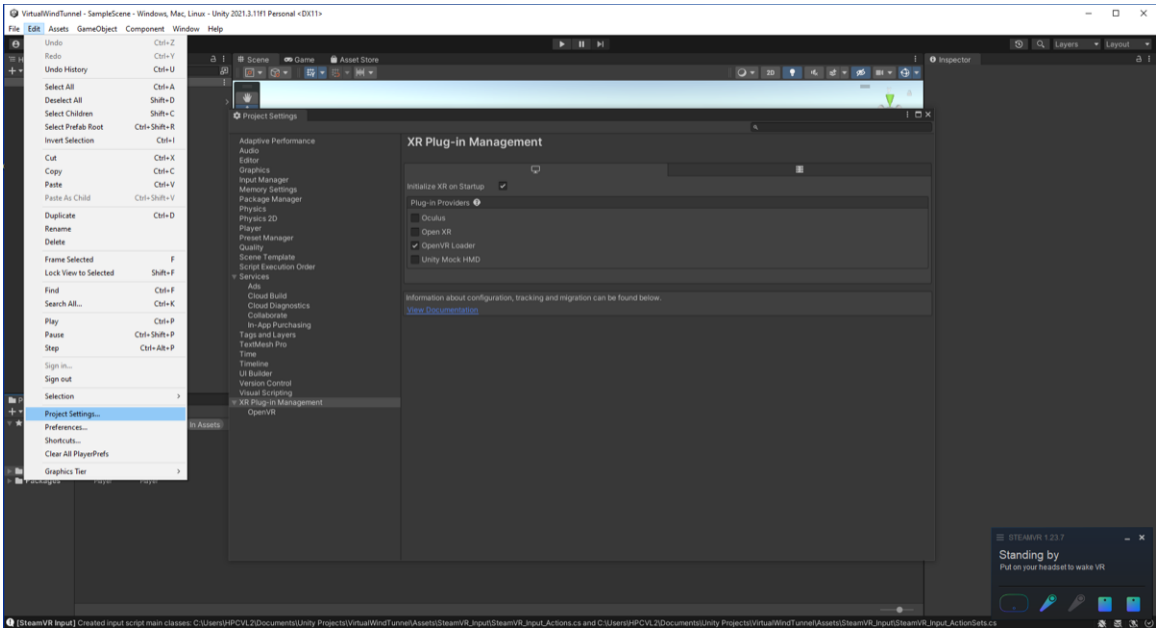


Figure 3.44: OpenVR Loader provider activation in XR Plug-in Management.

After pressing play on Unity, it will recognize the HTC VIVE device, but it will say that have missed an input profile. Accept and generate a new profile from the default example files that Unity will offer in the SteamVR Input window. After generating the

input profile and saving the scene, Unity will look similar to Figure 3.45. There is no need to edit any binding.

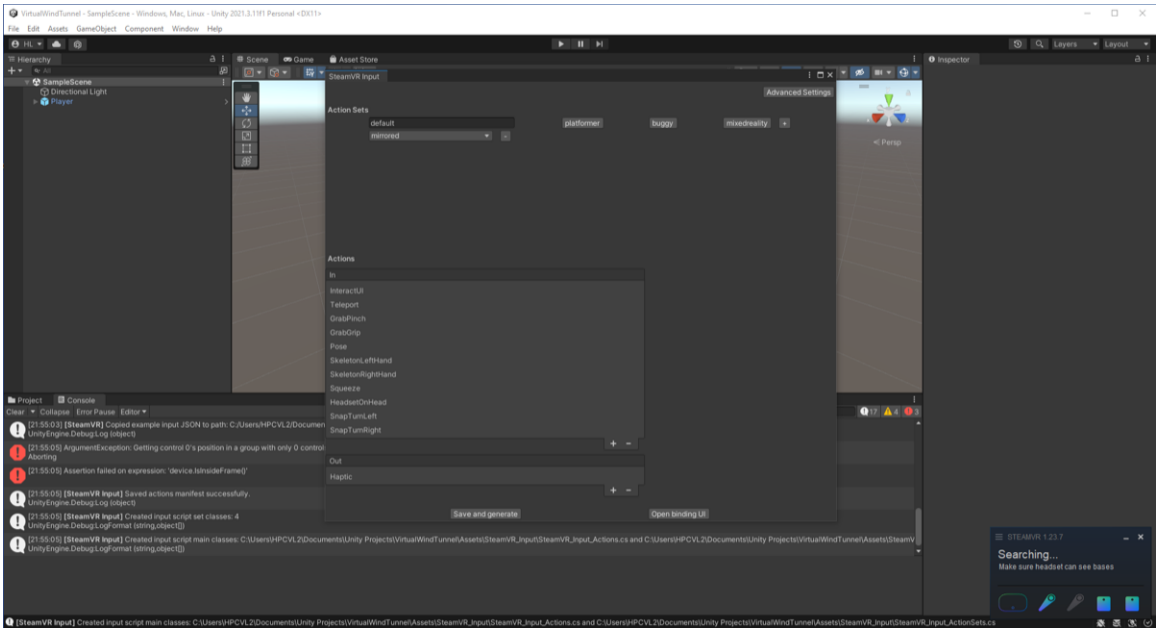


Figure 3.45: SteamVR input profile generation.

The Unity project is already fully set up for VR experiences with the HTC VIVE. However, the present guide uses the Virtual Wind Tunnel as a dynamic flow visualization with VR. Thus, as part of VR technology, we must build a virtual environment where the user will immerse himself. The virtual environment could be a simple building with walls and a floor; on the world wide web, there are many tutorials on creating rooms, facilities, etc. However, in this methodology, we are going to import a virtual warehouse that has been previously created in Unity. It will not be shown how the author built the virtual warehouse since it is out of the work's scope and is irrelevant to flow visualization. The virtual warehouse is imported as a custom Unity package with the name of *VWT_warehouse.unitypackage*. This package was created using a specific rendering package for a realistic appearance. Therefore, from the Package Manager window, change the list's filter to *Unity Registry* and import the package named *High Definition RP* (version 12.1.7). After installing a new Render Pipeline (RP), generally, there are a lot of materials and properties adjustments to make. Luckily, HDRP Wizard can apply most of all adjustments by simply pressing *Fix All*, as shown in Figure 3.46. After applying all the suggested fixes, there are a few more to do manually. First, go to *Edit* > *Rendering* > *Materials* and select C*Convert All Built-in Materials to HDRP*. Figure 3.47 display the VTW with all built-in materials converted to HDRP and from where you can do it.

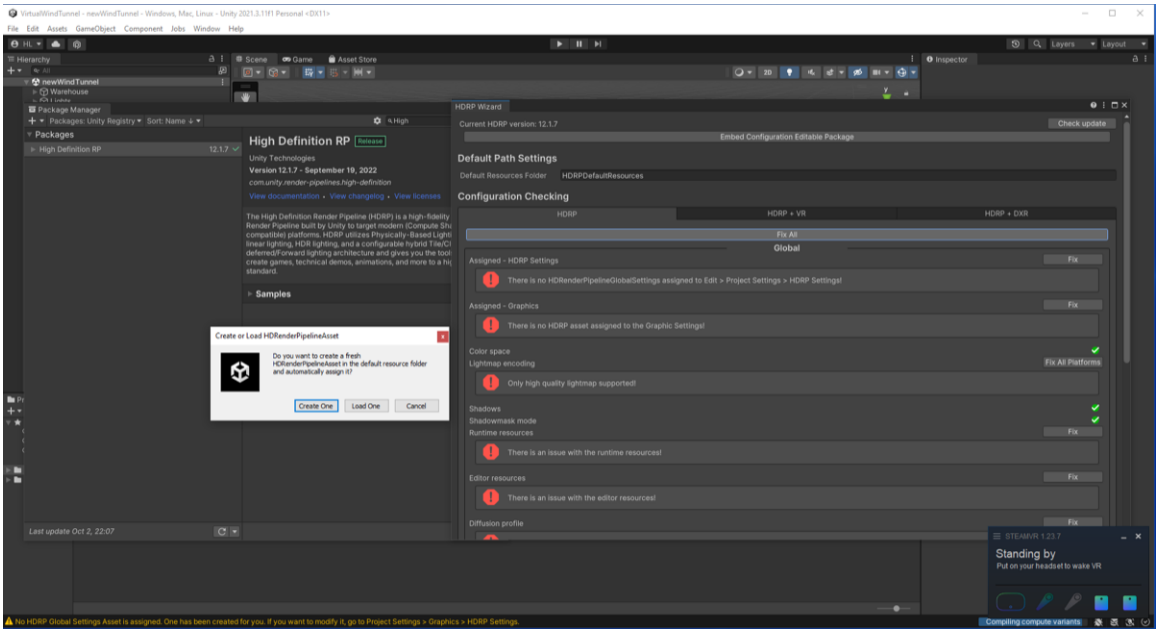


Figure 3.46: High Definition RP package imported to the Unity project.

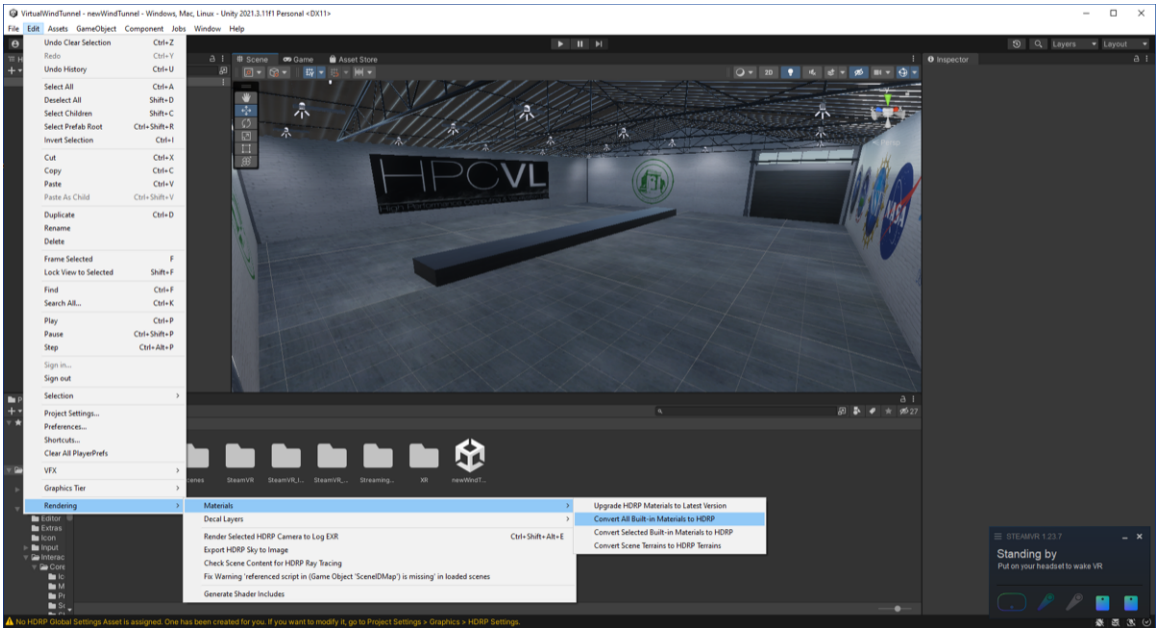


Figure 3.47: All built-in materials converted to HDRP.

The SteamVR Plugin package has materials incompatible with HDRP. You can identify the damaged materials with a hot pink color or showing completely invisible. Figure 3.48 shows a collection of materials that appears invisible and are located in *Assets*

> *SteamVR* > *InteractionSystem* > *Teleport* > *Materials*. These are being fixed by first changing the materials' *Shader* to *Standard*. Like in the previous image, select the materials one by one while holding the *Cntrl* key down, then, in the *Inspector* panel, change the materials' *Shader* to *Standard*. Figure 3.49 shows the materials with a hot pink color. Without deselecting the materials, go to *Edit* > *Rendering* > *Materials* and select *Convert Selected Built-in Materials to HDRP*. Apply the same fixing process to the materials that have the invisible resemblance or hot pink color located in *Assets* > *SteamVR* > *InteractionSystem* > *Core* > *Materials*. Ignore those materials with URP (Ultra Render Pipeline) in their name since these are for a different Render Pipeline and are not used in this project.

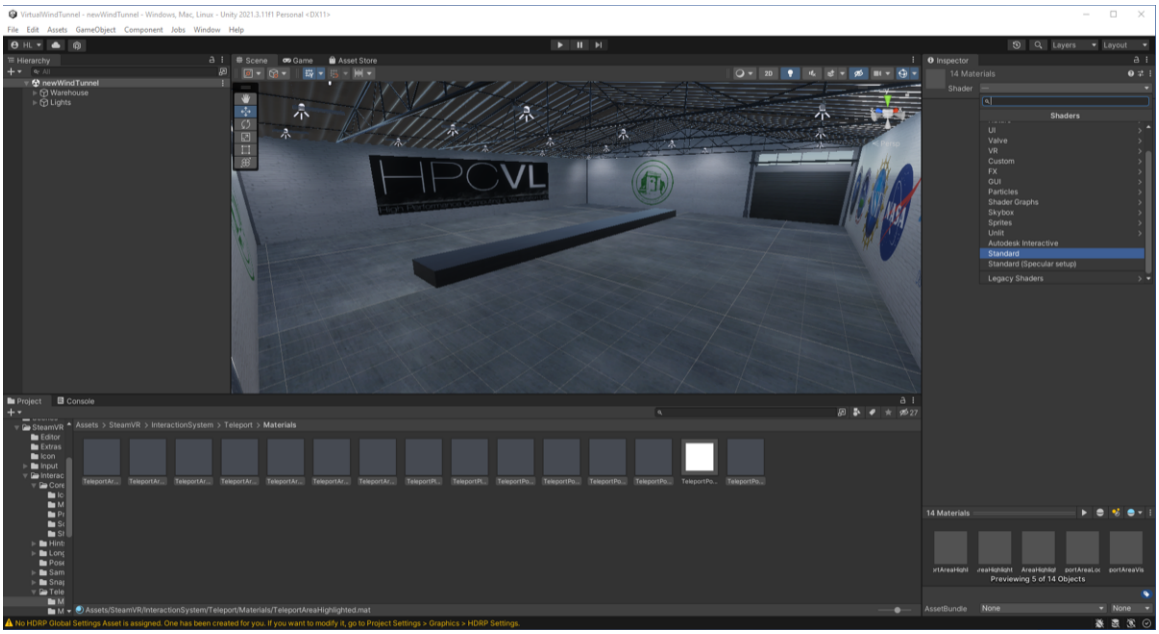


Figure 3.48: SteamVR materials resembling as invisible due to incompatibility with HDRP.

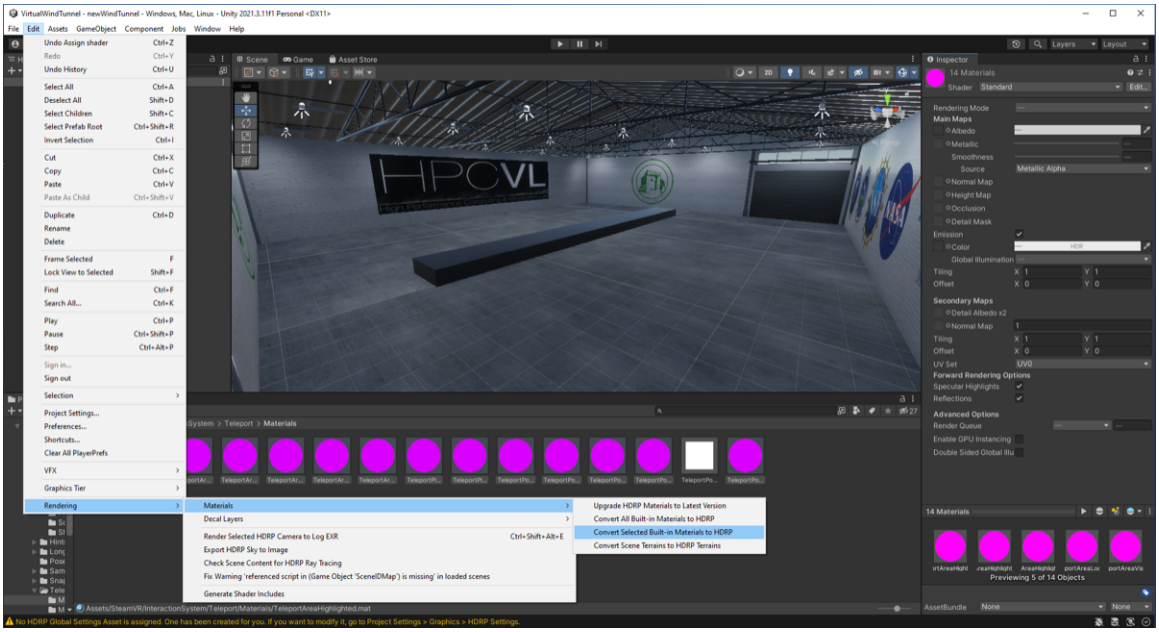


Figure 3.49: Converting the SteamVR materials to HDRP.

Because the warehouse's scene is different from the one used in the begging, again, lookup for the *Player* prefab in the *Project* panel. Drag and drop it in the *Hierarchy* panel. Figure 3.50 displays the VWT with the *Player* gameObject in the scene and also shows the values applied in the Transform component of the *Inspector* panel. The next step is to create the ability to teleport for the VR user and the floor where the command can be used. From the *Project* panel, lookup for *Teleporting* prefab and add it to the *Hierarchy* panel. After selecting the *Teleporting* gameObject from the Hierarchy panel, identify the line of *Area Visible Material* in the *Teleport (script)* component. To this line, change the material to *HoverHighlight*, located in *Assets* > *SteamVR* > *InteractionSystem* > *Core* > *Materials* from the *Assets* panel. Figure 3.51 shows the *HoverHighlight* material used for the *Area Visible Material*. Note material appears to be invisible, but in a few steps is going to be fixed.

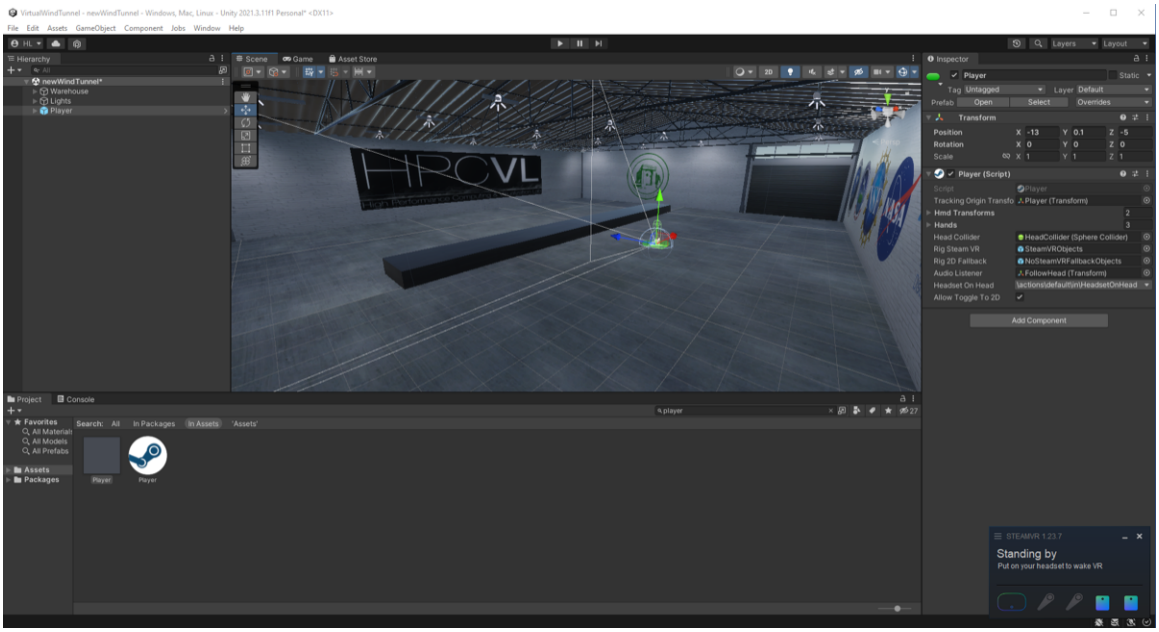


Figure 3.50: Player prefab added to the VWT scene.

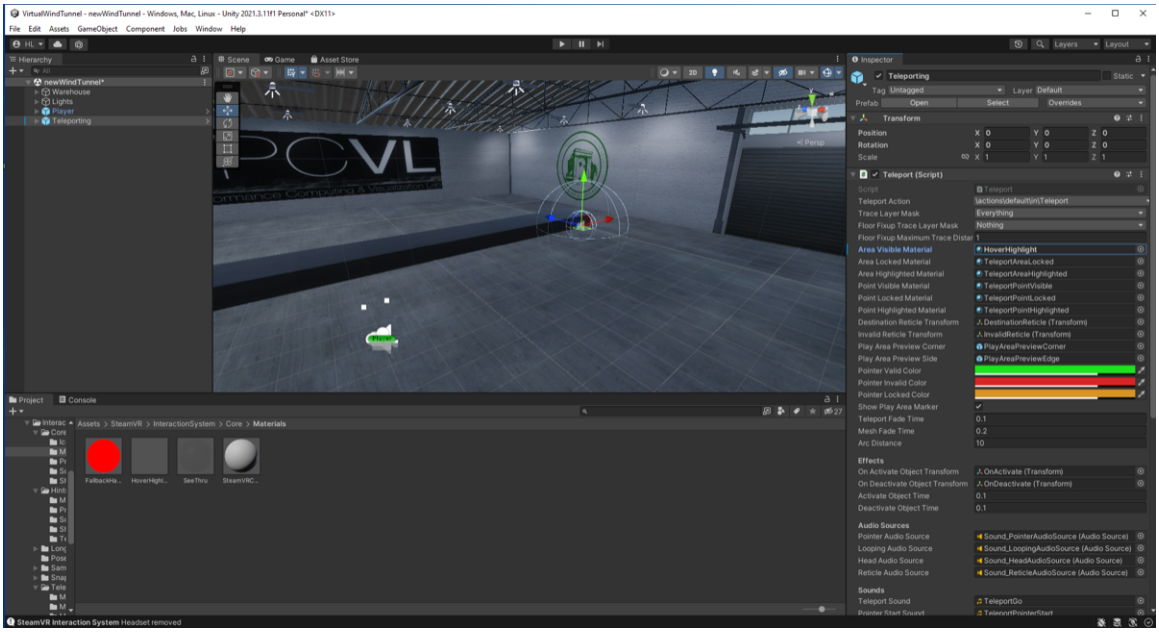


Figure 3.51: Changing the Area Visible Material in the Teleport script component.

