ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
& ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

GPU-enabled rigged model animation in Elements framework

Vlachos Elias

Supervisor: Dr. Antonis Protopsaltis

KOZANI/OCTOBER/2024

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
& ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αλγόριθμοι σχεδιοκίνησης 3Δ μοντέλων με σκελετό, στη GPU, με το εκπαιδευτικό Framework Elements

Βλάχος Ηλίας

Επιβλέπων: Δρ. Αντώνης Πρωτοψάλτης

ΚΟΖΑΝΗ/ΟΚΤΩΒΡΙΟΣ/2024

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
& ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΗΛΩΣΗ ΜΗ ΛΟΓΟΚΛΟΠΗΣ ΚΑΙ ΑΝΑΛΗΨΗΣ ΠΡΟΣΩΠΙΚΗΣ ΕΥΘΥΝΗΣ

Δηλώνω ρητά ότι, σύμφωνα με το άρθρο 8 του Ν. 1599/1986 και τα άρθρα 2,4,6 παρ. 3 του Ν. 1256/1982, η παρούσα Διπλωματική Εργασία με τίτλο: ”GPU-enabled rigged model animation in Elements framework” καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας και αναφέρονται ρητώς μέσα στο κείμενο που συνοδεύουν, και η οποία έχει εκπονηθεί στο Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Πανεπιστημίου Δυτικής Μακεδονίας, υπό την επίβλεψη του μέλους του Τμήματος κ.: ”Δρ. Αντώνιος Πρωτοψάλτης” αποτελεί αποκλειστικά προϊόν προσωπικής εργασίας και δεν προσβάλλει κάθε μορφής πνευματικά δικαιώματα τρίτων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων, αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα. Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και μόνο.

Υπογραφή Φοιτητή:.....

Περίληψη

Η παρούσα μελέτη εισάγει μια πρωτοποριακή μέθοδο για την υλοποίηση τρισδιάστατης κίνησης σε μια σκηνή μέσω της ενσωμάτωσης φωτισμού και υφών. Η καινοτόμος προσέγγιση βασίζεται στο προηγμένο αρχιτεκτονικό πλαίσιο Entity-Component-System (ECS) και στον σχεδιασμό γράφου σκηνής. Αυτές οι τεχνικές αιχμής υλοποιούνται με τη βοήθεια του επαναστατικού εκπαιδευτικού εργαλείου Elements, το οποίο συμπληρώνεται από τις βιβλιοθήκες pyECSS και pyGLV, που επίσης παρέχονται από το Elements. Η μελέτη αναλύει με ακρίβεια κάθε προγραμματιστική και μαθηματική πτυχή της διαδικασίας υλοποίησης. Περιλαμβάνεται επεξήγηση για το πώς να ενσωματωθούν ομαλά τα μοντέλα astroboy και ρομποτικού βραχίονα, να διαχειριστούν αποτελεσματικά οι παράμετροι φωτισμού, να οργανωθούν οι ακολουθίες κίνησης και, τέλος, να εναρμονιστούν αυτά τα πολυδιάστατα στοιχεία ώστε να παραχθεί το επιθυμητό οπτικό αποτέλεσμα. Ο κώδικας, γραμμένος σε Python και GLSL, αξιοποιεί τις δυνατότητες των βιβλιοθηκών Elements, καθώς και της βιβλιοθήκης pyassimp για αποδοτική εισαγωγή μοντέλων. Η συνέργεια αυτών των στοιχείων όχι μόνο αποδεικνύει την ευελιξία της προτεινόμενης μεθοδολογίας, αλλά και υπογραμμίζει τις δυνατότητές της για την προώθηση του τομέα των τρισδιάστατων γραφικών και κινούμενων σχεδίων.

Abstract

This study introduces a pioneering method for realizing three-dimensional animation within a scene through the integration of lighting and textures. The innovative approach is anchored in the avant-garde architectural framework of Entity-Component-System (ECS) and scene graph design. These cutting-edge techniques are implemented with the aid of the revolutionary educational toolkit, Elements, complemented by the pyECSS and pyGLV libraries also provided by Elements. The study meticulously dissects each programming and mathematical facet of the implementation process. This includes elucidation on how to seamlessly incorporate the astroboy and robot arm model, effectively manage lighting parameters, orchestrate animation sequences, and finally, harmonize these multifaceted elements to produce the desired visual outcome. The codebase, crafted in Python and GLSL, leverages the capabilities of the Elements libraries mentioned earlier, as well as the pyassimp library for efficient model importation. The synergy of these components not only demonstrates the versatility of the proposed methodology but also underscores its potential for advancing the field of three-dimensional graphics and animation.