To create the floor where the teleport will be used, right-click on the *Hierarchy* panel and select *3D object > Plane*. Rename it to *Teleporting Area*, and in his Transform values set for the position: x=0, y=0, z=0, and scale: x=100, y=1, z=100, making the floor

the same shape and location as the warehouse's floor. To the *Teleporting Area* gameObject, add a new component called *Teleporting Area*, which is a script. Continuing in the *Inspector* panel, identify the component called *Hover Highlight (Material)*, and in the category of *Surface Options*, set the following settings: (i) Surface Type = Transparent, (ii) Double-Sided GI = Off, (iii) Material Type= Translucent, and (iv) all boxed unchecked. Also, for the *Surface Inputs* category, set the Base Maps color to R=255, G=255, B=255, and A=0. Figure 3.52 can be used as an example of the *Hover Highlight (Material)*.

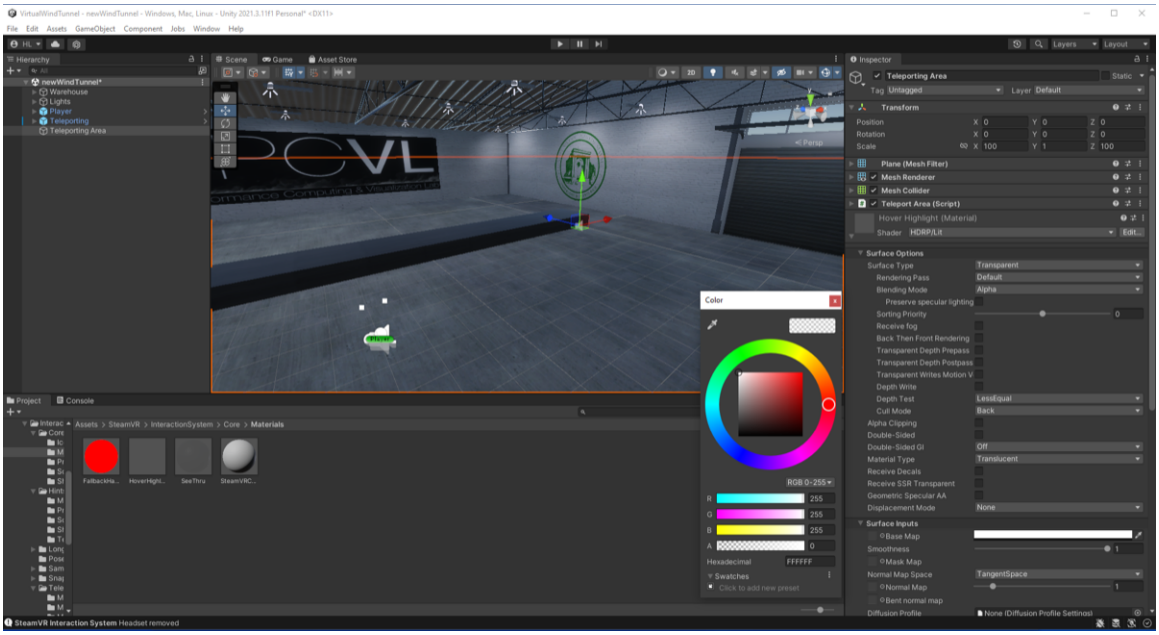


Figure 3.52: Hover Highlight Material setup.

The virtual environment has been completed to this point, including the teleport ability within the virtual warehouse. Nevertheless, the goal here is to transform the virtual warehouse into a virtual wind tunnel. We need to find a way to automate the import process (for the high number of USDZ per time-frames) and the flow animation using the stop-motion technique. To use USDZ files in this Unity project, we will have to use the same Unity packages used for the FlowVisXR app, called *USD*. Open the *Project Manager* window, click on the plus (+) symbol, and import the asset by name: *com.unity.formats.usd*. After importing the *USD* package, right-click on the *Assets* panel to create a folder with the name *Resources*. Within the *Resources* folder, create two new folders with names: (i) *ToImport* and (ii) *Prefabs*. In the *ToImport* folder, drag and drop all the USDZ files you want to import into the Unity project. The *Prefabs* folder is where all the USDZ files converted into prefabs will be stored. You might think that the USDZ files' import

requires many manual steps. Well, for the VWT, two scripts were made to automate the process and make it more convenient, which can easily be re-used or adapted for other Unity Projects. The scripts are named *usdzToPrefab.cs* and *prefabMeshAnimator.cs* and can be found in Appendix E.

Drag and drop the files into the Unity project to add the automation scripts. However, to use these in Unity, the fundamental instructions are: (i) In the *Hierarchy* panel, create an Empty gameObject and name it *Importer*. To *Importer*, add the script *usdzToPrefab* as a component. If the *Importer* gameObject is activated with the correct settings, the automated importing of the USDZ files will start after pressing play in Unity. (ii) Similarly, in the *Hierarchy* panel, create an Empty gameObject with the name *Flow1*. To *Flow1*, add the script *prefabMeshAnimator* as a component. If the *Flow1* gameObject is activated with the correct setting, the animation will start after pressing play in Unity. Never run Unity with both scripts on the *Importer* and *Flow1* gameObjects activated simultaneously. The names of the gameObjects are not required to be precisely the ones used in this guide. For the *usdzToPrefab* script, the addresses are relative to the Assets directory; meanwhile, for the *prefabMeshAnimator*, the addresses are relative to the Resources directory.

In the example of this guide, the first time-frame file is named *PUVWT_1-565.266000.usdz*, and the last one is *PUVWT_1-565.267820.usdz*. Given that we have every ten time-frames, the total number of files is 183. Note that the first 14 characters in their name never change; this combination is called the *General File Name* and is used in both automation scripts. In this example, the counter goes from 600 to 782, meaning the *Initial Frame* is actually 600. Moreover, because we know both addresses of the *ToImport* and *Prefabs* folders, we can now fulfill the configuration for both scripts. Figure 3.53 shows the configuration of the *usdzToPrefab* script as an example. The last thing to create is the material used for the *prefabMeshAnimator* script. As it was done for the FlowVisXR project, right-click on the *Assets* panel and select *Create > Material*. To this new material, apply the Shader named *HDRP > 3DSMaxPhysicalMaterial* since we are now using HDRP. Import to Unity the same *ColdHot.png* (see Fig. 3.23) into the *Assets* panel. Add the image file into the *BaseColorMap* box, as shown in Figure 3.54. Figure 3.55 presents the settings used for the *prefabMeshAnimator* script and is a screenshot of the Virtual Wind Tunnel running perfectly with its flow animation. For more details with SteamVR access: https://youtu.be/5C6zr4Q5AlA.

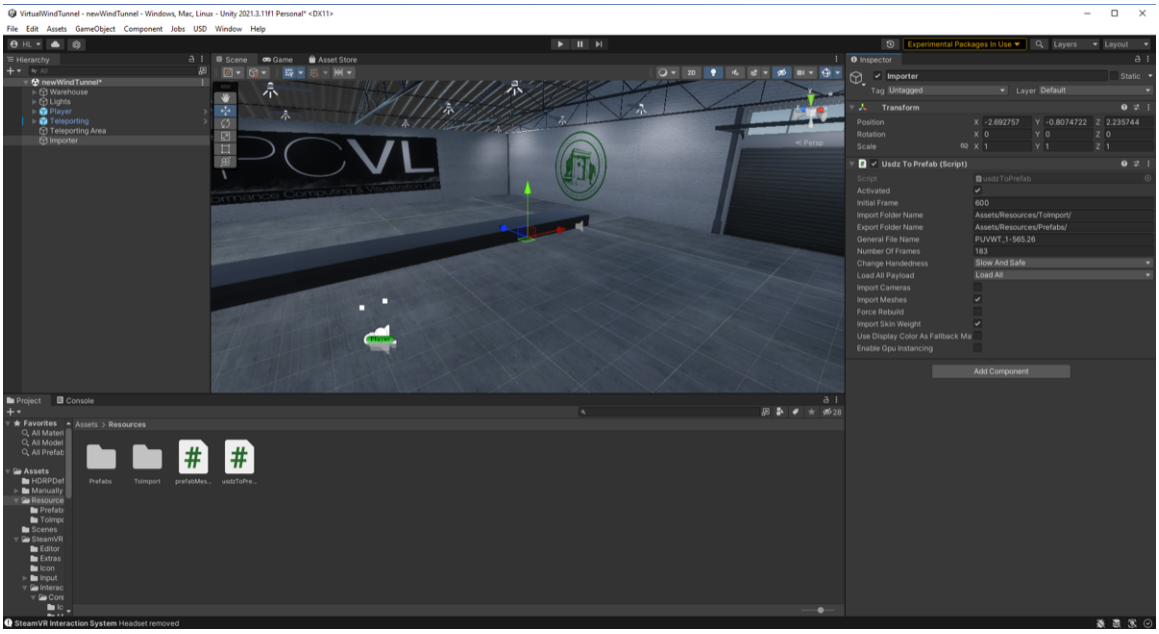


Figure 3.53: Example of the configuration for the usdzToPrefab script.

Figure 3.54: Creation of the material used for the VWT flow animation.

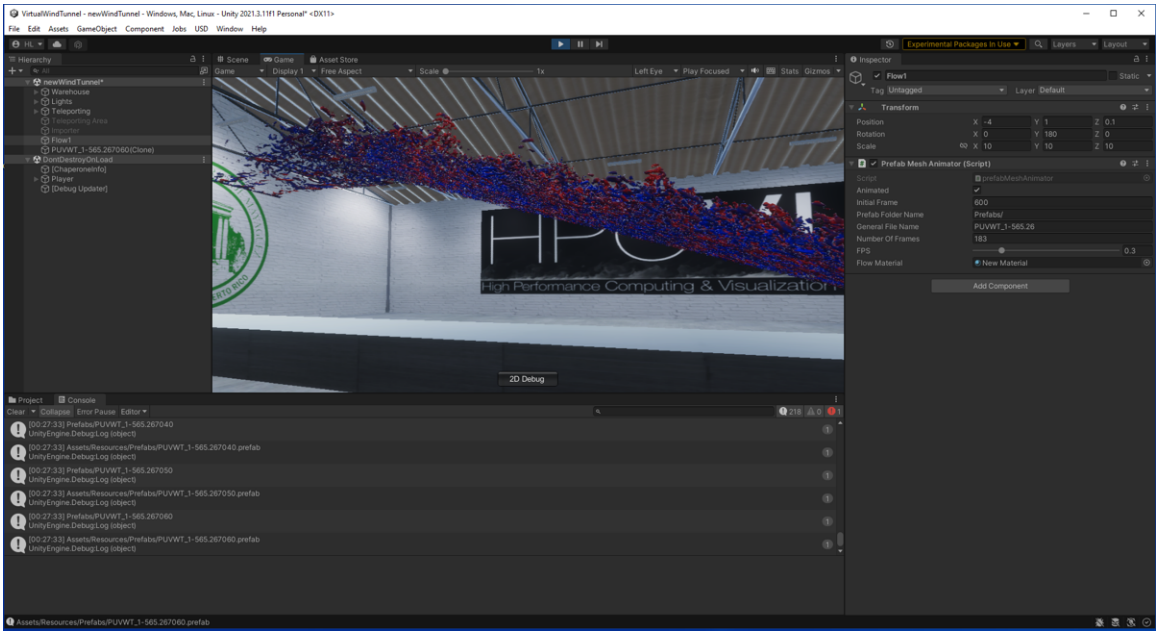
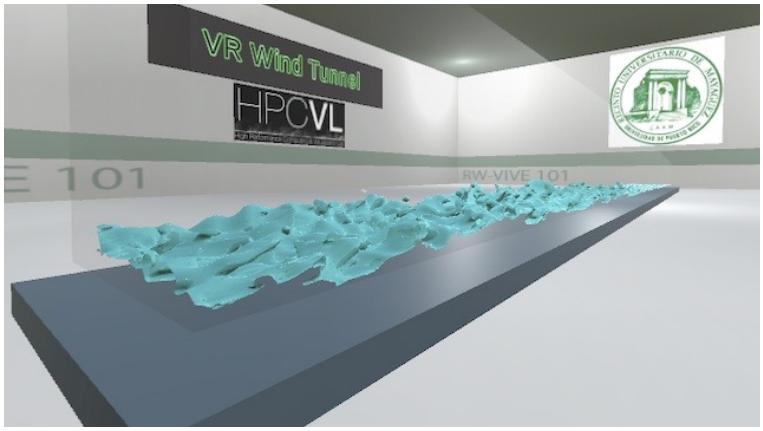


Figure 3.55: Virtual Wind Tunnel running the flow animation.

3.4 Conclusions of flow visualization with XR

It is concluded that the presented visualization methods have achieved the objectives and goals set for the current thesis. Figure 3.56 shows two versions of the Virtual Wind Tunnel; the variations are in the building designs or decorations applied, in other words, aesthetic differences. Figure 3.56(a) also shows the VWT performing visualization of the iso-surface of instantaneous streamwise velocity extracted from our lab's DNS database (HPCV 2018). The algorithms developed in this research have been used for other works, meaning the Broad Impact expected from the present work has been accomplished. XR technology is amazingly growing in applications and methods, which is hard to track. For example, it was not shown as part of the guide, but the FlowVisXR app can recognize images designated as triggers for the emergence of virtual objects from a database into the device's eyesight using additional libraries. This technologies' evolution is so accelerated that a weakness has been identified. The constant tools' creations and updates are now frequently causing compatibility issues, leading to the withdrawal of the development and support of potentially powerful tools.



(a)

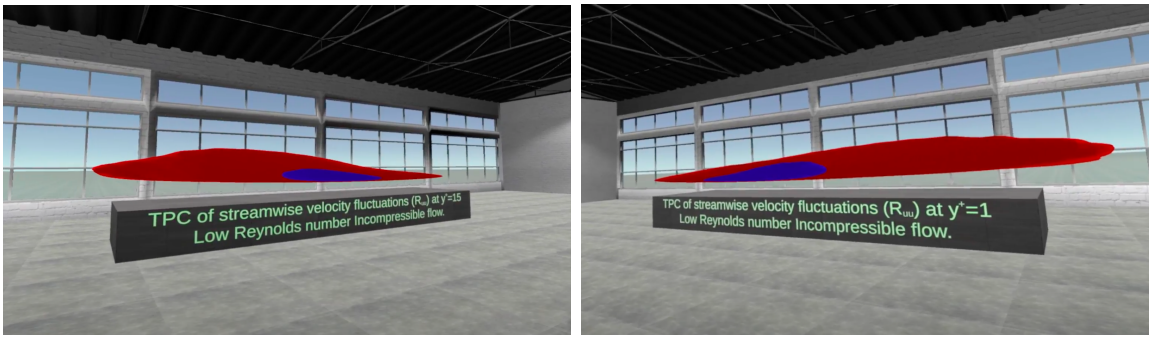
(b)

Figure 3.56: Virtual Wind Tunnel variations: (a) Simple room design of VWT; (b) Realistic warehouse design of VWT.

Figure 3.57 show examples of Two-Point Correlation (TPC) coherent structures for an incompressible flow at relatively low Reynolds numbers. These iso-surfaces of TPCs were previously obtained via spatial correlations of streamwise velocity fluctuations, u', at $y^+ = 15$ (buffer layer) and $y^+ = 1$ (linear viscous layer), respectively; all the databases from Paeres et al. (2020). The display is performed over a different "virtual" room inside the "Virtual Wind Tunnel facility" called the Gallery of Coherent Structures.

Figure 3.58 displays a supersonic flow with adiabatic wall condition corresponding to the work of Paeres et al. (2021). The figure shows a Microsoft HoloLens and FlowVisXR user rotating the temperature fluctuations field target $\pm 0.4 T_\infty$. FlowVisXR app is currently supported for iOS devices and Microsoft HoloLens. Android devices have not been tested

yet, but to the author's knowledge, building the app for Android is straightforward. Demonstrations can be watched at the official APS Gallery of Fluid Motion site. Videos DOIs (Digital Object Identifier) are: (i) https://doi.org/10.1103/APS.DFD.2020.GFM.V0045 and (ii) https://doi.org/10.1103/APS.DFD.2021.GFM.V0028.



(a) (b)

Figure 3.57: Streamwise velocity fluctuations at (a) $y^+ = 15$, (b) $y^+ = 1$.

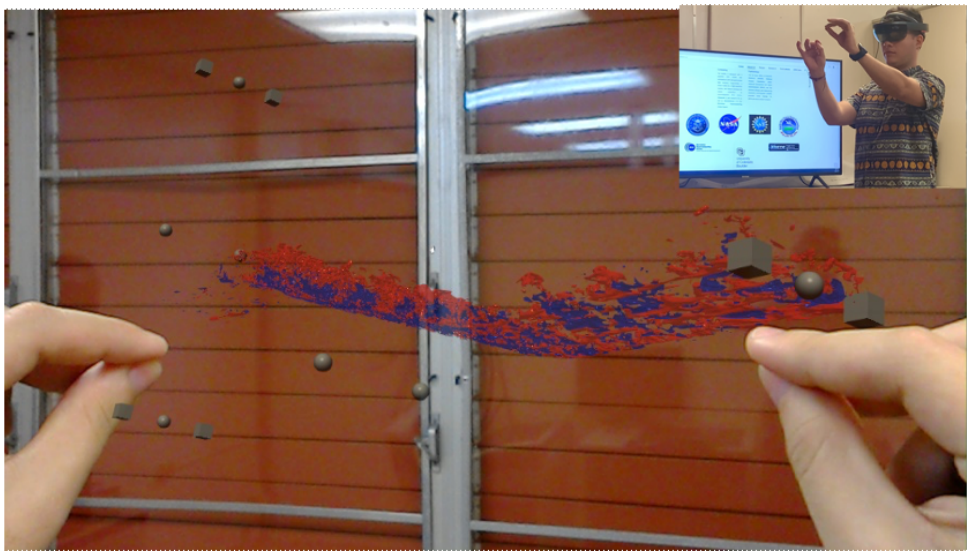


Figure 3.58: Microsoft HoloLens MR visualization example.

Chapter 4
Final Remarks and Future Work

The present thesis was based upon CFD simulations utilizing the RANS approach with OpenFOAM software, numerical results post-processing via scripting and scientific visualization toolkit (i.e., Python and ParaView), to finally elucidate through the analysis and XR, the details behind momentum and passive scalar transport phenomena during turbulent boundary layer separation. An extensive DNS database from our team was used as validation data for the CFD cases' initialization. In the post-processing phase, a new scheme to calculate boundary layer parameters was proposed as a hypothesis. Furthermore, experimental data from Baskaran, Smits, and Joubert (1987) was used as corroboration evidence for the hypothesis.

Chapter §1 gave a broad theoretical preparation for the reader. The chapter communicated fundamental background regarding **(i)** fluid dynamics, **(ii)** computational fluid simulations, **(iii)** flow separation phenomena, **(iv)** passive scalar transport (e.g., heat and pollutants), and **(v)** visualization of objects through XR (i.e., Virtual, Mixed and Augmented Reality). From the resistance's notion in history to the Reynolds-Averaged Navier-Stokes (RANS) equations' derivation was covered, including the boundary layer detachment phenomenon. After explaining DNS, LES, and RANS methods, the most common turbulent models were explained. The reader acquired nomenclature preparation and enough parameters definitions needed to understand the flow's analysis performed. Finally, Chapter §1 provided a sufficient background of scientific visualization with Extended Reality (XR) to educate the reader in identifying each regime of the Reality Continuum since XR is expected to take over the daily basis related to any technology.

Chapter §2 discussed CFD post-processing via RANS and analyzed several aspects of the selected scenario: an incompressible turbulent flow over a curved hill. A laminar flow over a flat plate was initially tested and validated, which complied with the classical Blasius solution. Plotted results of **(i)** dimensionalized & non-dimensionalized velocity and temperatures fields, **(ii)** Skin friction coefficient, and **(iii)** Stanton number over the plate were shown to prove the theoretical and numerical solutions agreement. Also, turbulent flow over a flat plate was presented. The turbulent flat plate (precursor) solutions were examined until the best match was injected as an inflow profile for the

turbulent flow over the curved hill. In terms of the curved hill scenario, two eddy-viscosity turbulence models (i.e., SST and SA) were used in this thesis. The pressure gauge, velocity, and temperature field contours were visible, as the pressure and skin friction coefficients agreed well with the experimental data.