Keywords: Three-dimensional animation, Phong lighting, Textures, Entity-Component-System (ECS), Scene graph design, Elements educational toolkit, pyECSS library, pyGLV library, pyassimp library, Model importation, Python, GLSL, Animation sequences, Lighting parameters, Visual outcome, Graphics and animation, Implementation process, Programming, Mathematical aspects, Three-dimensional graphics

Acknowledgments

I would like to express my deepest gratitude to my thesis advisor, Dr. Antonis Protopsaltis, for his guidance, support, and encouragement throughout this research. I am also deeply grateful to Dr. Manos Kamarianakis for his invaluable help, feedback, and suggestions, which greatly enhanced the quality of this work. Additionally, I would like to extend my sincere thanks to Dr. George Papagiannakis for providing me with essential materials that contributed to the success of this project.

I extend my heartfelt thanks to my parents and sister for their unwavering support, love, and encouragement. Your belief in me has been a constant source of motivation.

Lastly, I would like to thank my close friends and loved ones who have supported me throughout this journey. Your moral support and encouragement have been indispensable.

Contents

Figure Catalog

Code Catalog

1 Introduction

In recent years, animation has evolved rapidly, offering immersive and realistic experiences across various fields, such as entertainment, education, and simulation. Traditionally, animation systems were built using an approach where each object in a scene was treated as an autonomous entity with tightly coupled logic and data. While effective for simpler systems, these approaches struggle to meet the growing demand for dynamic and interactive applications.

To address these limitations, the Entity Component System (ECS) model has emerged as an efficient alternative. ECS separates an object's identity (Entity), its characteristics and behavior (Component), and its processing logic (System). This decoupling not only enables a more modular and flexible design but also enhances scalability and performance, making it ideal for handling real-time animations.

In this context, Elements plays a significant role as an educational framework for computer graphics. Elements combines the power of ECS with the flexibility of scene graph architectures, offering students the ability to rapidly develop complex 3D animations and simulations. Designed with education in mind, it provides an accessible platform for teaching fundamental concepts of computer graphics, while also offering advanced tools for creating professional-level scenes.

The Elements framework provides us with a way to explore the integration of GPU-enabled algorithms within the ECS framework to enhance the animation of textured 3D objects in a radiance-transferred scene. This research seeks to leverage the strengths of ECS that Elements provides to improve the performance and flexibility of animation systems, addressing the growing need for more dynamic and interactive 3D graphics in various applications. The primary purpose is to demonstrate how ECS can be effectively utilized to create more efficient and scalable animation systems. By incorporating GPU acceleration, the thesis aims to achieve to make a tool for Elements so that there is an easy way for students to make and understand animations.

The traditional approach to animation involves designing systems where each object in the scene is self-contained, encapsulating both its data and behavior. In this approach, each object typically has its own update and rendering methods, which manage its state and appearance. While this can work well for simple applications with a limited number of objects, it presents several challenges as the complexity of the scene increases.

As the number of objects in the scene grows, the system becomes more difficult to manage and scale. Each object must handle its own logic, leading to a proliferation of code that can be hard to maintain, extend and understand for students. Additionally, interactions between objects can become increasingly complex, as each object needs to be aware of the state and behavior of other objects.

Resource management is often less efficient in the traditional approach. Each object may independently load and manage its resources, such as textures and meshes, leading to potential duplication and wastage of memory. This can sig-

nificantly impact performance, especially in memory-constrained environments.

The traditional design tightly couples data and behavior, making it difficult to reuse components across different objects. Modifying the behavior of an object often requires changes to its core code, reducing flexibility and increasing the risk of introducing bugs.

The ECS model addresses these issues by decoupling the components of an animation system, allowing for better management of resources and more straightforward implementation of complex behaviors. However, the integration of GPU-enabled algorithms within ECS remains a relatively unexplored area, presenting an opportunity to further enhance the capabilities of ECS-based animation systems. Solving this problem is crucial for several reasons.

The GPU acceleration can significantly enhance the performance of animation systems, making real-time rendering of complex scenes possible. ECS-based systems are inherently more scalable, as they allow for better management of resources and parallel processing. The separation of concerns in ECS facilitates easier modification and extension of the system, which is essential for developing adaptable and reusable animation frameworks. By enhancing the Elements educational toolkit with new tools and ideas, this research contributes to providing the best education for students in Computer Graphics, fostering innovation and skill development in this field.

1.1 Overview of chapters

The second chapter provides a detailed analysis of various educational frameworks relevant to teaching computer graphics. It reviews their strengths and weaknesses, with a focus on how they facilitate learning shader-based programming and real-time rendering. This chapter lays the groundwork for understanding the educational value of the Elements framework, which is discussed later.

The third chapter introduces the Entity-Component-System (ECS) architecture and its integration within the Elements framework. It explains the structure and functionality of ECS and scene graphs, along with their use in managing 3D scenes and real-time animations.

The fourth chapter delves into the practical aspects of the project, covering topics such as skinning/rigging, model data extraction, and the overall design and implementation process.

The fifth chapter presents the results of the research, analyzing the performance improvements achieved through the integration of GPU-enabled algorithms in ECS, while including code examples and discusses the challenges encountered during development.

The final chapter summarizes the key contributions of the thesis, reflecting on the research objectives and outcomes. It also outlines potential directions for future research, suggesting ways to further enhance the capabilities of ECS-based animation systems.

By addressing these aspects, this thesis aims to contribute to the field of com-

puter graphics by demonstrating the benefits of integrating GPU-enabled algorithms within the ECS framework, ultimately paving the way for more efficient and scalable animation systems.

2 Educational frameworks

This chapter will provide an analysis of various educational frameworks relevant to teaching computer graphics, with a particular focus on shader-based programming and real-time rendering techniques. We will review several frameworks, evaluating their strengths and weaknesses, and compare how they facilitate learning and practical applications in computer graphics. By examining these frameworks, we aim to establish a foundation for understanding the educational role and significance of the Elements framework, which will be explored in detail later in this thesis.

2.1 Describing the educational frameworks

2.1.1 GL-Socket

The first framework we are going to discuss is the GL-Socket Andújar Gran et al. (2018), it has several key strengths and weaknesses. One of its primary strengths is its ability to foster independent learning by providing students with structured tasks they can work on autonomously. The plugin-based design promotes reusability and modular organization, allowing educators to create reusable, modular tasks that build upon each other. Moreover, the framework includes self-assessment tools, which allow students to test their work against reference solutions. This encourages active learning and immediate feedback. The framework is also multi-platform compatible, making it adaptable for different environments, including exams, where it automates the assessment process.

However, there are some challenges associated with the framework. The initial setup can be complex, particularly for students who are not familiar with OpenGL or GLSL, leading to a steep learning curve. Additionally, the framework does not fully support modern shader types and OpenGL features, which may limit its application in more cutting-edge topics within computer graphics education. Furthermore, instructors are required to develop custom tasks, which can be time-consuming and resource-intensive.

Overall, this framework offers a balance between flexibility in teaching and technical challenges, with its strengths in promoting independent learning, reusability, and feedback, though it requires careful management to address its setup complexity and shader limitations.

2.1.2 Shader-Based OpenGL

The Shader-Based OpenGL Miller (2014) framework aids in teaching shader-based OpenGL using a simplified version of the Model-View-Controller (MVC) pattern. The framework allows students to incrementally grasp key concepts such as GPU-CPU interaction and object-oriented programming in a structured way. It is particularly helpful in guiding students through the complexities of shader programming, offering practical, hands-on experience that prepares them for large-scale applications, both in terms of code management and data handling.

Despite its strengths, the framework presents certain challenges, particularly for students new to graphics programming. Shader-based OpenGL has a steep learning curve, which was evident in the early implementations of the course, where students struggled with the complexity of the content. The paper acknowledges that while the framework provides clear instructional value, the initial version required significant restructuring to improve the clarity and pacing of the material. Even with these improvements, students are expected to dedicate considerable time to fully understand the underlying concepts.

Moreover, the framework exposes students to intermediate and advanced object-oriented programming concepts within a real-world context, which helps them understand design patterns such as MVC. The course also includes CPU-GPU coordination, which gives students valuable experience in GPU programming and prepares them to deal with real-time rendering and large-scale interactive applications.

However, working with established code bases, as required by the framework, can be a challenge for students who are accustomed to building projects from scratch, making it a double-edged sword for teaching advanced graphics techniques.

In summary, the framework enhances learning by providing structured, incremental exposure to shader-based OpenGL, but its complexity demands significant effort from students, especially those without prior experience, requiring careful instructional design to mitigate challenges.

2.1.3 glGA

The glGA Papagiannakis et al. (2014) framework was developed to simplify the teaching of complex graphics principles, offering students the tools to create interactive 3D applications across multiple platforms such as Windows, Linux, macOS, and iOS.

One of its major advantages is the balance it strikes between providing students with hands-on experience in real-time graphics programming and minimizing the overwhelming complexities of OpenGL, C++ and GLSL programming, which allows students to focus on core graphics concepts rather than unrelated software engineering tasks. The framework includes a range of examples and assignments that cover topics like window initialization, geometric transformations, Blinn-Phong lighting, texture mapping, and skinned character animation. This helps students build from basic examples to more sophisticated, shader-based applications.