Overall, the SA model had a better agreement with the experimental data in those zones where the turbulent boundary layer remained attached, for instance, in C_p, C_f, and U_s predictions. On the contrary, the SST model has depicted slightly superior agreement with experiments in the separation bubble (in terms of C_p and C_f) as well as in U_s profiles just upstream of the bubble, capturing more evidently the positively correlated characteristics of u' and v' or positive $< u'v >$. Both models have detected outer peaks of $< u'v >$, turbulence production, and the inclined shear layer caused by strong APG. We also highlighted the more notable effect of the detachment on the velocity profile (with significant differences between turbulent model predictions) than on the thermal profile. Strong streamline curvature-driven pressure gradients cause a noticeable Reynolds analogy breakdown. However, more data are needed to objectively judge their overall accuracy in the separation bubble. Additionally, there is a more dominant effect of the molecular Prandtl number (Pr) in the near wall region of turbulent boundary layer flows than in the outer region, where turbulent mixing prevails over molecular diffusion. As expected, the local Stanton number increases as Pr decreases.

Finally, Chapter §3 showed the flow visualization automated algorithm and the two XR visualization tools conceived for the presented thesis. The algorithm utilizes the capacities of the ParaView visualization toolkit and the USDZ data type to generate virtual objects. It has already been used for other works, meaning the Broad Impact expected has been accomplished. The Virtual Wind Tunnel (used for Virtual Reality) and FlowVisXR app (used for Augmented Reality) achieved the objectives and goals set for the current thesis. Five research items related to this work have been published, one is a journal paper, and three others were directly based on the XR methods presented in this job. XR technology is amazingly growing in applications and methods, which is hard to track. The evolution is so accelerated that a weakness has been identified. The constant tools creations and updates are now causing compatibility issues between the tools and libraries, leading to the withdrawal of the development and support of potentially powerful tools.

4.1 Future Work

Recalling, the main goals of this work are: **(i)** Improve the boundary layer's behavior knowledge in the separation zones related to pressure gradients, **(ii)** Discern the passive scalar transport in the separation zones, **(iii)** Sharpen the reader's fluid mechanics knowledge and scientific data visualization with XR, and **(iv)** Develop a tool for immersive visualization of CFD data results, enabling manipulation of virtual objects for clear examination.

To reach the desired outcomes, the following objectives were chosen:
• Perform laminar and turbulent simulations over a flat plate.
• Simulate the turbulent flow over the curved hill.
• Compute statistics of the velocity and passive scalar transport field considering the effect of different Prandtl numbers.
• Present a scheme to calculate boundary layer parameters to assure the CFD results.
• Define a simple methodology for VR and AR data visualization.
• Present the Virtual Wind Tunnel and FlowVisXR app as the enhancement approaches for scientific data visualization.

It can be said that all the objectives for this research still have been fully met. Given the results and conclusions, additional validation sources will be included for future steps to better characterize the considered model's performance in strong pressure gradients. Further, we will extend this study to a 3D geometry and conduct a large eddy simulation (LES) analysis to better capture the unsteadiness of the detached boundary layer and the laminarescent state we have hypothesized inside the separation bubble. Additionally, we will further investigate the proposed approach for boundary layer detection and its resiliency in more complex geometries and stronger pressure gradients and extend the study to other turbulence closure models.

Appendix A
From Navier-Stokes Equations to RANS

For an incompressible fluid, the continuity and Navier-Stokes equations in Cartesian coordinates are, respectively:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{A.1}$$

$$\begin{aligned}
\rho\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z}\right) &= -\frac{\partial P}{\partial x} + \mu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) + F_x \\
\rho\left(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z}\right) &= -\frac{\partial P}{\partial y} + \mu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) + F_y \\
\rho\left(\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z}\right) &= -\frac{\partial P}{\partial z} + \mu\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) + F_z
\end{aligned} \tag{A.2}$$

where u, v, w are the velocity components, P is the pressure, μ is the molecular kinematic viscosity, and x, y, z are the spatial unit vectors; in its tensor forms would it be:

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{A.3}$$

$$\rho\frac{Du_i}{Dt} = -\frac{\partial P}{\partial x_i} + \mu\Delta u_i + F_i \tag{A.4}$$

Because we might want to adapt these equations considering extra volume forces, we add the term of F_i.

For the Reynolds Averaged Navier-Stokes (RANS) approach, the goal is to only solve

the mean fields numerically, in other words, transform the Navier-Stokes equations to only have averaged terms but maintain the same form as the original equations. To achieve this we apply the following mathematically decomposition (known as Reynolds decomposition):

$$u = \bar{u} + u' \quad , \quad v = \bar{v} + v' \quad , \quad w = \bar{w} + w' \tag{A.5}$$

where momentary (or instantaneous) values, u, are decomposed into the sum of mean values, $\bar{u}$, and fluctuations values, u', using here the velocity component u as an example, but analogously would be the same formulation for others parameters like pressure (P), density (ρ) or temperature (t). This averaging method causes the mean values to be independent of time, while its values are fixed in space. In the case of fluctuating values, the time-averaged of these is zero, by definition of:

$$\overline{u'} = 0 \quad , \quad \overline{v'} = 0 \quad , \quad \overline{w'} = 0 \quad , \quad \overline{P'} = 0 \tag{A.6}$$

Applying equation A.5 into continuity equation A.1 and time-averaging we get:

$$\overline{\frac{\partial \bar{u}}{\partial x} + \frac{\partial u'}{\partial x} + \frac{\partial \bar{v}}{\partial y} + \frac{\partial v'}{\partial y} + \frac{\partial \bar{w}}{\partial z} + \frac{\partial w'}{\partial z}} = 0 \tag{A.7}$$

and with the rules of time averaging:

$$\bar{\bar{f}} = \bar{f} \quad , \quad \overline{f+g} = \bar{f} + \bar{g} \quad , \quad \overline{\bar{f} \cdot g} = \bar{f} \cdot \bar{g} \quad , \quad \overline{\int f ds} = \int \bar{f} ds \quad , \quad \overline{f \cdot g} \neq \bar{f} \cdot \bar{g} \tag{A.8}$$

$$\overline{\frac{\partial \bar{u}}{\partial x}} = \frac{1}{\Delta t} \int_{t_0}^{t_0+t_1} \frac{\partial \bar{u}}{\partial x} dt = \frac{\partial}{\partial x} \frac{1}{\Delta t} \int_{t_0}^{t_0+t_1} \bar{u} dt = \frac{\partial \bar{\bar{u}}}{\partial x} = \frac{\partial \bar{u}}{\partial x}$$

$$\overline{\frac{\partial u'}{\partial x}} = \frac{1}{\Delta t} \int_{t_0}^{t_0+t_1} \frac{\partial u'}{\partial x} dt = \frac{\partial}{\partial x} \frac{1}{\Delta t} \int_{t_0}^{t_0+t_1} u' dt = 0 \tag{A.9}$$

the continuity equation of RANS (A.7) is reduced to:

$$\frac{\partial \bar{u}}{\partial x} + \frac{\partial \bar{v}}{\partial y} + \frac{\partial \bar{w}}{\partial z} = 0 \tag{A.10}$$

Now, for the Navier-Stokes equations, first let's transform the advection term from equation A.2, and with help of the continuity equation A.10 we have:

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = \frac{\partial u^2}{\partial x} + \frac{\partial uv}{\partial y} + \frac{\partial uw}{\partial z} - u\cancel{\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)} \tag{A.11}$$

replacing eq. A.11 into eq. A.2, adding eq.A.5 and the time averaging we would see:

$$\overline{\rho\left(\frac{\partial(\bar{u}+u')}{\partial t} + \frac{\partial(\bar{u}+u')^2}{\partial x} + \frac{\partial(\bar{u}+u')(\bar{v}+v')}{\partial y} + \frac{\partial(\bar{u}+u')(\bar{w}+w')}{\partial z}\right)} = \overline{-\frac{\partial(\bar{P}+P')}{\partial x} + \mu\left(\frac{\partial^2(\bar{u}+u')}{\partial x^2} + \frac{\partial^2(\bar{u}+u')}{\partial y^2} + \frac{\partial^2(\bar{u}+u')}{\partial z^2}\right) + F_x} \quad ,$$

$$\overline{\rho\left(\frac{\partial(\bar{v}+v')}{\partial t} + \frac{\partial(\bar{v}+v')(\bar{u}+u')}{\partial x} + \frac{\partial(\bar{v}+v')^2}{\partial y} + \frac{\partial(\bar{v}+v')(\bar{w}+w')}{\partial z}\right)} = \overline{-\frac{\partial(\bar{P}+P')}{\partial y} + \mu\left(\frac{\partial^2(\bar{v}+v')}{\partial x^2} + \frac{\partial^2(\bar{v}+v')}{\partial y^2} + \frac{\partial^2(\bar{v}+v')}{\partial z^2}\right) + F_y} \quad ,$$

$$\overline{\rho\left(\frac{\partial(\bar{w}+w')}{\partial t} + \frac{\partial(\bar{w}+w')(\bar{u}+u')}{\partial x} + \frac{\partial(\bar{w}+w')(\bar{v}+v')}{\partial y} + \frac{\partial(\bar{w}+w')^2}{\partial z}\right)} = \overline{-\frac{\partial(\bar{P}+P')}{\partial z} + \mu\left(\frac{\partial^2(\bar{w}+w')}{\partial x^2} + \frac{\partial^2(\bar{w}+w')}{\partial y^2} + \frac{\partial^2(\bar{w}+w')}{\partial z^2}\right) + F_z} \tag{A.12}$$

and with the same rules of equations A.8 & A.9, the Navier-Stokes equations for RANS are reduced to:

$$\rho\left(\frac{\partial\bar{u}}{\partial t}+\frac{\partial\overline{uu}}{\partial x}+\frac{\partial\overline{u'u'}}{\partial x}+\frac{\partial\overline{uv}}{\partial y}+\frac{\partial\overline{u'v'}}{\partial y}+\frac{\partial\overline{uw}}{\partial z}+\frac{\partial\overline{u'w'}}{\partial z}\right)=-\frac{\partial\bar{P}}{\partial x}+\mu\left(\frac{\partial^2\bar{u}}{\partial x^2}+\frac{\partial^2\bar{u}}{\partial y^2}+\frac{\partial^2\bar{u}}{\partial z^2}\right)+F_x \quad ,$$

$$\rho\left(\frac{\partial\bar{v}}{\partial t}+\frac{\partial\overline{vu}}{\partial x}+\frac{\partial\overline{v'u'}}{\partial x}+\frac{\partial\overline{vv}}{\partial y}+\frac{\partial\overline{v'v'}}{\partial y}+\frac{\partial\overline{vw}}{\partial z}+\frac{\partial\overline{v'w'}}{\partial z}\right)=-\frac{\partial\bar{P}}{\partial y}+\mu\left(\frac{\partial^2\bar{v}}{\partial x^2}+\frac{\partial^2\bar{v}}{\partial y^2}+\frac{\partial^2\bar{v}}{\partial z^2}\right)+F_y \quad ,$$

$$\rho\left(\frac{\partial\bar{w}}{\partial t}+\frac{\partial\overline{wu}}{\partial x}+\frac{\partial\overline{w'u'}}{\partial x}+\frac{\partial\overline{wv}}{\partial y}+\frac{\partial\overline{w'v'}}{\partial y}+\frac{\partial\overline{ww}}{\partial z}+\frac{\partial\overline{w'w'}}{\partial z}\right)=-\frac{\partial\bar{P}}{\partial z}+\mu\left(\frac{\partial^2\bar{w}}{\partial x^2}+\frac{\partial^2\bar{w}}{\partial y^2}+\frac{\partial^2\bar{w}}{\partial z^2}\right)+F_z \tag{A.13}$$

moving fluctuations terms to the RHS and using the Laplacian symbol ($\Delta=\frac{\partial^2}{\partial x_i^2}$) for shorthand in the diffusion terms we get:

$$\rho\left(\frac{\partial\bar{u}}{\partial t}+\bar{u}\frac{\partial\bar{u}}{\partial x}+\bar{v}\frac{\partial\bar{u}}{\partial y}+\bar{w}\frac{\partial\bar{u}}{\partial z}\right)=-\frac{\partial\bar{P}}{\partial x}+\mu\Delta\bar{u}-\rho\left(\frac{\partial\overline{u'u'}}{\partial x}+\frac{\partial\overline{u'v'}}{\partial y}+\frac{\partial\overline{u'w'}}{\partial z}\right)+F_x \quad ,$$

$$\rho\left(\frac{\partial\bar{v}}{\partial t}+\bar{u}\frac{\partial\bar{v}}{\partial x}+\bar{v}\frac{\partial\bar{v}}{\partial y}+\bar{w}\frac{\partial\bar{v}}{\partial z}\right)=-\frac{\partial\bar{P}}{\partial y}+\mu\Delta\bar{v}-\rho\left(\frac{\partial\overline{v'u'}}{\partial x}+\frac{\partial\overline{v'v'}}{\partial y}+\frac{\partial\overline{v'w'}}{\partial z}\right)+F_y \quad ,$$

$$\rho\left(\frac{\partial\bar{w}}{\partial t}+\bar{u}\frac{\partial\bar{w}}{\partial x}+\bar{v}\frac{\partial\bar{w}}{\partial y}+\bar{w}\frac{\partial\bar{w}}{\partial z}\right)=-\frac{\partial\bar{P}}{\partial z}+\mu\Delta\bar{w}-\rho\left(\frac{\partial\overline{w'u'}}{\partial x}+\frac{\partial\overline{w'v'}}{\partial y}+\frac{\partial\overline{w'w'}}{\partial z}\right)+F_z \tag{A.14}$$

or in its tensor form:

$$\rho\frac{D\bar{u}_i}{Dt}=-\frac{\partial\bar{P}}{\partial x_i}+\mu\Delta\bar{u}_i-\rho\left(\frac{\partial\overline{u_i'u_j'}}{\partial x_j}\right)+F_i \tag{A.15}$$

Paying attention to the following two terms of the RHS of eq. A.15, notice how we can group them as:

$$\mu\Delta\bar{u}_i-\rho\left(\frac{\partial\overline{u_i'u_j'}}{\partial x_j}\right)\longrightarrow\mu\frac{\partial}{\partial x_j}\left(\frac{\partial\bar{u}_i}{\partial x_j}\right)-\rho\frac{\partial}{\partial x_j}\left(\overline{u_i'u_j'}\right)\longrightarrow\frac{\partial}{\partial x_j}\left(\mu\frac{\partial\bar{u}_i}{\partial x_j}-\rho\overline{u_i'u_j'}\right) \tag{A.16}$$

where if we define $\hat{\tau}_{ij}$ as the total shear stress tensor:

$$\hat{\tau}_{ij} = \mu \frac{\partial \bar{u}_i}{\partial x_j} - \rho \overline{u_i' u_j'} \tag{A.17}$$

our final equation for momentum conservation uses the eddy viscosity principle of Boussinesq Hypothesis (Moukalled et al. 2016) where the contribution of the fluctuations terms can be estimated from a parameter called turbulent kinetic energy. This momentum conservation equation for RANS is given as:

$$\rho \frac{D\bar{u}_i}{Dt} = -\frac{\partial \bar{P}}{\partial x_i} + \frac{\partial}{\partial x_i} \hat{\tau}_{ij} + F_i \tag{A.18}$$

with

$$\hat{\tau}_{ij} = \mu \frac{\partial \bar{u}_i}{\partial x_j} + \rho \left(\nu_t \left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i} \right) - \frac{2}{3} k \delta_{ij} \right)$$