A key feature of glGA is its cross-platform compatibility, with minimal modifications the students can develop their projects across various platforms, even including mobile environments like iOS. The examples and assignments which are included in the framework make it easier for students to grasp the foundational concepts of shader-based programming, while the inclusion of GUI-based scene manipulation through AntTweakBar further enhances their understanding by allowing real-time parameter adjustment.

However, we need to acknowledges some challenges. For instance, novice students, particularly those with limited exposure to third-party open-source libraries, initially faced difficulties in setting up the build environment. To address this, the framework developers included a support system with online forums, video tutorials, and real-time QA sessions during lectures. Additionally, while the glGA framework is designed to simplify shader programming, students still need to invest significant effort to fully grasp its advanced applications.

The results of using glGA in the Computer Science Department at the University of Crete over three semesters have been positive. Students, even those without prior experience in OpenGL, C++ and GLSL, were able to create moderately complex CG applications using shaders by the middle of the course. Despite some early difficulties, student feedback was generally positive, and the framework was instrumental in providing a more hands-on, project-based learning environment for graphics programming. Its simplicity, combined with the power of shader programming, allows students to learn complex GPU-based development while avoiding the steep learning curve typically associated with advanced graphics frameworks. The framework continues to evolve, with plans to expand its capabilities to include geometry and tessellation shaders, further broadening its scope in CG education.

2.1.4 CodeRunnerGL

The CodeRunnerGL Wünsche et al. (2019) is a system built on the CodeRunner platform designed to enhance the teaching of computer graphics, particularly OpenGL programming. The tool allows for automatic assessment of OpenGL code, providing a highly interactive web-based environment where students can visualize and manipulate 3D renderings.

One of the key strengths of CodeRunnerGL is its ability to offer interactive 3D output and real-time feedback on code submissions. This allows students to experiment with OpenGL programming in a more engaging way. The tool provides sandboxes for experimenting with different graphics concepts such as geometric primitives, transformations, and illumination models. By offering this hands-on experimentation space, students can modify parameters and immediately see the impact of their changes, which improves understanding of core concepts. Additionally, CodeRunnerGL incorporates multiple test cases and automated feedback. The system compares students' 3D outputs and underlying OpenGL states with reference solutions, providing both visual and textual feedback. This helps students understand complex transformations and rendering steps in a practical context. Furthermore, the tool supports automated grading by analyzing student solutions based on the correctness of transformations and rendered scenes, making it an effective tool for large classes.

There are some limitations that have to be noted. One challenge is that creating effective test cases and generating meaningful automated feedback can be difficult, particularly for more complex programming tasks. While the interactive 3D window improves understanding, users have suggested improvements, such as a larger OpenGL window, automated updates without needing to press a "check" button, and animating solutions to show intermediate results of complex trans-

formations. A collaborative mode for peer programming and assessment was also suggested. Additionally, the current system reverts to default views after each re-evaluation, which can confuse students when tracking changes made in their solutions.

CodeRunnerGL provides a powerful platform for teaching and assessing OpenGL in a highly interactive and automated way. Its features help students to actively engage with and better understand computer graphics concepts, although improvements in feedback mechanisms and user interaction are needed to fully maximize its potential.

2.1.5 ShaderLabFramework

ShaderLabFramework Toisoul et al. (2017) is an educational tool designed to simplify learning GLSL shader programming. This framework, used at Imperial College London, provides an accessible interface for students to work with OpenGL4 in a structured and interactive environment. It bridges the gap between highly complex graphics APIs and game engines, focusing on shader-based programming. The framework includes features such as a two-pass rendering pipeline, allowing students to learn essential concepts like illumination, transformations, texture mapping, and simple GPU-based ray tracing.

The main strength of ShaderLabFramework lies in its user-friendly interface and simplified programming environment, which makes it accessible even to students with limited prior experience in computer graphics or OpenGL programming. It incorporates a range of lab exercises aligned with the course syllabus, covering key topics like vertex and fragment shaders, geometry processing, and advanced techniques like bump mapping and Monte-Carlo path tracing. The structured design helps students progressively learn through hands-on exercises, building up their understanding of shader programming concepts while eliminating the need to grapple with complex setup procedures.

The framework's ability to save entire shader pipelines and configurations as XML files also simplifies grading and feedback for instructors. Students can experiment with various shader techniques and submit their work for evaluation, while the framework's integration into a standardized lab infrastructure ensures consistency in student experiences across different machines.