Appendix B
Post-processing Tools for CFD

B.1 postProcessing.py

```
import numpy as np
import matplotlib.pyplot as plt
np.seterr(divide='ignore', invalid='ignore')

meshQuality = 'fine'      #"mid"      # 'fine'
set=0                   ###   CP=0, CF,St=1, INT=2
workingPATH = './'

### Preparing the path for the three turbulent cases
SSTpath= f"{workingPATH}../VTKs/SST_nCells2708000_100000_100x50000.txt"
SApath= f"{workingPATH}../VTKs/SA_nCells2708000_100000_100x50000.txt"
POTENTIALpath= f"{workingPATH}../VTKs/
                                potential_nCells2708000_0_100x50000
                                .txt"


Nx=100
Ny=50000

def nan_helper(y):
    """Helper to handle indices and logical indices of NaNs.

    Input:
        - y, 1d numpy array with possible NaNs
    Output:
        - nans, logical indices of NaNs
        - index, a function, with signature indices= index(
                                            logical_indices),
          to convert logical indices of NaNs to 'equivalent' indices
    Example:
        >>> # linear interpolation of NaNs
        >>> nans, x= nan_helper(y)
```

```
        >>> y[nans]= np.interp(x(nans), x(~nans), y[~nans])
    """

    return np.isnan(y), lambda z: z.nonzero()[0]
def solve_nan(y):
    nans, x= nan_helper(y)
    y[nans]= np.interp(x(nans), x(~nans), y[~nans])
    return y

def transform(x, y):
    Nx, Ny = x.shape
    s = np.zeros_like(x)
    n = np.zeros_like(y)

    s[0, :] = x[0, :]

    for ii in range(1,Nx):
        for jj in range(Ny):
            s[ii,jj] = s[ii-1, jj] + np.sqrt((x[ii,jj] - x[ii-1,jj])**2
                                             + (y[ii,jj] - y[ii-1,
                                             jj])**2)
            n[ii,jj] = np.sqrt((x[ii,jj] - x[ii,0])**2 + (y[ii,jj] - y[
                                             ii,0])**2)
    return s, n

def rotateVelocity_in2D(U: np.ndarray, V: np.ndarray, theta: np.ndarray
                                        ):
    cosTheta = np.cos(theta).reshape((-1,1))
    sinTheta = np.sin(theta).reshape((-1,1))
    Us = cosTheta * U - sinTheta * V
    Vn = sinTheta * U + cosTheta * V
    return Us, Vn

def postProcessing_on_wall2D(Cx,n,Ux,Ux_potential,nu,Temp,Pr):

    pOut= 10 # [y_blt u_tau SkinFCoef dispThick momentumT shapeFactor
                                        Re_theta y_tempBLT dTheta/dy
                                        Stanton]
    (Nx, Ny) = Cx.shape
    Output = np.zeros((Nx,pOut))

    ###  New Parameters with shape (Nx,Ny)
    Temp_inf = np.mean(Temp[:,9*Ny//10:Ny])

                                        # np.mean(Temp[:,-1])
    Temp_0 = np.mean(Temp[:,0])
```

```
temp_normalized = (Temp-Temp_0)/(Temp_inf-Temp_0)

### Preparing arrays
yindex_blt, u_edge, y_blt = np.zeros(Nx,dtype=int), np.zeros(Nx),
                                    np.zeros(Nx)
u_tau, SkinFCoef = np.zeros(Nx), np.zeros(Nx)
dispThick, momentumT = np.zeros(Nx), np.zeros(Nx)
yindex_tempBLT, y_tempBLT, Stanton = np.zeros(Nx,dtype=int), np.
                                    zeros(Nx), np.zeros(Nx)
temp_normalized, dThetady0 = np.zeros((Nx,Ny)), np.zeros(Nx)

### First derivative with 5-points Stencil
dudn=np.zeros_like(Ux)
dudn[:,2:Ny-2] = (-Ux[:,4:] +8*Ux[:,3:-1] -8*Ux[:,1:-3] +Ux[:,:-4])
                                    /(12*(n[:,4:]-n[:,:-4]))
dudn_potential=np.zeros_like(Ux_potential)
dudn_potential[:,2:Ny-2] = (-Ux_potential[:,4:] +8*Ux_potential[:,3
                                    :-1] -8*Ux_potential[:,1:-3] +
                                    Ux_potential[:,:-4])/(12*(n[:,4
                                    :]-n[:,:-4]))
### Second derivative with 5-points Stencil
dudn2=np.zeros_like(dudn)
dudn2[:,2:Ny-2] = (-dudn[:,4:] +8*dudn[:,3:-1] -8*dudn[:,1:-3] +
                                    dudn[:,:-4])/(12*(n[:,4:]-n[:,:
                                    -4]))
dudn_potential2=np.zeros_like(dudn_potential)
dudn_potential2[:,2:Ny-2] = (-dudn_potential[:,4:] +8*
                                    dudn_potential[:,3:-1] -8*
                                    dudn_potential[:,1:-3] +
                                    dudn_potential[:,:-4])/(12*(n[:
                                    ,4:]-n[:,:-4]))

### For each X-stations:
for x in range(Nx):

    Ux[x,:]=solve_nan(Ux[x,:]) #np.argwhere(np.isnan(x))

    ### Finding yindex_blt, u_edge, y_blt
    for y in range(5,Ny-5):
        if  abs(dudn2[x,y]) <= abs(dudn_potential2[x,y]) and (Ux[x,
                                        y]**2 >= 0.99*
                                        Ux_potential[x,y]**2) :
            yindex_blt[x] = int(y)
            u_edge[x] = Ux_potential[x,y]
```

```
            y_blt[x] = n[x,y]                                   #+ abs(
                                                            Ux_potential[x,y]-
                                                            Ux[x,y])*(n[x,y+1]-
                                                            n[x,y])/(Ux[x,y+1]-
                                                            Ux[x,y])
            break

    ### Skin Friction Velocity
    dudy0 = 0.5*((-Ux[x,4] +8*Ux[x,3] -8*Ux[x,1] +Ux[x,0])/(12*(n[x
                                        ,1]-n[x,0]))) + 0.5*((-Ux[x
                                        ,5] +8*Ux[x,4] -8*Ux[x,2] +
                                        Ux[x,1])/(12*(n[x,1]-n[x,0]
                                        )))
    if dudy0 > 0 :
            u_tau[x] = np.sqrt(nu*dudy0)                        # sqrt(
                                                            nu*(du/dy)) @y=0
    elif dudy0 < 0 :
            u_tau[x] = -1*np.sqrt(nu*(abs(dudy0)))              # -sqrt
                                                            (nu*(du/dy)) @y=0

    ### Skin Friction Coefficient
    if u_tau[x] > 0 :
        SkinFCoef[x]= 2*(u_tau[x]/u_edge[x])**2                 # 2*(
                                                        u_tau/u_inf)^2
    elif u_tau[x] < 0 :
        SkinFCoef[x]= -2*(u_tau[x]/u_edge[x])**2                # -2*(
                                                        u_tau/u_inf)^2

    ### Integral parameters
    if (yindex_blt[x] % 2) == 0: # A number is even if division by
                                        2 gives a remainder of 0.
        pass
    else:
        yindex_blt[x]+=1

    ratio0=Ux[x,0]/u_edge[x]
    ratioF=Ux[x,yindex_blt[x]]/u_edge[x]
    dispThick[x] = (1-ratio0) + (1-ratioF)
    momentumT[x] = (ratio0)*(1-ratio0) + (ratioF)*(1-ratioF)

    for y in range(1,yindex_blt[x]):
        ratio=Ux[x,y]/u_edge[x]
        if (y % 2) == 0:
            dispThick[x] = dispThick[x] + 2*(1-ratio)
            momentumT[x] = momentumT[x] + 2*(ratio)*(1-ratio)
        else :
```

```
            dispThick[x] = dispThick[x] + 4*(1-ratio)
            momentumT[x] = momentumT[x] + 4*(ratio)*(1-ratio)

    dispThick[x] = dispThick[x]*(n[x,1]-n[x,0])/3
    momentumT[x] = momentumT[x]*(n[x,1]-n[x,0])/3

    ### Thermal calculations
    Temp_inf = np.mean(Temp[x,9*Ny//10:Ny])

                                        # np.mean(Temp[:,-1])
    Temp_0 = np.mean(Temp[x,0])
    temp_normalized[x,:] = (Temp[x,:]-Temp_0)/(Temp_inf-Temp_0)
    for y in range(Ny-1):
        if temp_normalized[x,y+1] >= 0.99:
            y_tempBLT[x]= n[x,y] + (0.99-temp_normalized[x,y])*(n[x
                                                ,y+1]-n[x,y])/(
                                                temp_normalized[x,y
                                                +1]-temp_normalized
                                                [x,y])
            yindex_tempBLT[x] = int(y)
            break

    ### Stanton = h/(rho*Uinf*Cp) = Nusselt/(Re_x * Pr)
    # h*(Tinf-Tw) = Q_w = -k*(dT/dy)_wall                h = -k*(dT/
                                        dy)_wall /(Tinf-Tw)
    # (nu/Pr) = alpha = k/(rho*Cp)
    # -k*(dT/dy)_wall /((Tinf-Tw)*(rho*Uinf*Cp))         -alpha* (dT
                                        /dy)_wall /((Tinf-Tw)*Uinf)
    dTdy0 = 0.5*((-Temp[x,4] +8*Temp[x,3] -8*Temp[x,1] +Temp[x,0])/
                                        (12*(n[x,1]-n[x,0]))) + 0.5
                                        *((-Temp[x,5] +8*Temp[x,4]
                                        -8*Temp[x,2] +Temp[x,1])/(
                                        12*(n[x,1]-n[x,0])))
    # Stanton number (here goes a -1*),
    Stanton[x]=  (nu/Pr)*dTdy0/(u_edge[x]*(Temp_inf-Temp_0))  # -
                                        alpha*dTdy_wall/(u_inf*(
                                        t_inf-T[0])      (nu/Pr) =
                                        alpha= k/(rho*Cp)
```

```
        dThetady0[x] = 0.5*((-temp_normalized[x,4] +8*temp_normalized[x
                                              ,3] -8*temp_normalized[x,1]
                                               +temp_normalized[x,0])/(12
                                              *(n[x,1]-n[x,0]))) + 0.5*((
                                              -temp_normalized[x,5] +8*
                                              temp_normalized[x,4] -8*
                                              temp_normalized[x,2] +
                                              temp_normalized[x,1])/(12*(
                                              n[x,1]-n[x,0])))

    Output[:,0]= y_blt
    Output[:,1]= u_tau
    Output[:,2]= SkinFCoef
    Output[:,3]= dispThick
    Output[:,4]= momentumT
    Output[:,5]= dispThick/momentumT                #Shape Factor
    Output[:,6]= np.mean(Ux_p[:2,1])*momentumT/nu   #Reynolds_theta
    Output[:,7]= y_tempBLT
    Output[:,8]= dThetady0
    Output[:,9]= Stanton

    return Output


### Reading CurveScan data files from Baskaran et al.'s figures
dataCP = np.loadtxt(f'{workingPATH}../DigitizedPlots/Cp.txt')
                                                              #
                                      Distances are in [mm]
dataCF = np.loadtxt(f'{workingPATH}../DigitizedPlots/Cf.txt', usecols=(
                                      0,1))                # Distances are
                                       in [mm]
dataINT = np.loadtxt(f'{workingPATH}../DigitizedPlots/
                                      CurvedHill_IntegralParameters.txt')
                                        # Distances are in [m]

### Reading all the csv files
SSTdata = np.loadtxt(SSTpath).reshape(Nx,Ny,7)
SAdata = np.loadtxt(SApath).reshape(Nx,Ny,7)
POTENTIALdata = np.loadtxt(POTENTIALpath).reshape(Nx,Ny,7)

### SST Columns Names= "T","Ux","Uy","arc_length","p","Cx","Cy"
T_SST = SSTdata[:,:,0]
Ux_SST,Uy_SST = SSTdata[:,:,1],SSTdata[:,:,2]
n_SST = SSTdata[:,:,3]
p_SST = SSTdata[:,:,4]
Cx_SST,Cy_SST= SSTdata[:,:,5],SSTdata[:,:,6]
```

```
### SA Columns Names= "T","Ux","Uy","arc_length","p","Cx","Cy"
T_SA = SAdata[:,:,0]
Ux_SA,Uy_SA= SAdata[:,:,1],SAdata[:,:,2]
n_SA = SAdata[:,:,3]
p_SA = SAdata[:,:,4]
Cx_SA,Cy_SA= SAdata[:,:,5],SAdata[:,:,6]

### Reading the angle of surface direction to rotate the velocities
angleTheta = POTENTIALdata[:,0,0]

### Potential Columns Names= "Normal_x","Ux","Uy","arc_length","p","Cx
                                        ","Cy"
Ux_p,Uy_p = POTENTIALdata[:,:,1],POTENTIALdata[:,:,2]
n_p = POTENTIALdata[:,:,3]
p_p = POTENTIALdata[:,:,4]
Cx_p,Cy_p = POTENTIALdata[:,:,5],POTENTIALdata[:,:,6]

offset=1.65
Cx_SST,Cy_SST = transform(Cx_SST,Cy_SST)
Cx_SST = (Cx_SST+offset)*1000
Cx_SA,Cy_SA = transform(Cx_SA,Cy_SA)
Cx_SA = (Cx_SA+offset)*1000
Cx_p,Cy_p = transform(Cx_p,Cy_p)
Cx_p = (Cx_p+offset)*1000

if set == 0:
    indexRef= 23*Nx//100

    Ux_p,Uy_p=rotateVelocity_in2D(Ux_p,Uy_p,angleTheta)
    freeVel = np.mean(Ux_p[:indexRef,1])

                                        #Only taking ZPG stations
    print(np.mean(Ux_p[:indexRef,1]) )

    plt.title('Wall-Static-Pressure Coeff. distribution')
    plt.plot( dataCP[:,0], dataCP[:,1] ,'o',color="black",alpha=.5,
                                        label= 'Baskaran et al.
                                        Experimental')
    plt.plot( Cx_SA[:,0], (p_SA[:,0]-p_SA[13*Nx//100,0])/(0.5*freeVel**
                                        2) ,'x', color="g" , label= f'{
                                        meshQuality} Mesh SA',alpha=.75
                                        )
```

```
    plt.plot( Cx_SST[:,0], (p_SST[:,0]-p_SST[13*Nx//100,0])/(0.5*
                                        freeVel**2) ,'x', color="r" ,
                                        label= f'{meshQuality} Mesh SST
                                        ',alpha=.75)
    plt.legend(loc='best')
    plt.subplots_adjust(bottom=0.18)
    plt.xlim((0,3000))
    plt.ylabel('$C_p$')
    plt.xlabel('$S$ [mm]')
    plt.xticks(rotation=45)
    plt.grid(True, which='both')
    #plt.show()
    plt.savefig(f'{workingPATH}CP_{meshQuality}.png', bbox_inches="
                                        tight", dpi=300)
    plt.cla()
    plt.clf()
    plt.close()

if set == 1:
    Ux_SST,Uy_SST=rotateVelocity_in2D(Ux_SST,Uy_SST,angleTheta)
    Ux_SA,Uy_SA=rotateVelocity_in2D(Ux_SA,Uy_SA,angleTheta)
    Ux_p,Uy_p=rotateVelocity_in2D(Ux_p,Uy_p,angleTheta)

    postData_SST = postProcessing_on_wall2D(Cx_SST,n_SST,Ux_SST,Ux_p,1.
                                        5e-5,T_SST,0.71)
    postData_SA = postProcessing_on_wall2D(Cx_SA,n_SA,Ux_SA,Ux_p,1.5e-5
                                        ,T_SA,0.71)

    plt.title('Skin-friction Coeff. distribution')
    plt.plot( dataCF[:,0], dataCF[:,1] ,'o',color="black",alpha=.5,
                                        label= 'Baskaran et al.
                                        Experimental')
    plt.plot( Cx_SA[:,0], postData_SA[:,2] ,'x', color="g" , label= f'{
                                        meshQuality} Mesh SA',alpha=.75
                                        )
    plt.plot( Cx_SST[:,0], postData_SST[:,2] ,'x', color="r" , label= f
                                        '{meshQuality} Mesh SST',alpha=
                                        .75)
    plt.legend(loc='best')
    plt.subplots_adjust(bottom=0.18)
    plt.xlim((0,3000))
    plt.ylim((-0.002,0.01))
    plt.ylabel('$C_f$')
    plt.xlabel('$S$ [mm]')
    plt.xticks(rotation=45)
    plt.grid(True, which='both')
```

```
plt.savefig(f'{workingPATH}CF_{meshQuality}.png', bbox_inches="
                                    tight", dpi=300)
plt.cla()
plt.clf()
plt.close()

plt.title('Stanton number (St)')
plt.plot( Cx_SA[:,0], postData_SA[:,9] ,'x', color="g" , label= f'{
                                    meshQuality} Mesh SA',alpha=.75
                                    )
plt.plot( Cx_SST[:,0], postData_SST[:,9] ,'x', color="r" , label= f
                                    '{meshQuality} Mesh SST',alpha=
                                    .75)
# [y_blt u_tau SkinFCoef dispThick momentumT shapeFactor Re_theta
                                    y_tempBLT dTheta/dy Stanton]
from scipy.optimize import curve_fit  #Use non-linear least squares
                                     to fit a function, f, to data.
def func(x, a, b):
    return a*np.exp(b*x)
#popt, pcov = curve_fit(func,dataCF[:11,0]/1000,postData_SA[:11,6])
popt, pcov = curve_fit(func,Cx_SA[:22*Nx//100,0]/1000,postData_SA[:
                                    22*Nx//100,6])
empirical_x = np.linspace(150,1000,500)/1000
empirical_Re_th = func(empirical_x, *popt)
St_kays = 0.01372*empirical_Re_th**(-0.25)
#St_kays = 0.0287*0.71**(-0.4)*(empirical_x)**(-0.2)

                                    #eq. 13-18 Kays & Crawford (
                                    1993)
plt.plot(empirical_x*1000,St_kays,'-',label= 'Kays & Crawford (1993
                                    ); $S_t$=$0.01372 Re^{-1/4}_ $
                                    ')
plt.legend(loc='best')
plt.subplots_adjust(bottom=0.18)
plt.xlim((0,3000))
plt.ylim(0,0.004)
plt.ylabel('[-]')
plt.xlabel('$S$ [mm]')
plt.xticks(rotation=45)
plt.grid(True, which='both')
plt.savefig(f'{workingPATH}St_{meshQuality}.png', bbox_inches="
                                    tight", dpi=300)
plt.cla()
plt.clf()
plt.close()
```

```
    plt.title('Ratio $S_t/(C_f/2)$')
    plt.plot( Cx_SA[:,0], abs(postData_SA[:,9]*2/postData_SA[:,2]) ,'x'
                                        , color="g" , label= f'{
                                        meshQuality} Mesh SA',alpha=.75
                                        )
    plt.plot( Cx_SST[:,0], abs(postData_SST[:,9]*2/postData_SST[:,2]) ,
                                        'x', color="r" , label= f'{
                                        meshQuality} Mesh SST',alpha=.
                                        75)
    empirical_x = np.linspace(150,1000,500)
    empirical_ratio = 0.71**(-0.4)*np.ones_like(empirical_x)

                                        # Kays & Crawford (1993)
    plt.plot(empirical_x,empirical_ratio,'-',label= 'Kays & Crawford (
                                        1993); $S_t/(C_f/2)$=$Pr^{-2/5}
                                        $')
    plt.legend(loc='best')
    plt.subplots_adjust(bottom=0.18)
    plt.xlim((0,3000))
    plt.ylim(0,8)
    plt.ylabel('[-]')
    plt.xlabel('$S$ [mm]')
    plt.xticks(rotation=45)
    plt.grid(True, which='both')
    plt.savefig(f'{workingPATH}ratioStCf_{meshQuality}.png',
                                        bbox_inches="tight", dpi=300)
    plt.cla()
    plt.clf()
    plt.close()

if set == 2:
    dataStats = [596,710,867,1015,1139,1183,1345,1469,1596,1665,1730,
                                        1862,1990]    # Integral
                                        parameters' Stations in [mm]


    Ux_SST,Uy_SST=rotateVelocity_in2D(Ux_SST,Uy_SST,angleTheta)
    Ux_SA,Uy_SA=rotateVelocity_in2D(Ux_SA,Uy_SA,angleTheta)
    Ux_p,Uy_p=rotateVelocity_in2D(Ux_p,Uy_p,angleTheta)

    postData_SST = postProcessing_on_wall2D(Cx_SST,n_SST,Ux_SST,Ux_p,1.
                                        5e-5,T_SST,0.71)
    postData_SA = postProcessing_on_wall2D(Cx_SA,n_SA,Ux_SA,Ux_p,1.5e-5
                                        ,T_SA,0.71)

    # More versatile wrapper to Create figure and subplot  manually
```

```
# https://stackoverflow.com/questions/9103166/multiple-axis-in-
                                    matplotlib-with-different-
                                    scales

### Plot of Integral parameters for SST: ( S vs y_blt ShapeFactor
                                    Reynolds_th )
fig, host = plt.subplots(figsize=(8,5)) # (width, height) in inches

par1 = host.twinx()
par2 = host.twinx()

host.set_xlabel(" s [mm]")
host.set_ylabel("     Boundary Layer Thickness [mm]")     # ($ $)
par1.set_ylabel("     Reynolds$_ $ [$10x^3$]")             # (  )
                                    Momentum Thickness [mm]
par2.set_ylabel("     Shape factor")                      # ($ ^*$
                                    ) Displacement Thickness [mm]

host.set_ylim(-60, 80)
host.set_yticks(np.arange(20, 80, step=10))
par1.set_ylim(-4, 18)
par1.set_yticks(np.arange(2, 14, step=2))
par2.set_ylim(1,3.5)
par2.set_yticks(np.arange(1.1, 2.1, step=.2))

# [y_blt u_tau SkinFCoef dispThick momentumT shapeFactor Re_theta
                                    y_tempBLT Stanton]
p0, = host.plot(Cx_SST[Nx//10:8*Nx//10,0], postData_SST[Nx//10:8*Nx
                                    //10,0]*1000,'s', color='b',
                                    alpha=.75)
p1, = par1.plot(Cx_SST[Nx//10:8*Nx//10,0], postData_SST[Nx//10:8*Nx
                                    //10,6]/1000,'o', color='c',
                                    alpha=.85)
p2, = par2.plot(Cx_SST[Nx//10:8*Nx//10,0], postData_SST[Nx//10:8*Nx
                                    //10,5],'^', color='m',alpha=.
                                    65)

par3 = host.twiny()
par4 = par1.twiny()
par5 = par2.twiny()

p3, = par3.plot(dataStats, dataINT[:,1],':s', color="black",alpha=.
                                    5)
p4, = par4.plot(dataStats, dataINT[:,9]/1000,':o', color="black",
                                    alpha=.5)
```

```
p5, = par5.plot(dataStats, dataINT[:,7],':^', color="black",alpha=.
                                        5)

p6, = host.plot([],[],':',color="k",alpha=.5,label='Baskaran et al.
                                         Experimental')
lns = [p6]
host.legend(handles=lns, loc='lower right',bbox_to_anchor=(1, -0.15
                                        ))

# Moving the y-axis
host.yaxis.set_label_coords(-0.005, .68)
host.tick_params(axis='y',direction='in',pad=-20, colors=p0.
                                        get_color())
par1.yaxis.set_label_coords(1.01,.5)
par1.tick_params(axis='y',direction='in',pad=-20, colors=p1.
                                        get_color())
par2.spines['left'].set_position(('outward', 0))
par2.yaxis.set_label_coords(-0.025,.16)
par2.yaxis.set_ticks_position('left')
par2.tick_params(axis='y',direction='in',pad=-25, colors=p2.
                                        get_color())

# Turning off redundant axis
par3.axis('off')
par4.axis('off')
par5.axis('off')

host.set_xlim(0, 3000)
par3.set_xlim(0, 3000)
par4.set_xlim(0, 3000)
par5.set_xlim(0, 3000)

host.yaxis.label.set_color(p0.get_color())
par1.yaxis.label.set_color(p1.get_color())
par2.yaxis.label.set_color(p2.get_color())
plt.title(f"Integral parameters of SST {meshQuality} mesh")
plt.savefig(f'{workingPATH}integralParameters_SST{meshQuality}.png'
                                        , bbox_inches="tight", dpi=300)
plt.cla()
plt.clf()
plt.close()

### Plot of Integral parameters for SA: ( S vs y_blt ShapeFactor
                                        Reynolds_th )
fig, host = plt.subplots(figsize=(8,5)) # (width, height) in inches
par1 = host.twinx()
```

```
par2 = host.twinx()
host.set_xlabel(" s [mm]")

host.set_ylim(-60, 80)
host.set_yticks(np.arange(20, 80, step=10))
host.set_ylabel("     Boundary Layer Thickness [mm]")     # ($ $)
par1.set_ylim(-4, 18)
par1.set_yticks(np.arange(2, 14, step=2))
par1.set_ylabel("     Reynolds$_ $ [$10x^3$]")            # (  )
                                    Momentum Thickness [mm]
par2.set_ylim(1,3.5)
par2.set_yticks(np.arange(1.1, 2.1, step=.2))
par2.set_ylabel("     Shape factor")                      # ($  ^*$
                                    ) Displacement Thickness [mm]

# [y_blt u_tau SkinFCoef dispThick momentumT shapeFactor Re_theta
                                    y_tempBLT Stanton]
p0, = host.plot(Cx_SA[Nx//10:8*Nx//10,0], postData_SA[Nx//10:8*Nx//
                                    10,0]*1000,'s', color='b',alpha
                                    =.75)
p1, = par1.plot(Cx_SA[Nx//10:8*Nx//10,0], postData_SA[Nx//10:8*Nx//
                                    10,6]/1000,'o', color='c',alpha
                                    =.85)
p2, = par2.plot(Cx_SA[Nx//10:8*Nx//10,0], postData_SA[Nx//10:8*Nx//
                                    10,5],'^', color='m',alpha=.65)

par3 = host.twiny()
par4 = par1.twiny()
par5 = par2.twiny()
p3, = par3.plot(dataStats, dataINT[:,1],':s', color="black",alpha=.
                                    5)
p4, = par4.plot(dataStats, dataINT[:,9]/1000,':o', color="black",
                                    alpha=.5)
p5, = par5.plot(dataStats, dataINT[:,7],':^', color="black",alpha=.
                                    5)
p6, = host.plot([],[],':',color="k",alpha=.5,label='Baskaran et al.
                                     Experimental')
lns = [p6]
host.legend(handles=lns, loc='lower right',bbox_to_anchor=(1, -0.15
                                    ))

# Moving and setting the y-axis
host.yaxis.set_label_coords(-0.005, .68)
host.tick_params(axis='y',direction='in',pad=-20, colors=p0.
                                    get_color())
host.yaxis.label.set_color(p0.get_color())
```

```
par1.yaxis.set_label_coords(1.01,.5)
par1.tick_params(axis='y',direction='in',pad=-20, colors=p1.
                                    get_color())
par1.yaxis.label.set_color(p1.get_color())
par2.spines['left'].set_position(('outward', 0))
par2.yaxis.set_label_coords(-0.025,.16)
par2.yaxis.set_ticks_position('left')
par2.tick_params(axis='y',direction='in',pad=-25, colors=p2.
                                    get_color())
par2.yaxis.label.set_color(p2.get_color())

# Turning off redundant axis and limiting horizontals
par3.axis('off')
par4.axis('off')
par5.axis('off')
host.set_xlim(0, 3000)
par3.set_xlim(0, 3000)
par4.set_xlim(0, 3000)
par5.set_xlim(0, 3000)

plt.title(f"Integral parameters of SA {meshQuality} mesh")
plt.savefig(f'{workingPATH}integralParameters_SA{meshQuality}.png',
                                     bbox_inches="tight", dpi=300)
plt.cla()
plt.clf()
plt.close()

### Plot of Integral parameters for SA ans SST: ( S vs
                                    Displacement_Thickness
                                    Momentum_Thickness )
fig, host = plt.subplots(figsize=(8,5)) # (width, height) in inches
host.set_xlabel(" s [mm]")
par1 = host.twinx()

host.set_ylim(-4, 17)
host.set_yticks(np.arange(0, 17, step=2))
host.set_ylabel("     Displacement Thickness [mm]     ")
par1.set_ylim(-1, 20)
par1.set_yticks(np.arange(0, 9, step=2))
par1.set_ylabel("     Momentum Thickness [mm]     ")

# [y_blt u_tau SkinFCoef dispThick momentumT shapeFactor Re_theta
                                    y_tempBLT Stanton]
p0, = host.plot(Cx_SST[Nx//10:8*Nx//10,0], postData_SST[Nx//10:8*Nx
                                    //10,3]*1000,'d', color='r',
                                    alpha=.75)
```

```
    p1, = host.plot(Cx_SA[Nx//10:8*Nx//10,0], postData_SA[Nx//10:8*Nx//
                                        10,3]*1000,'d', color='g',alpha
                                        =.75)
    p2, = host.plot(dataStats, dataINT[:,3],':d', color="k",alpha=.5)

    p3, = par1.plot(Cx_SST[Nx//10:8*Nx//10,0], postData_SST[Nx//10:8*Nx
                                        //10,4]*1000,'p', color="r",
                                        alpha=.75)
    p4, = par1.plot(Cx_SA[Nx//10:8*Nx//10,0], postData_SA[Nx//10:8*Nx//
                                        10,4]*1000,'p', color="g",alpha
                                        =.75)
    p5, = par1.plot(dataStats, dataINT[:,5],':p', color="k",alpha=.5)
    p6, = host.plot([],[],'d',color="g",alpha=.75,label='
                                        Turbulent model $SA$')
    p7, = host.plot([],[],'d',color="r",alpha=.75,label='
                                        Turbulent model $SST$')
    p8, = host.plot([],[],':d',color="k",alpha=.5,label='-   - Baskaran
                                         et al. Experimental')
    lns = [p6,p7,p8]
    host.legend(handles=lns, loc='best',labelcolor='linecolor')#,
                                        bbox_to_anchor=(1, -0.15))

    # Moving and setting the y-axis
    host.yaxis.set_label_coords(-0.005, .55)
    host.tick_params(axis='y',direction='in',pad=-20, colors='k')
    host.yaxis.label.set_color('k')
    par1.yaxis.set_label_coords(1.01,.3)
    par1.tick_params(axis='y',direction='in',pad=-20, colors='k')
    par1.yaxis.label.set_color('k')

    # Turning off redundant axis and limiting horizontals
    host.set_xlim(0, 3000)

    plt.title(f"Integral parameters of SA and SST {meshQuality} mesh")
    plt.savefig(f'{workingPATH}integralParameters_SAandSST{meshQuality}
                                        .png', bbox_inches="tight", dpi
                                        =300)
    plt.cla()
    plt.clf()
    plt.close()

print("DONE: postProcessing")
```