However, while ShaderLabFramework is highly effective for shader-based programming, it does not cover more complex graphics frameworks used in professional game development, and it is somewhat limited to the scope of teaching fundamental concepts. The framework also doesn't support broader game development elements like physics engines or extensive real-time rendering effects.

Overall, ShaderLabFramework provides a solid foundation for teaching modern shader programming, offering a streamlined and accessible learning environment for undergraduate computer graphics students. The open-source nature of the tool encourages further development and adaptation, making it a valuable resource for both students and educators.

2.1.6 Rayground

Rayground Vitsas et al. (2020) is a web-based educational platform aimed at teaching ray tracing in an accessible and interactive way. The platform allows students to experiment with ray tracing concepts without the complexity of setting up extensive codebases, focusing instead on the core principles of ray tracing. Rayground provides a WebGL-based development environment where students can write shader programs for different stages of ray tracing, such as ray generation, hit/miss handling, and post-processing.

One of the main strengths of Rayground is its user-friendly, shader-based approach, which allows students to explore the intricacies of ray tracing without needing to learn complex APIs like Vulkan or DirectX. The platform emphasizes a gradual introduction to ray tracing, making it suitable for both undergraduate and graduate courses. It supports the practical application of theoretical lessons, helping students understand key concepts such as ray-object intersections, shading models, and light transport in a visually interactive manner.

Moreover, Rayground supports multiple programming stages and provides live feedback, allowing students to see the results of their changes immediately, which enhances engagement and understanding. The platform is designed to be accessible across various devices and operating systems, as it only requires a WebGL-compliant browser, eliminating the need for specialized hardware or software setups.

However, Rayground does not yet support certain advanced ray tracing techniques like bidirectional path tracing or photon mapping, and it lacks support for animations and more complex, real-time interactions, which are important for advanced image synthesis. These limitations are partly due to the constraints of current web-based graphics technology.

In conclusion, Rayground provides an effective platform for introducing ray tracing to students, offering a simplified, interactive approach to understanding fundamental concepts. While it has some limitations in terms of advanced features, its accessibility and ease of use make it a valuable tool for computer graphics education.

2.1.7 pyGANDALF

The pyGANDALF Petropoulos et al. (2024) framework introduces an educational tool designed to modernize the teaching of computer graphics by combining the Entity-Component-System (ECS) architecture with dual support for both legacy and modern APIs—OpenGL and WebGPU. The framework aims to simplify learning by offering a Python-based environment where students can explore a range of techniques, from basic rendering to advanced topics like Physically Based Rendering (PBR) and tessellation. pyGANDALF is designed for both teaching and research, making it suitable for real-world applications while maintaining a focus on accessibility.

The primary strength of pyGANDALF lies in its WebGPU integration, which provides students with access to cutting-edge graphics technologies like compute

shaders and ray tracing pipelines. This sets the framework apart from others by preparing students for the future of graphics programming, where APIs like WebGPU are expected to replace older ones like OpenGL. The dual API support allows students to seamlessly explore both WebGPU and OpenGL, gaining a better understanding of the differences between legacy and modern graphics APIs. Furthermore, the framework's use of Entity-Component-System (ECS) architecture helps manage complex scenes efficiently, reflecting real-world game development practices. By combining modern API access with the simplicity of Python, pyGANDALF strikes a balance between accessibility and advanced functionality, making it an ideal tool for teaching computer graphics.

On the downside, WebGPU's immaturity poses challenges. Since the API is relatively new, some features may be unstable, and ongoing changes to the API could disrupt the learning process. Additionally, the framework's editor, which significantly enhances the user experience for OpenGL users, does not yet support WebGPU, limiting the interactivity and learning experience for students focused on WebGPU. Another significant limitation is that pyGANDALF has not undergone a full in-class evaluation. Without comprehensive testing in an educational environment, it is difficult to fully assess its effectiveness in teaching, as well as its ease of use and integration into a structured curriculum.

In conclusion, pyGANDALF represents an exciting step forward for computer graphics education, particularly with its focus on WebGPU and modern API integration. The framework offers a flexible and accessible environment for exploring both legacy and cutting-edge graphics technologies. However, its reliance on a relatively new API and the absence of full classroom testing are areas that need further development to fully realize its potential in education.

2.1.8 The Value of Elements

The Elements framework Papagiannakis et al. (2023) offers a unique approach to teaching computer graphics by combining an Entity-Component-System (ECS) architecture with a scene graph structure. This combination facilitates better management of large, complex scenes and allows for parallel processing, which is crucial for rendering millions of objects in real time. One of the main advantages of Elements is its ability to provide both a high-level, conceptual understanding and hands-on practical experience for students. Unlike traditional frameworks that often present parts of the graphics pipeline as black-boxes, Elements encourages exploration by exposing all stages of the rendering pipeline. This white-box approach empowers students to understand and manipulate the core components of the pipeline, such as lighting, shading, and texture mapping.

Furthermore, Elements is built with educational needs in mind. It is a lightweight, open-source tool that is accessible to students with varying levels of experience in computer graphics. Through its integration with Python, the framework allows for rapid prototyping, helping students focus on core concepts without being overwhelmed by low-level implementation details. By offering structured, progressively challenging assignments, Elements enables students to build foundational skills in 3D rendering, animation, and GPU programming, while also providing room for advanced exploration, such as geometric algebra

and neural computing.

The framework's modular design ensures extensibility, allowing students and educators to introduce new components and systems easily. This feature makes Elements not only suitable for classroom teaching but also for research and rapid prototyping in areas like geometric deep learning and real-time scientific visualization. Its unit-tested libraries, including pyECSS, pyGLV, and pyEEL, offer a range of applications that extend beyond traditional graphics teaching and into fields like machine learning and immersive analytics.

In summary, the value of Elements lies in its ability to provide a holistic, flexible, and scalable learning environment that adapts to the evolving needs of computer graphics education. By integrating modern graphics concepts with educational tools, Elements effectively bridges the gap between theoretical knowledge and practical application, making it an essential resource for both students and educators.

As we move forward to Chapter 3, where the implementation of ECS in Elements is discussed in greater detail. We will delve into the specifics of the ECS architecture and its use in 3D animations, providing a closer look at the technical aspects of Elements and how it facilitates complex, real-time rendering in educational and research settings.

3 Elements framework: ECS in a Scene Graph

In this chapter we will focus on the Entity-Component-System (ECS) architecture and its implementation within the Elements framework. We will explore how ECS, combined with scene graphs, manages complex 3D scenes and real-time animations. The chapter will provide detailed explanations of the technical aspects of ECS, including its structure, functionality, and how it interacts with the Elements framework's key components, such as pyECSS and pyGLV, to enhance rendering, transformations, and scalability in 3D computer graphics.

3.1 Understanding ECS

The Entity Component System (ECS) Härkönen (2019) is a software architectural model primarily used in 3D applications and game development. This model decouples data from behavior, significantly simplifying the application development process.

Based on the principle of data-oriented design and composition, the ECS approach assigns components independently to entities. This contrasts with object-oriented design, where components are typically inherited from classes. The ECS approach, however, leads to better performance, especially in applications involving numerous objects, such as those used in physics-based simulations, while also offering enhanced maintainability and understanding of the application's objects.

More specifically, in an Entity Component System:

Entity: An entity refers to a versatile object used for general purposes. It stands for individual "things" in your game or application. An entity has neither behavior nor data; systems provide the behavior, and components store the data. Instead, it identifies which pieces of data belong together. Typically, it consists of only a unique identifier (unique ID).

Component: A component defines a particular attribute of an entity and holds the data required to represent that attribute. It is typically implemented using structures, classes, or associative arrays.

System: A system is a process that operates on all entities possessing the relevant components. It contains the logic or behavior that acts on the components stored in entities. Systems operate independently and can be specialized for specific tasks (e.g., rendering system, physics system).

It is important to note that, throughout this project, when referring to "Entity," "Component," or "System," we are specifically referencing the definitions outlined earlier in the thesis, rather than other potential interpretations of these terms.

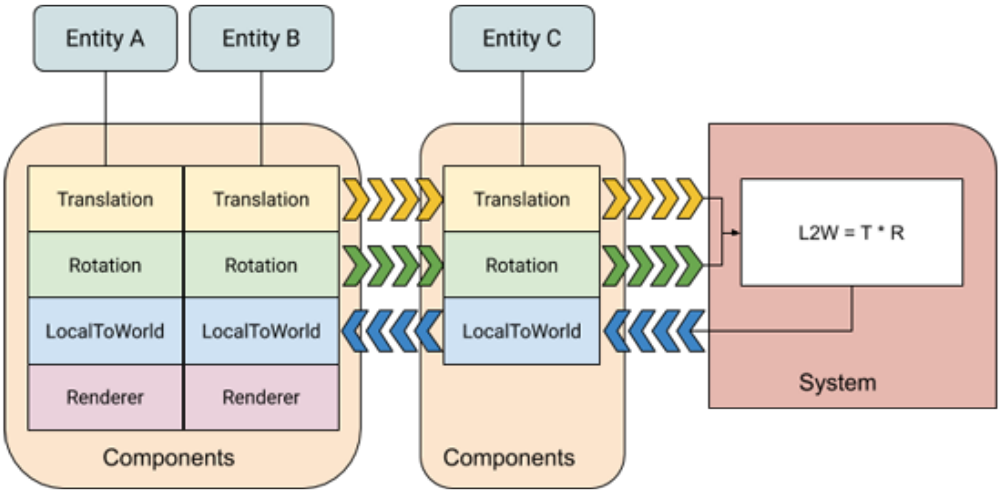


Figure 1: A system reads Translation and Rotation components, multiplies them, and then updates the corresponding LocalToWorld components (L2W = T*R) docs.unity3d.com (2024).