B.2 plot_gridIndependence.py

```
import matplotlib.pyplot as plt
from pyOF_case import label,nu,Cx,Cy,onWall,Ux,T,Nx,Ny,Tplus
from pyOF_case2 import label2,Cx2,Cy2,onWall2,Ux2,T2,Nx2,Ny2,Tplus2
from pyOF_case3 import label3,Cx3,Cy3,onWall3,Ux3,T3,Nx3,Ny3,Tplus3
from pyOF_case4 import label4,Cx4,Cy4,onWall4,Ux4,T4,Nx4,Ny4,Tplus4
from pyOF_case5 import label5,Cx5,Cy5,onWall5,Ux5,T5,Nx5,Ny5,Tplus5
from pyOF_case6 import label6,Cx6,Cy6,onWall6,Ux6,T6,Nx6,Ny6,Tplus6
import scipy.signal as scp
import numpy as np
import _tools as tls

workingPATH = './'

h_fine= np.sqrt( np.mean(Cx[1:,:,0]  - Cx[:-1,:,0])  * np.mean(Cy[:,1:,
                                    0]  - Cy[:,:-1,0]) )
h_mid=  np.sqrt( np.mean(Cx3[1:,:,0] - Cx3[:-1,:,0]) * np.mean(Cy3[:,1:
                                    ,0] - Cy3[:,:-1,0]))
h_coarse= np.sqrt(np.mean(Cx5[1:,:,0] - Cx5[:-1,:,0])* np.mean(Cy5[:,1:
                                    ,0] - Cy5[:,:-1,0]))

r_midFine= h_mid/h_fine             #r21
r_coarseMid= h_coarse/h_mid         #r32

offset=1.65
y=Cy*1000
Cx, Cy = tls.transform(Cx,Cy)
Cx = (Cx+offset)*1000
Cx2, Cy2 = tls.transform(Cx2,Cy2)
Cx2 = (Cx2+offset)*1000
Cx3, Cy3 = tls.transform(Cx3,Cy3)
Cx3 = (Cx3+offset)*1000
Cx4, Cy4 = tls.transform(Cx4,Cy4)
Cx4 = (Cx4+offset)*1000
Cx5, Cy5 = tls.transform(Cx5,Cy5)
Cx5 = (Cx5+offset)*1000
Cx6, Cy6 = tls.transform(Cx6,Cy6)
Cx6 = (Cx6+offset)*1000

print(np.argwhere(np.isnan(Ux)))
print(np.argwhere(np.isnan(Ux2)))
print(np.argwhere(np.isnan(Ux3)))
print(np.argwhere(np.isnan(Ux4)))
print(np.argwhere(np.isnan(Ux5)))
```

```
print(np.argwhere(np.isnan(Ux6)))

####                  ####
####    Plotting      ####
####                  ####

plt.figure(figsize=(9,7))
plt.title('First off-wall $ _y^+$')
plt.plot(Cx[::10,0,0],  abs(scp.medfilt( (Cy[::10,1,0]-Cy[::10,0,0])  *
                                    onWall[::10,0]/nu )),'-', color="r"
                                     , label= f'{label}', alpha=1)
plt.plot(Cx2[::10,0,0], abs(scp.medfilt( (Cy2[::10,1,0]-Cy2[::10,0,0])*
                                    onWall2[::10,0]/nu)),'-', color="g"
                                     , label= f'{label2}',alpha=1)
plt.plot(Cx3[::5,0,0],  abs(scp.medfilt( (Cy3[::5,1,0]-Cy3[::5,0,0])  *
                                    onWall3[::5,0]/nu )),'--', color="r
                                    ", label= f'{label3}',alpha=.85)
plt.plot(Cx4[::5,0,0],  abs(scp.medfilt( (Cy4[::5,1,0]-Cy4[::5,0,0])  *
                                    onWall4[::5,0]/nu )),'--', color="g
                                    ", label= f'{label4}',alpha=.85)
plt.plot(Cx5[::5,0,0],  abs(scp.medfilt( (Cy5[::5,1,0]-Cy5[::5,0,0])  *
                                    onWall5[::5,0]/nu )),'-.', color="r
                                    ", label= f'{label5}',alpha=.75)
plt.plot(Cx6[::5,0,0],  abs(scp.medfilt( (Cy6[::5,1,0]-Cy6[::5,0,0])  *
                                    onWall6[::5,0]/nu )),'-.', color="g
                                    ", label= f'{label6}',alpha=.75)
plt.legend(loc='best')
plt.subplots_adjust(bottom=0.18)
plt.ylim(-0.5,2.5)
plt.ylabel('[$-$]')
plt.xlabel('x [m]')
plt.xticks(rotation=45)
plt.grid(True, which='both')
plt.savefig(f'{workingPATH}turbulent_Delta1OffWall.png', bbox_inches="
                                    tight", dpi=300)
plt.cla()
plt.clf()
plt.close()


### VALIDATION
for stat in [596,2505]:

    ### Velocity Ux GCI
    if stat==596:
        height= np.array(range(5,40,2))/1000
```

```
    elif stat==2505:
        height= np.array(range(14,144,10))/1000
    errorArray= np.zeros((len(height),3)) #indexx,indexy,error%
    errorArray2= np.zeros_like(errorArray)
    errorArray3= np.zeros_like(errorArray)
    errorArray4= np.zeros_like(errorArray)
    errorArray5= np.zeros_like(errorArray)
    errorArray6= np.zeros_like(errorArray)
    for i in range(len(height)):
        indexX_fine,indexY_fine =tls.get_indexes(stat,height[i],Cx,Cy)
        indexX_mid,indexY_mid =tls.get_indexes(stat,height[i],Cx3,Cy3)
        indexX_coarse,indexY_coarse =tls.get_indexes(stat,height[i],Cx5
                                            ,Cy5)

        ### SST
        phi_fineSST =Ux[indexX_fine,indexY_fine,0]
        phi_midSST =Ux3[indexX_mid,indexY_mid,0]
        phi_coarseSST =Ux5[indexX_coarse,indexY_coarse,0]
        (gci_midFineSST,gci_coarseMidSST) =tls.Get_DiscretizationError(
                                            r_midFine,r_coarseMid,
                                            phi_fineSST,phi_midSST,
                                            phi_coarseSST)
        errorArray3[i,:] =[Cy3[indexX_mid,indexY_mid,0],phi_midSST,
                                            phi_midSST*gci_midFineSST]
        errorArray5[i,:] =[Cy5[indexX_coarse,indexY_coarse,0],
                                            phi_coarseSST,
                                            phi_coarseSST*
                                            gci_coarseMidSST]

        ### SA
        phi_fineSA =Ux2[indexX_fine,indexY_fine,0]
        phi_midSA =Ux4[indexX_mid,indexY_mid,0]
        phi_coarseSA =Ux6[indexX_coarse,indexY_coarse,0]
        (gci_midFineSA,gci_coarseMidSA) =tls.Get_DiscretizationError(
                                            r_midFine,r_coarseMid,
                                            phi_fineSA,phi_midSA,
                                            phi_coarseSA)
        errorArray4[i,:] =[Cy4[indexX_mid,indexY_mid,0],phi_midSA,
                                            phi_midSA*gci_midFineSA]
        errorArray6[i,:] =[Cy6[indexX_coarse,indexY_coarse,0],
                                            phi_coarseSA, phi_coarseSA*
                                            gci_coarseMidSA]

    plt.figure(figsize=(9,7))
    plt.title('$U_x$ vs $y$')
```

```
    plt.plot(Cy[indexX_fine,:,0], Ux[indexX_fine,:,0],'--', color="
                                        deeppink", label= f'{label} at
                                        s={str(round(Cx[indexX_fine,0,0
                                        ],3))} mm',alpha=.75)
    plt.plot(Cy2[indexX_fine,:,0], Ux2[indexX_fine,:,0],'--', color="
                                        seagreen", label= f'{label2} at
                                         s={str(round(Cx2[indexX_fine,0
                                        ,0],3))} mm',alpha=.75)

    plt.plot(Cy3[indexX_mid,:,0], Ux3[indexX_mid,:,0],':', color="
                                        maroon", label= f'{label3} at s
                                        ={str(round(Cx3[indexX_mid,0,0
                                        ],3))} mm',alpha=.75)
    plt.plot(Cy4[indexX_mid,:,0], Ux4[indexX_mid,:,0],':', color="g",
                                        label= f'{label4} at s={str(
                                        round(Cx4[indexX_mid,0,0],3))}
                                        mm',alpha=.75)
    eb3= plt.errorbar(errorArray3[:,0],errorArray3[:,1],yerr=
                                        errorArray3[:,2],fmt=' ',ecolor
                                        ='maroon',capsize=3,capthick=3,
                                        elinewidth=1)
    eb3[-1][0].set_linestyle(':')
    eb4= plt.errorbar(errorArray4[:,0],errorArray4[:,1],yerr=
                                        errorArray4[:,2],fmt=' ',ecolor
                                        ='g',capsize=3,capthick=3,
                                        elinewidth=1)
    eb4[-1][0].set_linestyle(':')

    plt.plot(Cy5[indexX_coarse,:,0], Ux5[indexX_coarse,:,0],'-.', color
                                        ="r", label= f'{label5} at s={
                                        str(round(Cx5[indexX_coarse,0,0
                                        ],3))} mm',alpha=1)
    plt.plot(Cy6[indexX_coarse,:,0], Ux6[indexX_coarse,:,0],'-.', color
                                        ="lime", label= f'{label6} at s
                                        ={str(round(Cx6[indexX_coarse,0
                                        ,0],3))} mm',alpha=1)
    eb5= plt.errorbar(errorArray5[:,0],errorArray5[:,1],yerr=
                                        errorArray5[:,2],fmt=' ',ecolor
                                        ='r',capsize=3,capthick=3,
                                        elinewidth=1)
    eb5[-1][0].set_linestyle('-.')
    eb6= plt.errorbar(errorArray6[:,0],errorArray6[:,1],yerr=
                                        errorArray6[:,2],fmt=' ',ecolor
                                        ='lime',capsize=3,capthick=3,
                                        elinewidth=1)
    eb6[-1][0].set_linestyle('-.')
```

```
plt.legend(loc='best')
plt.subplots_adjust(bottom=0.18)
plt.ylabel('$[m/s]$')
if stat==596:
    plt.xlim(0,0.04)
elif stat==2505:
    plt.xlim(0,0.14)
plt.xlabel('$[m]$')
plt.xticks(rotation=45)
plt.grid(True, which='both')
plt.savefig(f'{workingPATH}Us-validation{stat}.png', bbox_inches="
                                    tight", dpi=300)
plt.cla()
plt.clf()
plt.close()


### Temperature T GCI
if stat==596:
    height= np.array(range(5,40,2))/1000
elif stat==2505:
    height= np.array(range(11,141,10))/1000
errorArray= np.zeros((len(height),3)) #indexx,indexy,error%
errorArray2= np.zeros_like(errorArray)
errorArray3= np.zeros_like(errorArray)
errorArray4= np.zeros_like(errorArray)
errorArray5= np.zeros_like(errorArray)
errorArray6= np.zeros_like(errorArray)
for i in range(len(height)):
    indexX_fine,indexY_fine =tls.get_indexes(stat,height[i],Cx,Cy)
    indexX_mid,indexY_mid =tls.get_indexes(stat,height[i],Cx3,Cy3)
    indexX_coarse,indexY_coarse =tls.get_indexes(stat,height[i],Cx5
                                    ,Cy5)

    ### SST
    phi_fineSST =    Tplus[indexX_fine,indexY_fine,0]*onWall[
                                    indexX_fine,1]
    phi_midSST =     Tplus3[indexX_mid,indexY_mid,0]*onWall3[
                                    indexX_mid,1]
    phi_coarseSST = Tplus5[indexX_coarse,indexY_coarse,0]*onWall5[
                                    indexX_coarse,1]
    (gci_midFineSST,gci_coarseMidSST) =tls.Get_DiscretizationError(
                                    r_midFine,r_coarseMid,
                                    phi_fineSST,phi_midSST,
                                    phi_coarseSST)
```

```
        errorArray3[i,:] =[Cy3[indexX_mid,indexY_mid,0],phi_midSST,
                                            phi_midSST*gci_midFineSST]
        errorArray5[i,:] =[Cy5[indexX_coarse,indexY_coarse,0],
                                            phi_coarseSST,
                                            phi_coarseSST*
                                            gci_coarseMidSST]

        ### SA
        phi_fineSA =    Tplus2[indexX_fine,indexY_fine,0]*onWall2[
                                            indexX_fine,1]
        phi_midSA =     Tplus4[indexX_mid,indexY_mid,0]*onWall4[
                                            indexX_mid,1]
        phi_coarseSA = Tplus6[indexX_coarse,indexY_coarse,0]*onWall6[
                                            indexX_coarse,1]
        (gci_midFineSA,gci_coarseMidSA) =tls.Get_DiscretizationError(
                                            r_midFine,r_coarseMid,
                                            phi_fineSA,phi_midSA,
                                            phi_coarseSA)
        errorArray4[i,:] =[Cy4[indexX_mid,indexY_mid,0],phi_midSA,
                                            phi_midSA*gci_midFineSA]
        errorArray6[i,:] =[Cy6[indexX_coarse,indexY_coarse,0],
                                            phi_coarseSA, phi_coarseSA*
                                            gci_coarseMidSA]

    plt.figure(figsize=(9,7))
    plt.title('$ $ vs $y$')
    plt.plot(Cy[indexX_fine,:,0], abs(Tplus[indexX_fine,:,0]*onWall[
                                        indexX_fine,1]),'--', color="
                                        deeppink", label= f'{label} at
                                        s={str(round(Cx[indexX_fine,0,0
                                        ],3))} mm',alpha=.75)
    plt.plot(Cy2[indexX_fine,:,0], abs(Tplus2[indexX_fine,:,0]*onWall2[
                                        indexX_fine,1]),'--', color="
                                        seagreen", label= f'{label2} at
                                         s={str(round(Cx2[indexX_fine,0
                                        ,0],3))} mm',alpha=.75)

    plt.plot(Cy3[indexX_mid,:,0], abs(Tplus3[indexX_mid,:,0]*onWall3[
                                        indexX_mid,1]),':', color="
                                        maroon", label= f'{label3} at s
                                        ={str(round(Cx3[indexX_mid,0,0
                                        ],3))} mm',alpha=.75)
```

```
    plt.plot(Cy4[indexX_mid,:,0], abs(Tplus4[indexX_mid,:,0]*onWall4[
                                        indexX_mid,1]),':', color="g",
                                        label= f'{label4} at s={str(
                                        round(Cx4[indexX_mid,0,0],3))}
                                        mm',alpha=.75)
    eb3= plt.errorbar(errorArray3[:,0],errorArray3[:,1],yerr=
                                        errorArray3[:,2],fmt=' ',ecolor
                                        ='maroon',capsize=3,capthick=3,
                                        elinewidth=1)
    eb3[-1][0].set_linestyle(':')
    eb4= plt.errorbar(errorArray4[:,0],errorArray4[:,1],yerr=
                                        errorArray4[:,2],fmt=' ',ecolor
                                        ='g',capsize=3,capthick=3,
                                        elinewidth=1)
    eb4[-1][0].set_linestyle(':')

    plt.plot(Cy5[indexX_coarse,:,0], abs(Tplus5[indexX_coarse,:,0]*
                                        onWall5[indexX_coarse,1]),'-.',
                                         color="r", label= f'{label5}
                                        at s={str(round(Cx5[
                                        indexX_coarse,0,0],3))} mm',
                                        alpha=1)
    plt.plot(Cy6[indexX_coarse,:,0], abs(Tplus6[indexX_coarse,:,0]*
                                        onWall5[indexX_coarse,1]),'-.',
                                         color="lime", label= f'{label6
                                        } at s={str(round(Cx6[
                                        indexX_coarse,0,0],3))} mm',
                                        alpha=1)
    eb5= plt.errorbar(errorArray5[:,0],errorArray5[:,1],yerr=
                                        errorArray5[:,2],fmt=' ',ecolor
                                        ='r',capsize=3,capthick=3,
                                        elinewidth=1)
    eb5[-1][0].set_linestyle('-.')
    eb6= plt.errorbar(errorArray6[:,0],errorArray6[:,1],yerr=
                                        errorArray6[:,2],fmt=' ',ecolor
                                        ='lime',capsize=3,capthick=3,
                                        elinewidth=1)
    eb6[-1][0].set_linestyle('-.')

    plt.legend(loc='best')
    plt.subplots_adjust(bottom=0.18)
    plt.ylabel('$(T-T_w)/( T _  -T_w)$')
    if stat==596:
        plt.xlim(0,0.04)
    elif stat==2505:
        plt.xlim(0,0.14)
```

```
    plt.xlabel('$[m]$')
    plt.xticks(rotation=45)
    plt.grid(True, which='both')
    plt.savefig(f'{workingPATH}Theta-validation{stat}.png', bbox_inches
                                        ="tight", dpi=300)
    plt.cla()
    plt.clf()
    plt.close()

print("DONE: plot_gridIndependence ")
```