3.2 Graphics Programming with Scene Graphs

In modern game engines and graphics systems, we organize scenes using a Scene Graph, it is based on a hierarchical data structure, which is a non-circular one-way graph that is traversed to generate each frame. It holds data such as camera settings, materials, geometry and lighting details, which is necessary for creating the scene. Each node in the graph inherits attributes from its parent nodes, while the mesh data for an object is stored in the leaf nodes.

In most game engines, these nodes are commonly called gameobjects, actors or objects and the data associated with them are represented as Components. The Scene Graph edges define the relationships and hierarchy, while different traversals handle the initialization, updates, culling, and rendering processes.

In figure [2] we have a scene graph example to help us understand. The scene graph depicted follows the Entity-Component-System (ECS) architecture, illustrating the hierarchical structure and relationships between entities within a virtual environment. At the top of this hierarchy is the "World," which serves as the parent entity encompassing all other objects and components within the scene. Each object, such as the camera, light source, table, plate, and food, is represented as an entity with its own set of associated components that define its behavior and appearance.

"TRS" (Translation, Rotation, and Scaling) determine an entity's position, orientation, and size within the scene. "Shader" define how the entity interacts with light, determining its visual appearance, while "Mesh" represents the geometrical shape or structure of the object. These components allow the entity's behavior and properties to be modified or extended without altering the underlying system.

The scene graph also visually represents the parent-child relationships between entities, highlighting how transformations and properties are inherited through

the hierarchy. For instance, the plate is placed on the table, inheriting its spatial relationship, and the food, in turn, is positioned on the plate, following its transformations. This structure ensures that changes to parent entities, such as moving the table, automatically propagate to their children, maintaining relative positioning and coherence within the scene.

The design follows the principles of ECS by decoupling data and behavior into discrete, reusable components, ensuring each entity is defined by its specific set of components. This approach fosters a highly modular system, where components can be easily interchanged or extended, making the system adaptable to complex scene structures. The scene graph, therefore, serves as an effective means of organizing and rendering the objects within the virtual world, while maintaining clarity and modularity in how objects and their attributes are managed.