B.3 __tools.py

```
import numpy as np
import matplotlib.pyplot as plt
from math import sqrt
np.seterr(divide='ignore', invalid='ignore')

def transform(x, y):
    Nx, Ny, Nz = x.shape
    s = np.zeros_like(x)
    n = np.zeros_like(y)

    s[0, :, :] = x[0, :, :]

    for ii in range(1,Nx):
        for jj in range(Ny):
            for kk in range(Nz):
                s[ii,jj,kk] = s[ii-1, jj, kk] + sqrt((x[ii,jj,kk] - x[
                                                    ii-1,jj,kk])**2 + (
                                                    y[ii,jj,kk] - y[ii-
                                                    1,jj,kk])**2)
                n[ii,jj,kk] = sqrt((x[ii,jj,kk] - x[ii,0,kk])**2 + (y[
                                                    ii,jj,kk] - y[ii,0,
                                                    kk])**2)
    return s, n

def calculate_3D_gradient(x=None, y=None, z=None, f=None):
    (nx, ny, nz) = x.shape

    J = np.zeros((nx, ny, nz, 3), dtype=f.dtype)
```

```
# First order boundary approximations O(h)
# Left, Lower, Front Boundary
J[0, :, :, 0] = (f[1, :, :] - f[0, :, :]) / (x[1, :, :] - x[0, :, :
                                            ])
J[1, :, :, 0] = (f[2, :, :] - f[0, :, :]) / (x[2, :, :] - x[0, :, :
                                            ])
J[:, 0, :, 1] = (f[:, 1, :] - f[:, 0, :]) / (y[:, 1, :] - y[:, 0, :
                                            ])
J[:, :, 0, 2] = (f[:, :, 1] - f[:, :, 0]) / (z[:, :, 1] - z[:, :, 0
                                            ])
J[:, :, 1, 2] = (f[:, :, 2] - f[:, :, 0]) / (z[:, :, 2] - z[:, :, 0
                                            ])
# First order boundary approximations O(h)
# Right, Upper, Back Boundary
J[nx-1, :, :, 0] = (f[nx-1, :, :] - f[nx-2, :, :]) / (x[nx-1, :, :]
                                             - x[nx-2, :, :])
J[nx-2, :, :, 0] = (f[nx-1, :, :] - f[nx-3, :, :]) / (x[nx-1, :, :]
                                             - x[nx-3, :, :])
J[:, ny-1, :, 1] = (f[:, ny-1, :] - f[:, ny-2, :]) / (y[:, ny-1, :]
                                             - y[:, ny-2, :])
J[:, :, nz-1, 2] = (f[:, :, nz-1] - f[:, :, nz-2]) / (z[:, :, nz-1]
                                             - z[:, :, nz-2])
J[:, :, nz-2, 2] = (f[:, :, nz-1] - f[:, :, nz-3]) / (z[:, :, nz-1]
                                             - z[:, :, nz-3])

# Second order boundary approximations
# Exactly Second order for uniform spacing O(h_{i} h_{i-1})
fx2p = f[4:nx, :, :]
fxp = f[3:nx-1, :, :]
fcx = f[2:nx-2, :, :]
fxm = f[1:nx-3, :, :]
fx2m = f[0:nx-4, :, :]
dx = ((x[4:nx, :, :] - x[3:nx-1, :, :]) + \
      (x[3:nx-1, :, :] - x[2:nx-2, :, :]) +\
      (x[2:nx-2, :, :] - x[1:nx-3, :, :]) +\
      (x[1:nx-3, :, :] - x[0:nx-4, :, :])) / 4
J[2:nx-2, :, :, 0] = (-fx2p + 8*fxp - 8*fxm + fx2m)/(12*dx)

fyp = f[:, 2:ny, :]
fcy = f[:, 1:ny-1, :]
fym = f[:, 0:ny-2, :]
hy = (y[:, 2:ny, :] - y[:, 1:ny-1, :])
ay = hy / (y[:, 1:ny-1, :] - y[:, 0:ny-2, :])
dy = hy * (1 + ay)
J[:, 1:ny-1, :, 1] = (fyp - ay*ay*fym - (1 - ay*ay)*fcy)/dy
```

```
    fz2p = f[:, :, 4:nz]
    fzp = f[:, :, 3:nz-1]
    fcz = f[:, :, 2:nz-2]
    fzm = f[:, :, 1:nz-3]
    fz2m = f[:, :, 0:nz-4]
    dz = ((z[:, :, 4:nz] - z[:, :, 3:nz-1]) + \
          (z[:, :, 3:nz-1] - z[:, :, 2:nz-2]) +\
          (z[:, :, 2:nz-2] - z[:, :, 1:nz-3]) +\
          (z[:, :, 1:nz-3] - z[:, :, 0:nz-4])) / 4
    J[:, :, 2:nz-2, 2] = (-fz2p + 8*fzp - 8*fzm + fz2m)/(12*dz)
    return J

def calculate_2D_gradient(x=None, y=None, f=None):
    (nx, ny, nz) = x.shape

    J = np.zeros((nx, ny, 1, 2), dtype=f.dtype)
    # First order boundary approximations O(h)
    # Left, Lower, Front Boundary
    J[0, :, :, 0] = (f[1, :, :] - f[0, :, :]) / (x[1, :, :] - x[0, :, :
                                                 ])
    J[1, :, :, 0] = (f[2, :, :] - f[0, :, :]) / (x[2, :, :] - x[0, :, :
                                                 ])
    J[:, 0, :, 1] = (f[:, 1, :] - f[:, 0, :]) / (y[:, 1, :] - y[:, 0, :
                                                 ])
    # First order boundary approximations O(h)
    # Right, Upper, Back Boundary
    J[nx-1, :, :, 0] = (f[nx-1, :, :] - f[nx-2, :, :]) / (x[nx-1, :, :]
                                                  - x[nx-2, :, :])
    J[nx-2, :, :, 0] = (f[nx-1, :, :] - f[nx-3, :, :]) / (x[nx-1, :, :]
                                                  - x[nx-3, :, :])
    J[:, ny-1, :, 1] = (f[:, ny-1, :] - f[:, ny-2, :]) / (y[:, ny-1, :]
                                                  - y[:, ny-2, :])

    # Fourth order streamwise approximations, O(h^4)
    fx2p = f[4:nx, :, :]
    fxp = f[3:nx-1, :, :]
    fcx = f[2:nx-2, :, :]
    fxm = f[1:nx-3, :, :]
    fx2m = f[0:nx-4, :, :]
    dx = ((x[4:nx, :, :] - x[3:nx-1, :, :]) + \
          (x[3:nx-1, :, :] - x[2:nx-2, :, :]) +\
          (x[2:nx-2, :, :] - x[1:nx-3, :, :]) +\
          (x[1:nx-3, :, :] - x[0:nx-4, :, :])) / 4
    J[2:nx-2, :, :, 0] = (-fx2p + 8*fxp - 8*fxm + fx2m)/(12*dx)
    # Second order boundary approximations
    # Exactly Second order for uniform spacing O(h_{i} h_{i-1})
```

```
    fyp = f[:, 2:ny, :]
    fcy = f[:, 1:ny-1, :]
    fym = f[:, 0:ny-2, :]
    hy = (y[:, 2:ny, :] - y[:, 1:ny-1, :])
    ay = hy / (y[:, 1:ny-1, :] - y[:, 0:ny-2, :])
    dy = hy * (1 + ay)
    J[:, 1:ny-1, :, 1] = (fyp - ay*ay*fym - (1 - ay*ay)*fcy)/dy
    return J

def gradient_of(Cx: np.ndarray, Cy: np.ndarray, Cz: np.ndarray, f: np.
                                    ndarray):
    """
    gradient_of(Cx,Cy,Cz,parameter)  return: parameter gradient with
                                        shape ((nx, ny, nz, 3))
    """
    (nx,ny,nz) = f.shape
    if nz == 2:
        gradiente = calculate_2D_gradient(Cx[:,:,:1],Cy[:,:,:1],f[:,:,:
                                            1])
        gradiente = np.append(gradiente, np.zeros_like(gradiente[:,:,:,
                                            :1]),axis=3) # dummy values
                                             for dZ
        gradiente = np.append(gradiente, gradiente,axis=2)
    elif nz > 2:
        gradiente = calculate_3D_gradient(Cx,Cy,Cz,f)

    return gradiente

def fistOffWall_DeltaPlus(Cx,Cy,Cz,Ux,nu,Temp,Pr=0.71):
    ### Cx,Cy,Cz=Cells spatial coordinates, Ux=streamwise velocity, nu=
                                        kinematic viscosity
    pOut= 2 # [ u_tau dThetady0]
    (Nx, Ny, Nz) = Cx.shape
    Output = np.zeros((Nx,pOut))
    z=0

    ### Preparing arrays
    u_tau, temp_normalized, dThetady0 = np.zeros(Nx), np.zeros((Nx,Ny,
                                        Nz)), np.zeros(Nx)
    dUx_d = gradient_of(Cx,Cy,Cz,Ux)

    ### For each X's stations vertex:
    for x in range(Nx):

        ### Skin Friction Velocity
        dudy0 = np.mean(dUx_d[x,0:4,z,1])
```

```
        if dudy0 > 0 :
                u_tau[x] = np.sqrt(nu*dudy0)                    # sqrt(
                                                        nu*(du/dy)) @y=0
        elif dudy0 < 0 :
                u_tau[x] = -1*np.sqrt(nu*(abs(dudy0)))          # -sqrt
                                                        (nu*(du/dy)) @y=0

        ### Thermal calculations
        Temp_inf = np.mean(Temp[x,Ny-2:Ny,:])
        Temp_0 = np.mean(Temp[x,0,:])
        temp_normalized[x,:,:] = (Temp[x,:,:]-Temp_0)/(Temp_inf-Temp_0)

    dThetady = gradient_of(Cx,Cy,Cz,temp_normalized)

    ### For each X's stations vertex:
    for x in range(Nx):
        dThetady0 = np.mean(dThetady[x,0:4,z,1])

    Output[:,0]= u_tau
    Output[:,1]= (nu/Pr)*dThetady0/u_tau

    return Output

### Procedure for Estimation of Discretization Error
def get_indexes(toFindx,toFindy,Cx: np.ndarray,Cy: np.ndarray):
    '''
        get_indexes: Find the indexes in Cx & Cy for ONE Point
        * toFindx: aimed distance in [mm] to find at the wall locations
                                            , following Baskaran
                                            Experiment stations
                                            reference.
        * toFindy: aimed height in [m] to find in the wall-normal
                                            direction.
        * Cx: 3D Array to search the locations. In [mm]
        * Cy: 3D Array to search the wall-normal heights. In [m]
        * loc: Index in Cx that matches the stations desired.
        * height: Index in Cy that matches the height desired.
    '''
    (nx,ny,_) = Cx.shape
    for x in range(nx):                 # Iterating through all
                                        possible stations
        if  abs(Cx[x,0,0] - toFindx) <= 1e-9:   # If the current x-
                                            station is equals to the
                                            desired station...
            loc = x                             # Take that index
            break
```

```
        elif Cx[x,0,0] > toFindx:                # If the current x-
                                                 station is higher than the
                                                 desired station... Take the
                                                  nearest index
            if abs(toFindx-Cx[x-1,0,0]) < abs(Cx[x,0,0] -toFindx):
                loc = x-1
            elif abs(toFindx-Cx[x-1,0,0]) > abs(Cx[x,0,0] -toFindx):
                loc = x
            break
    for y in range(ny):
        if  abs(Cy[loc,y,0] - toFindy) <= 1e-9:    # If the current
                                                 height is equals to the
                                                 desired height...
            height = y                                 # Take that index
            break
        elif Cy[loc,y,0] > toFindy:                    # If the current
                                                 height is higher than the
                                                 desired height... Take the
                                                 nearest index
            if abs(toFindy-Cy[loc,y-1,0]) < abs(Cy[loc,y,0] -toFindy):
                    height = y-1
            elif abs(toFindy-Cy[loc,y-1,0]) > abs(Cy[loc,y,0] -toFindy)
                                                      :
                    height = y
            break

    return loc , height

def Get_DiscretizationError(r21, r32, phi1, phi2, phi3):
    eps21= phi2 - phi1
    eps32= phi3 - phi2
    p= abs(np.log(abs(eps32/eps21)))/np.log(r21)
    s= 1*np.sign(eps32/eps21)

    for i in range(5) :
        q= np.log((r21**p - s)/(r32**p - s))
        p= abs(np.log(abs(eps32/eps21))+q)/np.log(r21)

    phiExt21= (r21**p*phi1 - phi2)/(r21**p-1)        # Extrapolated
                                                 values
    phiExt32= (r32**p*phi2 - phi3)/(r32**p-1)
    ea21=abs((phi1-phi2)/phi1)                       # Approximate
                                                 relative error
    ea32=abs((phi2-phi3)/phi2)
    eExt21= abs((phiExt21-phi1)/phiExt21)            # Extrapolated
                                                 relative error
```

```
    eExt32= abs((phiExt32-phi2)/phiExt32)
    gci21= (1.25*ea21)/(r21**p-1)                    # Fine-grid
                                              convergence index
    gci32= (1.25*ea32)/(r32**p-1)

    return gci21 , gci32
```

B.4 __openfoam__utilities.py

```
import numpy as np
import vtk
from pyevtk.hl import pointsToVTK
from vtk.util import numpy_support as VN
from vtk.util.numpy_support import vtk_to_numpy

import matplotlib.pyplot as plt

def permute_along_y_axis(f, p):
    for ii, pvec in enumerate(p):
        f[ii, :, :] = f[ii, pvec, :]


def read_pvtu(src):
    reader = vtk.vtkXMLPUnstructuredGridReader()
    reader.SetFileName(src)
    reader.Update()

    vtkdata = reader.GetOutput()
    celldata = vtkdata.GetCellData()
    pointdata = vtkdata.GetPointData()
    points = vtkdata.GetPoints()
    cell_connectivity = vtkdata.GetCells()

    coordinates = vtk_to_numpy(points.GetData())

    x = coordinates[:, 0]
    y = coordinates[:, 1]
    z = coordinates[:, 2]
    ind = np.lexsort((z, y, x))
    x = x[ind]
    y = y[ind]
    z = z[ind]
```

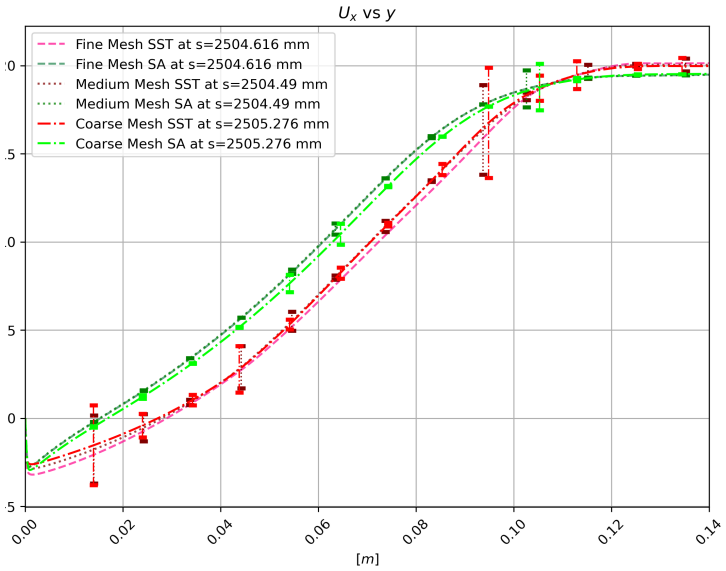
```
    arrays = {}

    for ii in range(pointdata.GetNumberOfArrays()):
        name = pointdata.GetArray(ii).GetName()
        data = vtk_to_numpy(pointdata.GetArray(ii))
        components = pointdata.GetArray(ii).GetNumberOfComponents()
        if components > 1:
            Dx = data[:, 0][ind]
            Dy = data[:, 1][ind]
            Dz = data[:, 2][ind]

            arrays[f"{name}_x"] = Dx
            arrays[f"{name}_y"] = Dy
            arrays[f"{name}_z"] = Dz
        else:
            arrays[name] = data[ind]
    return x, y, z, arrays


def read_vtk(src, Nx, Ny, Nz):
    reader = vtk.vtkUnstructuredGridReader()
    reader.SetFileName(src)
    reader.ReadAllFieldsOn()
    reader.ReadAllVectorsOn()
    reader.ReadAllScalarsOn()
    reader.Update()

    vtkdata = reader.GetOutput()
    celldata = vtkdata.GetCellData()
    pointdata = vtkdata.GetPointData()
    points = vtkdata.GetPoints()
    cell_connectivity = vtkdata.GetCells()

    coordinates = vtk_to_numpy(points.GetData())

    x = coordinates[:, 0]
    y = coordinates[:, 1]
    z = coordinates[:, 2]
    ind = np.lexsort((z, y, x))
    x = x[ind].reshape((Nx, Ny, Nz))
    y = y[ind].reshape((Nx, Ny, Nz))
    z = z[ind].reshape((Nx, Ny, Nz))

    permutation_vectors = [[yid for yid in range(Ny)] for _ in range(Nx
                                                                     )]
```

```
    for xid in range(Nx):
        permutation_vectors[xid] = np.argsort(y[xid, :, 0])

    permute_along_y_axis(x, permutation_vectors)
    permute_along_y_axis(y, permutation_vectors)
    permute_along_y_axis(z, permutation_vectors)
    arrays = {}

    for ii in range(pointdata.GetNumberOfArrays()):
        name = pointdata.GetArray(ii).GetName()
        data = vtk_to_numpy(pointdata.GetArray(ii))
        components = pointdata.GetArray(ii).GetNumberOfComponents()
        if components > 1:
            Dx = data[:, 0][ind].reshape((Nx, Ny, Nz))
            Dy = data[:, 1][ind].reshape((Nx, Ny, Nz))
            Dz = data[:, 2][ind].reshape((Nx, Ny, Nz))
            permute_along_y_axis(Dx, permutation_vectors)
            permute_along_y_axis(Dy, permutation_vectors)
            permute_along_y_axis(Dz, permutation_vectors)
            arrays[f"{name}_x"] = Dx
            arrays[f"{name}_y"] = Dy
            arrays[f"{name}_z"] = Dz
        else:
            tmp = data[ind].reshape((Nx, Ny, Nz))
            permute_along_y_axis(tmp, permutation_vectors)
            arrays[name] = tmp


    return x, y, z, arrays


def translate_vtk_to_hdf5(src, dst):
    raise NotImplementedError


if __name__ == "__main__":
    path = "/Users/davidpaeres/My Documents/ARAYA/OF/VTK_SA/
                                        flatPlateSA_100000.vtk"
    target = "/Users/davidpaeres/Desktop/flatPlateSA"
```

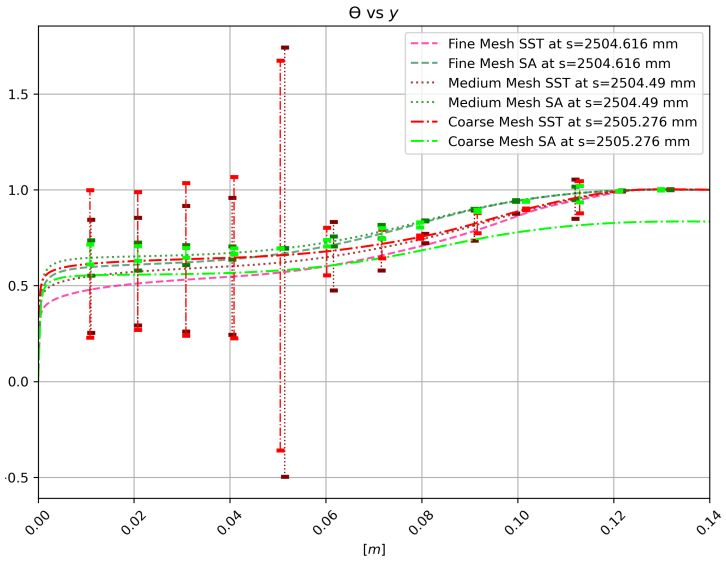
Appendix C
Grid Sensitivity Study

Figure C.1 depicts the results of the grid independence test performed over the curved hill domain. Some clarifications: (i) we have picked up a highly distorted flow, namely, the flow separation zone, (ii) additionally, we have explored the performance of grid point distribution in zones where the boundary layer is attached. Turning to fig. C.1 (a) of the U_s profiles, it can be seen a clear convergence of the three types of meshes (coarse, medium and fine) in the SA and SST models; in particular, there is a high level of similarity between the medium and fine mesh. This confirms that the refinement strategy has been adequate. By contrasting both models, the SST model predicts a much thicker separation bubble and significant discrepancies are visualized inside the momentum boundary layer. It is important to mention that the SST model by Menter (Menter 1994) considers a further improvement to the eddy viscosity model and is based on the idea of the Johnson–King model. It determines that the transport of the main turbulent shear stresses is critical in the simulations of strong APG flows. Specifically, the Menter SST turbulence model has been developed to outperform in turbulent boundary layer flows subject to APG, with eventual separation. Unfortunately, in (Baskaran, Smits, and Joubert 1987), the streamwise velocity profiles were not measured at the separation bubble in order to validate our RANS results, and thus, further studies should be performed in the future to assess the SST model under flow detachment. The thermal distribution inside the temperature boundary layer is displayed in fig. C.1 (b). Similarly, the SST model underpredicts temperature (passive scalar) with respect to the SA model. Significant differences are observed in thermal results at each refinement level. However, the number of vertical points was increased from the coarse to the fine mesh by an x2.66 factor, and additionally, special care was taken in the near wall region of the fine mesh ($\Delta y^+ \approx 0.15$ in the separation bubble, or lower). At this point, we can only ensure that numerical predictions are error-free of grid resolution. Therefore, the numerical predictions' accuracy of the thermal field is warranted and the fine mesh is deemed appropriate for the goals of this study. The errors presented in figures C.1 and C.2 are regarding to fine-grid convergence index (GCI), where the latter figure is for an additional station $s \approx 596mm$ corresponding to the ZPG region before the curved hill. The errors shown are much lower in the ZPG station compared to the complicated station at the

separation zone of fig. C.1, supporting our conclusion on the mesh convergence. The estimation of discretization errors follows the procedure as described by Celik *et al.* (Celik, Ghia, Roache, and Freitas 2008), which is obtained from coarse mesh quality relative to medium mesh quality and medium mesh relative to fine mesh, respectively.

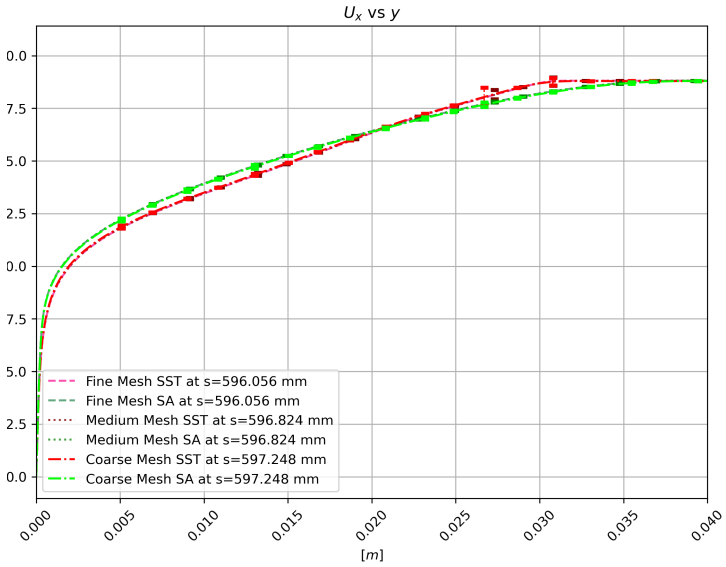


(a)

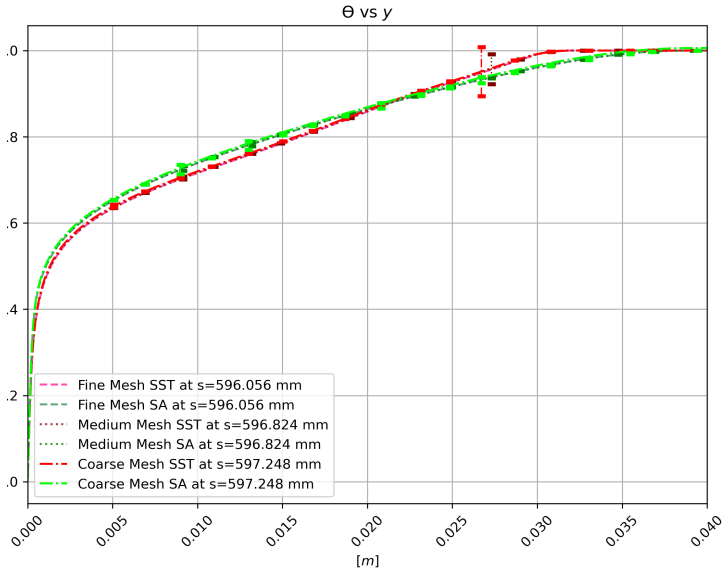


(b)

Figure C.1: Grid resolution independence assessment of: (a) streamwise velocity, $U_s[m/s]$, and (b) normalized temperature, $\overline{T} = (T - T_w)/(T_\infty - T_w)$, at the separation bubble.



(a)



(b)

Figure C.2: Grid resolution independence assessment of: (a) streamwise velocity, $U_s[m/s]$, and (b) normalized temperature, $\overline{T} = (T - T_w)/(T_\infty - T_w)$, at different streamwise locations.

Tables C.1 and C.2 unveil the momentum and displacement thickness results and relative errors between meshes' qualities at several streamwise stations. In general, integral values of the turbulent boundary layers, i.e. δ^* and θ, exhibit a tendency to converge as the mesh is refined. Looking at Table C.1, both SST and SA models had the highest discrepancy

among grids at $s \approx 1139mm$, with -6.82% and 8.15%, respectively. Table C.2 shows how outstanding and low the relative errors for the momentum thickness are, where the most elevated error for the SST cases is -4.31% in the FPG region ($s \approx 1596mm$). Meanwhile, for the SA cases is 4.34% at $s \approx 1139mm$. SA cases had more consistency and lower errors, even some reaching zero differences. Altogether, the relative errors are more than suitable for the presented assessment, favoring the momentum thickness compared to displacement thickness and SA slightly over SST for consistency. It is also observed that maximum discrepancies are concentrated at $s \approx 1139mm$ in both models and are within 30%. This zone, located around the first concave-convex surface intersection, seems to be one of the "hardest" to be simulated (aside from the flow separation bubble) since peaks on the C_f were found there. Table C.3 supplies a summary of the relative errors between different types of meshes at all streamwise stations. The low obtained values on the average relative errors ($\sim 2\%$) confirm the numerical convergence in the refinement process.

Table C.1: Relative error of the displacement thickness.

Stations [mm]	**596**	**710**	**867**	**1015**	**1139**	**1183**	**1345**	**1469**	**1596**	**1665**	**1730**	**1862**	**1990**
Experimental δ^* [mm]	5.3108	6.2568	9.3987	11.5946	4.6352	3.7230	2.2703	2.0000	1.9325	2.4730	2.3716	3.4189	7.0338
SST fine mesh δ^* [mm]	6.4351	7.6986	12.1624	15.2198	6.5256	4.7115	2.3776	2.0155	2.0727	2.2290	2.4768	3.4724	5.9165
SST medium mesh δ^* [mm]	6.3852	7.6428	12.1447	15.1597	6.0952	4.6767	2.3269	1.9677	2.0225	2.1779	2.4090	3.3711	5.8081
SST coarse mesh δ^* [mm]	6.3403	7.5589	12.0473	15.2093	6.5252	4.6499	2.2817	1.9098	1.9505	2.1029	2.3374	3.2521	5.5843
Rel. Error (fine to exp.)	19.14%	20.66%	25.64%	27.04%	33.88%	23.44%	4.62%	0.77%	7.01%	-10.38%	4.34%	1.55%	-17.26%
Rel. Error (medium to fine)	-0.78%	-0.73%	-0.15%	-0.40%	-6.82%	-0.74%	-2.16%	-2.40%	-2.45%	-2.32%	-2.78%	-2.96%	-1.85%
Rel. Error (coarse to medium)	-0.71%	-1.10%	-0.80%	0.33%	6.81%	-0.57%	-1.96%	-2.99%	-3.63%	-3.50%	-3.02%	-3.59%	-3.93%
Experimental δ^* [mm]	5.3108	6.2568	9.3987	11.5946	4.6352	3.7230	2.2703	2.0000	1.9325	2.4730	2.3716	3.4189	7.0338
SA fine mesh δ^* [mm]	6.1274	7.2940	11.3322	14.4704	6.3664	4.6301	2.3285	1.9750	2.0251	2.1694	2.3982	3.2684	5.3129
SA medium mesh δ^* [mm]	6.1019	7.2670	11.3463	14.3841	5.9822	4.6167	2.3026	1.9526	2.0026	2.1372	2.3769	3.2297	5.3106
SA coarse mesh δ^* [mm]	6.1584	7.3110	11.4325	14.6143	6.4907	4.6748	2.3088	1.9493	2.0002	2.1509	2.3815	3.2517	5.3431
Rel. Error (fine to exp.)	14.28%	15.31%	18.65%	22.07%	31.47%	21.72%	2.53%	-1.26%	4.68%	-13.08%	1.11%	-4.50%	-27.88%
Rel. Error (medium to fine)	-0.42%	-0.37%	0.12%	-0.60%	-6.22%	-0.29%	-1.12%	-1.14%	-1.12%	-1.49%	-0.89%	-1.19%	-0.04%
Rel. Error (coarse to medium)	0.92%	0.60%	0.76%	1.59%	8.15%	1.25%	0.27%	-0.17%	-0.12%	0.64%	0.19%	0.68%	0.61%

Table C.2: Relative error of the momentum thickness.

Stations [mm]	**596**	**710**	**867**	**1015**	**1139**	**1183**	**1345**	**1469**	**1596**	**1665**	**1730**	**1862**	**1990**
SST fine mesh θ [mm]	4.3102	5.0204	6.9113	8.0405	4.5770	3.6047	1.9612	1.6479	1.6661	1.7738	1.9503	2.6493	4.1432
SST medium mesh θ [mm]	4.2982	5.0054	6.9092	8.0269	4.4109	3.5962	1.9384	1.6272	1.6437	1.7511	1.9145	2.5915	4.0891
SST coarse mesh θ [mm]	4.2629	4.9495	6.8559	8.0036	4.5478	3.5571	1.8894	1.5681	1.5744	1.6807	1.8486	2.4937	3.9427
Rel. Error (fine to exp.)	6.45%	6.47%	4.46%	0.79%	15.57%	10.55%	-0.06%	-8.22%	-12.27%	-12.73%	-3.12%	1.51%	-0.87%
Rel. Error (medium to fine)	-0.28%	-0.30%	-0.03%	-0.17%	-3.69%	-0.23%	-1.17%	-1.27%	-1.36%	-1.29%	-1.86%	-2.20%	-1.31%
Rel. Error (coarse to medium)	-0.82%	-1.12%	-0.77%	-0.29%	3.06%	-1.09%	-2.56%	-3.70%	-4.31%	-4.10%	-3.50%	-3.85%	-3.64%
Experimental θ [mm]	4.0407	4.7058	6.6096	7.9771	3.9158	3.2434	1.9625	1.7892	1.8840	2.0151	2.0122	2.6097	4.1793
SA fine mesh θ [mm]	4.2706	4.9702	6.8606	8.1599	4.5825	3.6021	1.9310	1.6236	1.6401	1.7414	1.9069	2.5261	3.8347
SA medium mesh θ [mm]	4.2728	4.9717	6.8777	8.1362	4.4378	3.6099	1.9285	1.6236	1.6401	1.7335	1.9083	2.5151	3.8494
SA coarse mesh θ [mm]	4.2944	4.9844	6.9055	8.1997	4.6345	3.6316	1.9211	1.6079	1.6246	1.7312	1.8979	2.5165	3.8549
Rel. Error (fine to exp.)	5.53%	5.46%	3.73%	2.27%	15.69%	10.48%	-1.62%	-9.71%	-13.84%	-14.57%	-5.37%	-3.26%	-8.60%
Rel. Error (medium to fine)	0.05%	0.03%	0.25%	-0.29%	-3.21%	0.22%	-0.13%	0.00%	0.00%	-0.45%	0.07%	-0.44%	0.38%
Rel. Error (coarse to medium)	0.50%	0.26%	0.40%	0.78%	4.34%	0.60%	-0.39%	-0.97%	-0.95%	-0.14%	-0.54%	0.06%	0.14%

Table C.3: Summary of Relative Errors for the displacement/momentum thickness.

δ^*	Average error [%]	Maximum error [%]
SST	2.29	6.82
SA	1.19	8.15
θ	Average error [%]	Maximum error [%]
SST	1.84	3.69
SA	0.56	4.34

Appendix D
Post-processing Tools for Data Visualization

D.1 preScript.py

```
import glob
workingPath = '/Users/davidpaeres/Downloads/'
vts_directory = 'COLD/'
vts_groupName = 'PUVWT_1-565.326*'
isoValue= 2500.0
gltf_directory= 'Qcriterion2500_colorTemp/'
vts_FileNames = sorted(glob.glob(f'{workingPath}{vts_directory}{
                                    vts_groupName}'))


# trace generated using paraview version 5.10.1
#import paraview
#paraview.compatibility.major = 5
#paraview.compatibility.minor = 10

#### import the simple module from the paraview
from paraview.simple import *
#### disable automatic camera reset on 'Show'
paraview.simple._DisableFirstRenderCameraReset()

# create a new 'XML Structured Grid Reader'
pUVWT_1565326 = XMLStructuredGridReader(registrationName= vts_groupName
                                        , FileName=vts_FileNames)
pUVWT_1565326.PointArrayStatus = ['Q Criterion', 'Normalized Q
                                    Criterion', 'p', 'u', 'v', 'w', 'T'
                                    , "p'", "u'", "v'", "w'", "T'", 'Q1
                                    ', 'Q2', 'Q3', 'Q4', 'omega_x', '
                                    omega_y', 'omega_z', "omega_x'", "
                                    omega_y'", "omega_z'"]
```

```
# Properties modified on pUVWT_1565326
pUVWT_1565326.TimeArray = 'None'

# get active view
renderView1 = GetActiveViewOrCreate('RenderView')

# get the material library
materialLibrary1 = GetMaterialLibrary()

# get display properties
pUVWT_1565326Display = GetDisplayProperties(pUVWT_1565326, view=
                                    renderView1)

# get animation scene
animationScene1 = GetAnimationScene()

# update animation scene based on data timesteps
animationScene1.UpdateAnimationUsingDataTimeSteps()

# create a new 'Contour'
contour1 = Contour(registrationName='Contour1', Input=pUVWT_1565326)
contour1.ContourBy = ['POINTS', 'Normalized Q Criterion']
contour1.Isosurfaces = [-0.4689942755095545]
contour1.PointMergeMethod = 'Uniform Binning'

# show data in view
contour1Display = Show(contour1, renderView1, 'GeometryRepresentation')

# get color transfer function/color map for 'NormalizedQCriterion'
normalizedQCriterionLUT = GetColorTransferFunction('
                                    NormalizedQCriterion')
normalizedQCriterionLUT.RGBPoints = [-0.46899428963661194, 0.231373, 0.
                                    298039, 0.752941, -0.
                                    46896377205848694, 0.865003, 0.
                                    865003, 0.865003, -0.
                                    46893325448036194, 0.705882, 0.
                                    0156863, 0.14902]
normalizedQCriterionLUT.ScalarRangeInitialized = 1.0

# trace defaults for the display properties.
contour1Display.Representation = 'Surface'
contour1Display.ColorArrayName = ['POINTS', 'Normalized Q Criterion']
contour1Display.LookupTable = normalizedQCriterionLUT
contour1Display.SelectTCoordArray = 'None'
contour1Display.SelectNormalArray = 'Normals'
```

```
contour1Display.SelectTangentArray = 'None'
contour1Display.OSPRayScaleArray = 'Normalized Q Criterion'
contour1Display.OSPRayScaleFunction = 'PiecewiseFunction'
contour1Display.SelectOrientationVectors = 'None'
contour1Display.ScaleFactor = 0.03456759452819824
contour1Display.SelectScaleArray = 'Normalized Q Criterion'
contour1Display.GlyphType = 'Arrow'
contour1Display.GlyphTableIndexArray = 'Normalized Q Criterion'
contour1Display.GaussianRadius = 0.0017283797264099122
contour1Display.SetScaleArray = ['POINTS', 'Normalized Q Criterion']
contour1Display.ScaleTransferFunction = 'PiecewiseFunction'
contour1Display.OpacityArray = ['POINTS', 'Normalized Q Criterion']
contour1Display.OpacityTransferFunction = 'PiecewiseFunction'
contour1Display.DataAxesGrid = 'GridAxesRepresentation'
contour1Display.PolarAxes = 'PolarAxesRepresentation'

# init the 'PiecewiseFunction' selected for 'ScaleTransferFunction'
contour1Display.ScaleTransferFunction.Points = [-0.46899428963661194, 0
                                    .0, 0.5, 0.0, -0.46893325448036194,
                                     1.0, 0.5, 0.0]

# init the 'PiecewiseFunction' selected for 'OpacityTransferFunction'
contour1Display.OpacityTransferFunction.Points = [-0.46899428963661194,
                                     0.0, 0.5, 0.0, -0.
                                    46893325448036194, 1.0, 0.5, 0.0]

# show color bar/color legend
contour1Display.SetScalarBarVisibility(renderView1, True)

# update the view to ensure updated data information
renderView1.Update()

# get opacity transfer function/opacity map for 'NormalizedQCriterion'
normalizedQCriterionPWF = GetOpacityTransferFunction('
                                    NormalizedQCriterion')
normalizedQCriterionPWF.Points = [-0.46899428963661194, 0.0, 0.5, 0.0,
                                    -0.46893325448036194, 1.0, 0.5, 0.0
                                    ]
normalizedQCriterionPWF.ScalarRangeInitialized = 1

# Properties modified on contour1
contour1.ContourBy = ['POINTS', 'Q Criterion']

# update the view to ensure updated data information
renderView1.Update()
```

```
# Rescale transfer function
normalizedQCriterionLUT.RescaleTransferFunction(-0.46899428963661194, -
                                    8.949182285011883e-08)

# Rescale transfer function
normalizedQCriterionPWF.RescaleTransferFunction(-0.46899428963661194, -
                                    8.949182285011883e-08)

# Properties modified on contour1
contour1.Isosurfaces = [isoValue]

# update the view to ensure updated data information
renderView1.Update()

# Rescale transfer function
normalizedQCriterionLUT.RescaleTransferFunction(-0.46899428963661194, 0
                                    .0004771007224917412)

# Rescale transfer function
normalizedQCriterionPWF.RescaleTransferFunction(-0.46899428963661194, 0
                                    .0004771007224917412)

# set scalar coloring
ColorBy(contour1Display, ('POINTS', 'T'))

# Hide the scalar bar for this color map if no visible data is colored
                                    by it.
HideScalarBarIfNotNeeded(normalizedQCriterionLUT, renderView1)

# rescale color and/or opacity maps used to include current data range
contour1Display.RescaleTransferFunctionToDataRange(True, False)

# show color bar/color legend
contour1Display.SetScalarBarVisibility(renderView1, True)

# get color transfer function/color map for 'T'
tLUT = GetColorTransferFunction('T')
tLUT.RGBPoints = [1.0065757036209106, 0.231373, 0.298039, 0.752941, 1.
                                    4728603959083557, 0.865003, 0.
                                    865003, 0.865003, 1.
                                    9391450881958008, 0.705882, 0.
                                    0156863, 0.14902]
tLUT.ScalarRangeInitialized = 1.0

# get opacity transfer function/opacity map for 'T'
tPWF = GetOpacityTransferFunction('T')
```

```
tPWF.Points = [1.0065757036209106, 0.0, 0.5, 0.0, 1.9391450881958008, 1
                                    .0, 0.5, 0.0]
tPWF.ScalarRangeInitialized = 1


################################
### Loop for Exporting GLTFs ###
################################
for fileName in vts_FileNames :
    gltf_fileName = fileName.replace(f'{vts_directory}', f'{
                                        gltf_directory}')
    gltf_fileName = gltf_fileName.replace('.vts', '.gltf')

    # export view
    ExportView(f'{gltf_fileName}', view=renderView1, InlineData=1,
                                        SaveNormal=1, SaveBatchId=1)

    animationScene1.GoToNext()
```

Appendix E
Unity automated scripts

E.1 usdzToPrefab.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO;
using UnityEditor;
using USD.NET;

namespace Unity.Formats.USD.Examples {
    //[ExecuteInEditMode]
    public class usdzToPrefab : MonoBehaviour {

        public bool Activated = false;
        public int InitialFrame = 600;
        public string ImportFolderName = "Assets/Resources/ToImport/";
        public string ExportFolderName = "Assets/Resources/Prefab/";
        public string GeneralFileName  = "PUVWT_1-565.26";
        public int NumberOfFrames = 100;

        /// Import Options setting
        public BasisTransformation m_changeHandedness =
↪   BasisTransformation.SlowAndSafe;
        public PayloadPolicy LoadAllPayload = PayloadPolicy.LoadAll;
        public bool ImportCameras = false;
        public bool ImportMeshes = true;
```

```
        public bool ForceRebuild = false;
        public bool ImportSkinWeight = true;
        public bool UseDisplayColorAsFallbackMaterial = false;
        [Tooltip("Enable GPU instancing on materials for USD point or
↪  scene instances.")]
        public bool m_enableGpuInstancing = false;

        /// Private variables
        private string FileName;        // Short name of specific USD
↪  iterating file without extension
        private string iteration;       // Frame iteration variable
        private string usdFilePath;     // Complete name of specific USD
↪  iterating file
        private string prefabPath;      // Complete name of specific
↪  iterating prefab
        private Scene m_scene;
        private PrimMap m_primMap;      /// Keep track of all objects
↪  loaded.
        private GameObject USDmesh;
        private MaterialImportMode materialImportMode =
↪  MaterialImportMode.ImportPreviewSurface; //This option enables
↪  correct material application

        void Start() {
            InitUsd.Initialize();

            if (Activated)
            { USDZconverter(); }
        }

        public void USDZconverter()
        {
            /// When converting right handed (USD) to left handed
↪  (Unity), there are two options:
            /// Convert all transforms and points to left-handed (deep
↪  change of basis).
            /// is more computationally expensive, but results in fewer
↪  down stream surprises.
```

```
            var importOptions = new SceneImportOptions();
            importOptions.changeHandedness = m_changeHandedness;
            importOptions.payloadPolicy = LoadAllPayload;
            importOptions.importCameras = ImportCameras;
            importOptions.importMeshes = ImportMeshes;
            importOptions.forceRebuild = ForceRebuild;
            importOptions.importSkinWeights = ImportSkinWeight;
            importOptions.materialImportMode = materialImportMode;
            importOptions.useDisplayColorAsFallbackMaterial =
↪   UseDisplayColorAsFallbackMaterial;
            importOptions.enableGpuInstancing = m_enableGpuInstancing;


            for (int i = 0; i < NumberOfFrames; i++)
            {
                /// Formulating files' paths and names
                iteration = (InitialFrame + i).ToString("000");
↪   // Three digits number format
                FileName = string.Concat(GeneralFileName, iteration,
↪   "0");                 //
                usdFilePath = string.Concat(ImportFolderName,FileName,
↪   ".usdz");
                Debug.Log(usdFilePath);


                /// Building a new gameObject with MeshFilter and
↪   MeshRenderer components
                var newPrefab = new GameObject("Unity Prefab");
                newPrefab.transform.SetParent(this.transform,
↪   worldPositionStays: false);
                newPrefab.AddComponent<MeshFilter>();
                newPrefab.AddComponent<MeshRenderer>();

                /// Formulating new path and filename to store the
↪   gameobject as prefab
                prefabPath = string.Concat(ExportFolderName, FileName,
↪   ".prefab");
```

```
                /// Store gameObject "Unity Prefab" as prefab asset and
↪  track it as a different gameObject
                GameObject rootPrefab =
↪  PrefabUtility.SaveAsPrefabAsset(newPrefab, prefabPath);

                /// Loading USD as scene
                m_scene = null;
                m_scene = Scene.Open(usdFilePath);

                /// Building an empty gameObject to further add the USD
↪  scene transformed into a Unity scene
                var rootXf = new GameObject("USD Scene");
                rootXf.transform.SetParent(this.transform,
↪  worldPositionStays: false);

                /// Transforming USD scene into a Unity scene
                m_primMap = SceneImporter.BuildScene(m_scene,
                rootXf,
                importOptions,
                new PrimMap(),
                composingSubtree: false);

                /// Finding gameObject child named "mesh1" (default
↪  gameObject where the USD mesh is stored)
                USDmesh = GameObject.Find("mesh1");

                /// Finding the "MeshFilter" component where the mesh's
↪  data is contained
                MeshFilter ModelFilterComponent =
↪  USDmesh.GetComponent("MeshFilter") as MeshFilter;

                // Adding Mesh to asset
                AssetDatabase.AddObjectToAsset(ModelFilterComponent.mesh,
↪  rootPrefab);
                AssetDatabase.SaveAssets();

                /// Add destroying methods
                m_primMap.DestroyAll();
```

```
                m_scene.Close();
                Destroy(rootXf);
                Destroy(newPrefab);

            }

        }
    }
}
```

E.2 prefabMeshAnimator.cs

```
using System;
using System.Collections;
using UnityEngine;
using UnityEditor;

public class prefabMeshAnimator : MonoBehaviour {

    public bool Animated = false;
    public int InitialFrame = 0;
    public string PrefabFolderName = "Prefab/QCriterionTprime/ADIAB/";
    public string GeneralFileName  = "PUVWT_1-565.26";
    public int NumberOfFrames = 10;
    [Range(0.03f, 1.5f)]
    public float FPS = 0.5f;
    public Material FlowMaterial;


    /// Private variables
    private string FileName;        // Short name of specific USD
↪   iterating file without extension
    private string iteration;       // Frame iteration variable
    private string prefabPath;      // Complete name of specific
↪   iterating prefab
```

```
    private string prefabPathExtension;     // Complete name of specific
↪   USD iterating file

    void Start() {
        if (Animated)
        { StartCoroutine(PrefabAnimator()); }
    }

    public IEnumerator PrefabAnimator()
    {
        for (int i = 0; i < NumberOfFrames; i++)
        {
            /// Formulating prefab' paths and names relative from
↪   "Assets/Resources"
            iteration = (InitialFrame + i).ToString("000");
↪   // three digits number format
            FileName = string.Concat(GeneralFileName, iteration, "0");
            prefabPath = string.Concat(PrefabFolderName,FileName);
            Debug.Log(prefabPath);

            /// Formulating prefab' absolute paths and names with
↪   extension to further get the mesh database
            prefabPathExtension =
↪   string.Concat("Assets/Resources/",prefabPath,".prefab");
            Debug.Log(prefabPathExtension);

            /// Instantiate a gameObject by loading a prefab located
↪   relative from "Assets/Resources"
            GameObject instance = Instantiate(Resources.Load(prefabPath,
↪   typeof(GameObject))) as GameObject;
            Mesh newMesh =
↪   (Mesh)AssetDatabase.LoadAssetAtPath(prefabPathExtension,
↪   typeof(Mesh));
            instance.GetComponent<MeshFilter>().mesh = newMesh;
            instance.GetComponent<MeshRenderer>().material =
↪   FlowMaterial;

            instance.transform.rotation = transform.rotation;
```

```
            instance.transform.position = transform.position;
            instance.transform.localScale = transform.localScale;

            yield return new WaitForSeconds(FPS);

            /// Destroying gameObject
            Destroy(instance);

            if (i == NumberOfFrames-1)
            { i = -1; }
        }

    }
}
```

Bibliography

Albert, R., A. Patney, D. Luebke, and J. Kim (2017). Latency requirements for foveated rendering in virtual reality. *ACM Transactions on Applied Perception (TAP) 14*(4), 1–13. [Cited on page(s): 24]

Anderson Jr, J. D. and J. D. Anderson (1998). *A history of aerodynamics: and its impact on flying machines*. Cambridge university press. [Cited on page(s): 1, 2, 6]

Araya, G., C. Castillo, and F. Hussain (2015). The log behaviour of the Reynolds shear stress in accelerating turbulent boundary layers. *Journal of Fluid Mechanics 775*, 189–200. [Cited on page(s): 54]

Araya, G. and C. Lagares (2022). Implicit subgrid-scale modeling of a mach-2.5 spatially-developing turbulent boundary layer. *Entropy 24*(4), 555. DOI:https://doi.org/10.3390/e24040555. [Cited on page(s): 15]

Baskaran, V., A. Smits, and P. Joubert (1987). A turbulent flow over a curved hill part 1. growth of an internal boundary layer. *Journal of Fluid Mechanics 182*, 47–83. [Cited on page(s): x, xi, 16, 28, 30, 31, 38, 43, 46, 48, 49, 51, 52, 53, 54, 57, 64, 111, 150]

Bradshaw, P. (1973). Effects of streamline curvature on turbulent flow. *AGARDograph 169*. [Cited on page(s): 12]

Brookes, J., M. Warburton, M. Alghadier, M. Mon-Williams, and F. Mushtaq (2020). Studying human behavior with virtual reality: The unity experiment framework. *Behavior research methods 52*(2), 455–463. [Cited on page(s): 24, 25]

Brown, G. L. and A. Roshko (1974). On density effects and large structure in turbulent mixing layers. *Journal of Fluid Mechanics 64*(4), 775–816. [Cited on page(s): 23]

Bryson, S. (1993). The virtual windtunnel: visualizing modern cfd datasets with a virtual environment. In *NASA. Johnson Space Center, Proceedings of the 1993 Conference on Intelligent Computer-Aided Training and Virtual Environment Technology*. [Cited on page(s): 25]

Byron, S. and C. Levit (1991). The virtual windtunnel: An environment for the exploration of three-dimensional unsteady flows. In *Proceedings of Visualization*, Volume 91, pp. 17–24. [Cited on page(s): 25]

Celik, I. B., U. Ghia, P. J. Roache, and C. J. Freitas (2008). Procedure for estimation and reporting of uncertainty due to discretization in cfd applications. *Journal of fluids Engineering-Transactions of the ASME 130*(7). [Cited on page(s): 151]

Cengel, Y. A. and J. M. Cimbala (2014). *Fluid Mechanics: Fundamentals and Applications*. McGraw-Hill Education. [Cited on page(s): 6, 7, 8, 9, 44]

Chaouat, B. (2017). The state of the art of hybrid rans/les modeling for the simulation of turbulent flows. *Flow Turbulence Combust 99*, 279–327. [Cited on page(s): 22]

Craig, A. B., W. R. Sherman, and J. D. Will (2009). *Developing virtual reality applications: Foundations of effective design*. Morgan Kaufmann. [Cited on page(s): 24]

Deming, D. (2020). The aqueducts and water supply of ancient rome. *Ground water 58*(1), 152. [Cited on page(s): 1]

Elbamby, M. S., C. Perfecto, M. Bennis, and K. Doppler (2018). Toward low-latency and ultra-reliable virtual reality. *IEEE Network 32*(2), 78–84. [Cited on page(s): 24]

Flavián, C., S. Ibáñez-Sánchez, and C. Orús (2019). The impact of virtual, augmented and mixed reality technologies on the customer experience. *Journal of business research 100*, 547–560. [Cited on page(s): x, 23]

Friendly, M. and D. J. Denis (2001). Milestones in the history of thematic cartography, statistical graphics, and data visualization. *URL http://www. datavis. ca/milestones 32*, 13. [Cited on page(s): 22]

G. Araya and L. Castillo (2013). DNS of turbulent thermal boundary layers subjected to adverse pressure gradients. *Physics of Fluids*, 095107. [Cited on page(s): 50, 56, 57]

Hansen, C. D. and C. R. Johnson (2011). *Visualization handbook*. Elsevier. [Cited on page(s): 23]

Heath, T. L. et al. (2002). *The works of Archimedes*. Courier Corporation. [Cited on page(s): 1]

Hill, H. D. R. (2019). *Islamic science and engineering*. Edinburgh University Press. [Cited on page(s): 1]

HPCV (2018). HPCVLab. https://www.uprm.edu/hpcvl/. [Cited on page(s): 16, 108]

Hunt, J. C., A. A. Wray, and P. Moin (1988). Eddies, streams, and convergence zones in turbulent flows. *Studying turbulence using numerical simulation databases, 2. Proceedings of the 1988 summer program*. [Cited on page(s): 73]

Kays, W. M. and M. E. Crawford (1993). *Convective Heat and Mass Transfer*, Volume 3rd ed. McGraw-Hill, New York. [Cited on page(s): 44, 50]

Knight, D. and P. Saffman (1978). Turbulence model predictions for flows with significant mean streamline curvature. *AIAA 78-258*. [Cited on page(s): 20]

Lagares, C., J. Santiago, and G. Araya (2021). Turbulence modeling in hypersonic turbulent boundary layers subject to convex wall curvature. *AIAA Journal 59*(12), 1–20. [Cited on page(s): 32]

Lagares, C. J. and G. Araya (2021). *Compressibility Effects on High-Reynolds Coherent Structures via Two-Point Correlations*. [Cited on page(s): 14, 45, 54, 56, 58]

Lagares, C. J., K. E. Jansen, J. Patterson, and G. Araya (2019). The effect of concave surface curvature on supersonic turbulent boundary layers. Presented at the 72^{nd} Annual Meeting of the American Physical Society's Division of Fluid Dynamics. [Cited on page(s): 60]

Lagares, C. J., D. Paeres, and G. Araya (2021). Wall temperature effect on thermal coherent structures over supersonic turbulent boundary layers subject to surface curvature. 74th Annual Meeting of the APS Division of Fluid Dynamics. DOI:https://doi.org/10.1103/APS.DFD.2021.GFM.V0027. [Cited on page(s): 72]

Lagares, C. J., W. Rivera, and G. Araya (2021, 1). Aquila: A distributed and portable post-processing library for large-scale computational fluid dynamics. *AIAA SciTech*. [Cited on page(s): 14]

Landels, J. G. (1979). Water-clocks and time measurement in classical antiquity. *Endeavour 3*(1), 32–37. [Cited on page(s): 1]

Launder, B. E. and D. B. Spalding (1983). The numerical computation of turbulent flows. In *Numerical prediction of flow, heat transfer, turbulence and combustion*, pp. 96–116. Elsevier. [Cited on page(s): 18]

Lee, C.-H. (2018). Rough boundary treatment method for the shear-stress transport k-ω model. *Engineering Applications of Computational Fluid Mechanics 12*(1), 261–269. [Cited on page(s): 18, 19]

Lewiner, T., H. Lopes, A. W. Vieira, and G. Tavares (2003). Efficient implementation of marching cubes' cases with topological guarantees. *Journal of graphics tools 8*(2), 1–15. [Cited on page(s): 67]

Li, Q., P. Schlatter, L. Brandt, and D. S. Henningson (2009). Dns of a spatially developing turbulent boundary layer with passive scalar transport. *International Journal of Heat and Fluid Flow 30*(5), 916–929. [Cited on page(s): 32, 45, 62]

McCormick, B. H. (1987). Visualization in scientific computing. *Computer graphics 21*(6). [Cited on page(s): 23]

Menter, F. R. (1994). Two-equation eddy-viscosity turbulence models for engineering applications. *AIAA Journal 32*(8), 1598–1605. [Cited on page(s): 19, 30, 32, 51, 150]

Menter, F. R., M. Kuntz, and R. Langtry (2003). Ten years of industrial experience with the sst turbulence model. *Turbulence, heat and mass transfer 4*(1), 625–632. [Cited on page(s): 19]

Milgram, P. and F. Kishino (1994). A taxonomy of mixed reality visual displays. *IEICE TRANSACTIONS on Information and Systems 77*(12), 1321–1329. [Cited on page(s): 23]

Milidonis, K., B. Semlitsch, and T. Hynes (2018). Effect of clocking on compressor noise generation. *AIAA Journal 56*(11), 4225–4231. [Cited on page(s): 20]

Mollicone, J.-P., F. Battista, P. Gualtieri, and C. M. Casciola (2017). Effect of geometry and reynolds number on the turbulent separated flow behind a bulge in a channel. *Journal of Fluid Mechanics 823*, 100–133. [Cited on page(s): 11]

Moser, R. and P. Moin (1984). Direct numerical simulation of curved turbulent channel flow. *NASA-TM-85974*. [Cited on page(s): 12]

Moser, R. and P. Moin (1987). The effects of curvature in wall-bounded turbulent flows. *Journal of Fluid Mechanics 175*, 479–510. [Cited on page(s): 12]

Moukalled, F., L. Mangani, M. Darwish, et al. (2016). *The finite volume method in computational fluid dynamics*, Volume 113. Springer. [Cited on page(s): 13, 14, 15, 22, 37, 118]

Narasimha, R. & Sreenivasan, K. (1979). Relaminarization of fluid flows. *Advances in Applied Mechanics 19*, 221–309. [Cited on page(s): 57]

Narasimha, R. (1983). Relaminarization-magnetohydrodynamic and otherwise. *AIAA Progress in Astronautics and Aeronautics 84*, 30–53. [Cited on page(s): 49]

Nielson, G., H. Hagen, and H. Müller (1997). Scientific visualization. Institute of Electrical & Electronics Engineers. [Cited on page(s): 23]

Paciorri, R., W. Dieudonné, G. Degrez, J.-M. Charbonnier, and H. Deconinck (1998). Exploring the validity of the spalart-allmaras turbulence model for hypersonic flows. *Journal of Spacecraft and Rockets 35*(2). [Cited on page(s): 32]

Paeres, D., C. Lagares, and G. Araya (2022a). Assessment of incompressible turbulent flow over a curved hill with passive scalar transport. *AIAA SciTech, AIAA 2022-0049*. [Cited on page(s): 29, 66]

Paeres, D., C. Lagares, and G. Araya (2022b). Assessment of turbulence models over a curved hill flow with passive scalar transport. *Energies 15*(16), 6013. DOI:https://doi.org/10.3390/en15166013. [Cited on page(s): xi, 14, 15, 22, 29, 43, 66]

Paeres, D., C. J. Lagares, and G. Araya (2021). The use of Augmented Reality (AR) in flow visualization. 74th Annual Meeting of the APS Division of Fluid Dynamics. DOI:https://doi.org/10.1103/APS.DFD.2021.GFM.V0028. [Cited on page(s): 29, 66, 85, 109]

Paeres, D., C. J. Lagares, J. Santiago, A. B. Craig, K. Jansen, and G. Araya (2020). Turbulent Coherent Structures via VR/AR. 73th Annual Meeting of the APS Division of Fluid Dynamics. DOI:https://doi.org/10.1103/APS.DFD.2020.GFM.V0045. [Cited on page(s): 29, 66, 85, 97, 109]

Paeres, D., J. Santiago, C. J. Lagares, W. Rivera, A. B. Craig, and G. Araya (2021). Design of a Virtual Wind Tunnel for CFD Visualization. In *AIAA Scitech 2021 Forum*, pp. 1600. [Cited on page(s): 23, 24, 25, 29, 66, 97]

Patrick, W. P. (1987). Flowfield measurements in a separated and reattached flat plate turbulent boundary layer. *NASA CR4052*. [Cited on page(s): 11]

Pirozzoli, S., M. Bernardini, and P. Orlandi (2016). Passive scalars in turbulent channel flow at high reynolds number. *Journal of Fluid Mechanics 788*, 614–639. [Cited on page(s): 21]

Pope, S. B. (2000). *Turbulent Flows*. Cambridge University Press. [Cited on page(s): 58]

Purohit, S., I. F. S. A. Kabir, and E. Y. K. Ng (2021). On the accuracy of urans and les-based cfd modeling approaches for rotor and wake aerodynamics of the (new) mexico wind turbine rotor phase-iii. *Energies 14*(16). [Cited on page(s): 22]

Quinones, C. (2020). Transport phenomena in crossflow jets subject to very strong favorable pressure gradient. *MSc thesis, University of Puerto Rico-Mayaguez*. [Cited on page(s): 47, 57]

Radhakrishnan, S., U. Piomelli, A. Keating, and A. S. Lopes (2006). Reynolds-averaged and large-eddy simulations of turbulent non-equilibrium flows. *Journal of Turbulence 7*, N63. [Cited on page(s): 22]

Radianti, J., T. A. Majchrzak, J. Fromm, and I. Wohlgenannt (2020). A systematic review of immersive virtual reality applications for higher education: Design elements, lessons learned, and research agenda. *Computers & Education 147*, 103778. [Cited on page(s): 24]

Raheem, M. A., P. Edi, A. A. Pasha, M. M. Rahman, and K. A. Juhany (2019). Numerical study of variable camber continuous trailing edge flap at off-design conditions. *Energies 12*(16). [Cited on page(s): 32]

Rosenblum, L. (2000). Virtual and augmented reality 2020. *IEEE Computer Graphics and Applications 20*(1), 38–39. [Cited on page(s): 24]

Rumsey, C. (November 2021). The menter shear stress transport turbulence model. *Turbulence Modeling Resource , Langley Research Center / National Aeronautics and Space Administration*. [Cited on page(s): 19]

Saidin, N. F., N. Halim, and N. Yahaya (2015). A review of research on augmented reality in education: Advantages and applications. *International education studies 8*(13), 1–8. [Cited on page(s): 24]

Schlatter, P. and Orlu, R. (2010). Assessment of direct numerical simulation data of turbulent boundary layers. *Journal of Fluid Mechanics 659*, 116–126. [Cited on page(s): 14, 45, 54, 56, 58]

Schlichting, H. and K. Gersten (2017). *Boundary-Layer Theory* (9th ed.). Springer. [Cited on page(s): 2, 8, 9, 11, 14]

Shapiro, A., G. Grossman, and D. Greenblatt (2021). Simplified transition and turbulence modeling for oscillatory pipe flows. *Energies 14*(5). [Cited on page(s): 32]

Simpson, R. (1985). Two-dimensional turbulent separated flow. *AGARDograph 287*, Vol I. [Cited on page(s): 11]

Simpson, R. (1989). Turbulent boundary layer separation. *Ann. Rev. Fluid Mechanics 21*, 205–234. [Cited on page(s): x, 11, 12]

Simpson, R. L., M. Ghodbane, and B. E. McGrath (1987). Surface pressure fluctuations in a separating turbulent boundary layer. *Journal of Fluid Mechanics 177*, 167–186. [Cited on page(s): 11]

Skote, M., D. Henningson, and R. Henkes (1998). Direct numerical simulation of self-similar turbulent boundary layers in adverse pressure gradients. *Flow, Turbul. Combust. 60*, 47–85. [Cited on page(s): 57, 58]

Smirnov, P. E. and F. R. Menter (2009, 07). Sensitization of the SST Turbulence Model to Rotation and Curvature by Applying the Spalart–Shur Correction Term. *Journal of Turbomachinery 131*(4). [Cited on page(s): 20]

Spalart, P. and S. Allmaras (1992). A one-equation turbulence model for aerodynamic flows. *30th Aerospace Science Meeting and Exhibit, Reno, NV, USA AIAA Paper 92-0439*. [Cited on page(s): 20, 30, 32]

Spalart, P. and M. Shur (1997). On the sensitization of turbulence models to rotation and curvature. *Aerospace Science and Technology 1*(5), 297–302. [Cited on page(s): 20]

Spalart, P. R. and C. L. Rumsey (2007). Effective inflow conditions for turbulence models in aerodynamic calculations. *AIAA Journal 45*(10), 2544–2553. [Cited on page(s): 32]

Versteeg, H. K. and W. Malalasekera (2007). *An introduction to computational fluid dynamics: the finite volume method*. Pearson education. [Cited on page(s): 13, 16, 17, 20, 37]

Wahba, E. (2016). The contribution of alexandria to mathematics and engineering. DOI: 10.13140/RG.2.2.27208.11528. [Cited on page(s): 1]

Warhaft, Z. (2000). Passive scalars in turbulent flows. *Ann. Rev. Fluid Mechanics 32*, 203–240. [Cited on page(s): 12]

White, F. (2011). *Viscous Fluid Flow 3e*. McGraw-Hill Education (India) Pvt Limited. [Cited on page(s): x, 4, 5, 6, 7, 10, 13]

Wilcox, D. C. (1988). Reassessment of the scale-determining equation for advanced turbulence models. *AIAA journal 26*(11), 1299–1310. [Cited on page(s): 19]

Wilcox, D. C. (2006). *Turbulence Modeling for CFD*. Third Edition, La Cañada, CA: DCW Industries. [Cited on page(s): 32]

William R. Sherman, Alan B. Craig, M. P. B. and C. Bushell (1997). Scientific visualization. *Allen B. Jr. Tucker (Ed.), Chapter 35 of The Computer Science and Engineering Handbook. CRC Press, Boca Raton, FL.*. [Cited on page(s): 23]

Williams, J. (1977). Incompressible boundary layer separation. *Ann. Rev. Fluid Mechanics 9*, 113–144. [Cited on page(s): 11]

Winant, C. D. and F. K. Browand (1974). Vortex pairing: the mechanism of turbulent mixing-layer growth at moderate reynolds number. *Journal of Fluid Mechanics 63*(2), 237–255. [Cited on page(s): 23]

Zhang, J., Z. Wang, M. Sun, H. Wang, C. Liu, and J. Yu (2020). Effect of the backward facing step on a transverse jet in supersonic crossflow. *Energies 13*(16). [Cited on page(s): 22]

Zhao, J., Q. Lu, and D. Yang (2022). Experimental and numerical analysis of rotor-rotor interaction characteristics inside a multistage transonic axial compressor. *Energies 15*(7). [Cited on page(s): 20]