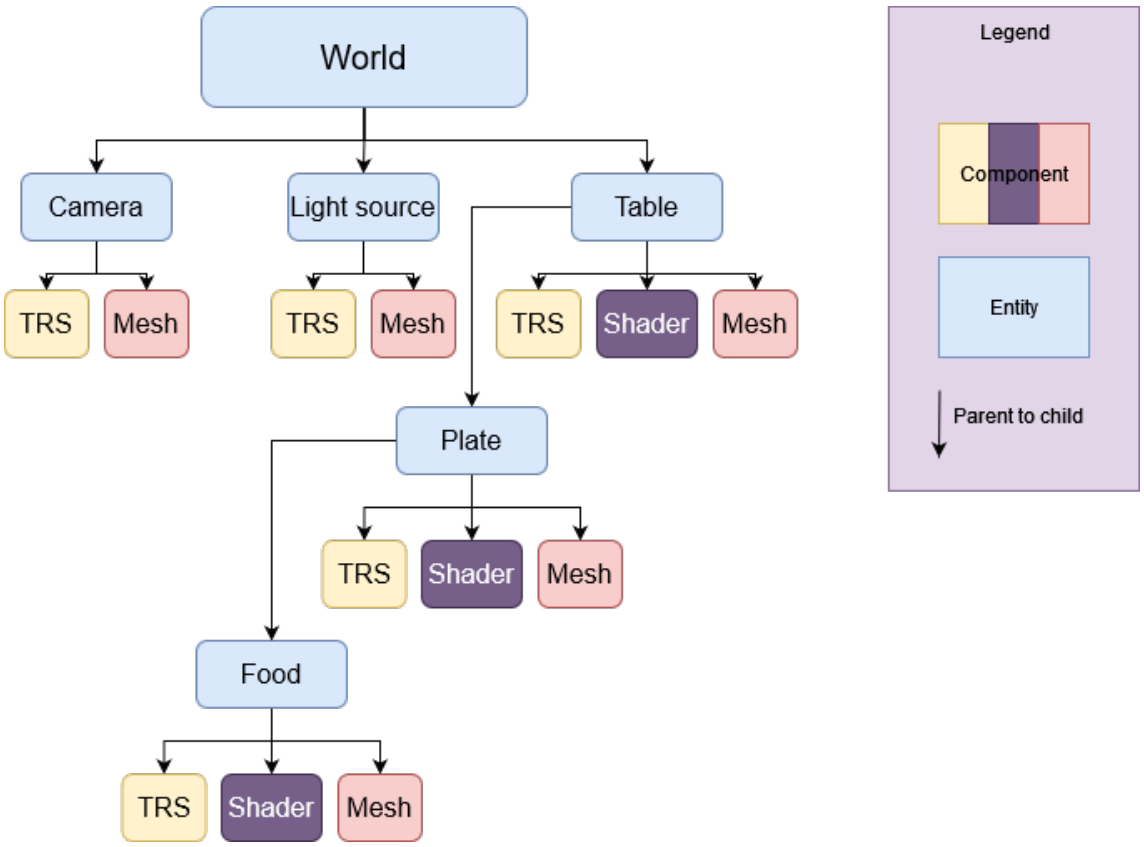


Figure 2: Example of a Scene Graph using ECS.

3.3 The Benefits of ECS

One of the main advantages of using ECS over other software design patterns is the improved scalability and flexibility it offers. Traditional object-oriented programming approaches can become increasingly complex and difficult to manage as a system grows and more features are added. In contrast, ECS separates the data and behavior of entities, allowing for easier modification and addition of new objects without the need to alter existing code. This facilitates the maintenance and expansion of the system, especially for large and complex applica-

tions.

Another advantage of ECS is the potential for performance improvements in certain types of applications. Because ECS stores data in contiguous memory blocks, it can be more cache-friendly, thus reducing memory fragmentation, leading to faster access times and improved performance.

Additionally, ECS can facilitate parallel processing and multithreading, allowing for better utilization of modern hardware and faster execution times. However, it should be noted that performance gains from ECS are not always guaranteed and depend on the specific application and implementation.

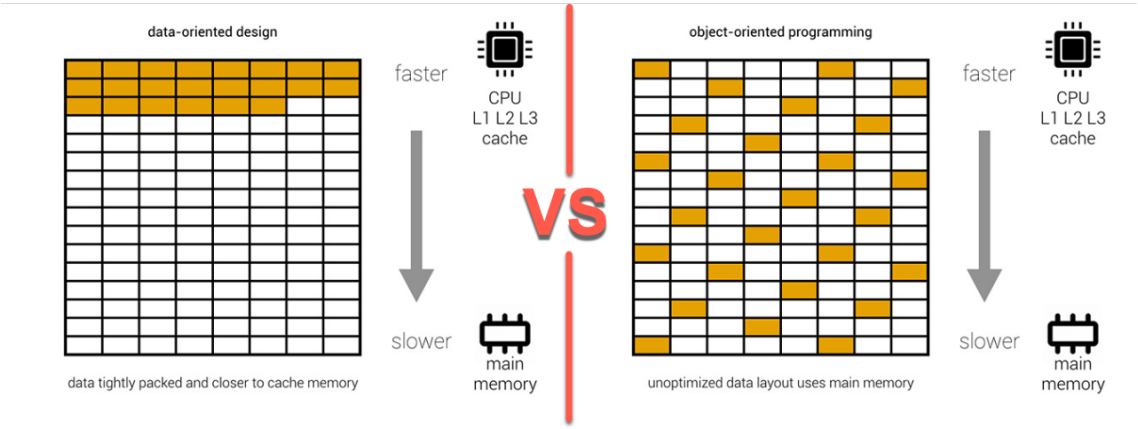


Figure 3: Example of data storage in ECS and object-oriented programming Lin (2020).

In general, the ECS model has gained significant attention in various game engines. Unity is currently restructuring its core game engine architecture towards adopting the DOTS (Data Oriented Technology Stack) system, which features ECS architecture. This shift aims to address violations caused by previous data-oriented programming principles and achieve better FPS performance in complex scenes.

3.4 Elements Framework Explained

Elements aims to combine the power of the Entity Component System (ECS) with the flexibility of Scene Graphs within the context of Computer Graphics. It also seeks to provide fundamental tools for anyone interested in topics related to Computer Graphics, such as Machine Learning, Geometric Algebra, and many more.

Following an educational approach accessible to individuals with minimal development experience, all related packages are available in Python.

The Elements project consists of three main Python libraries:

At the heart of the Elements project is the pyECSS library. This library serves as the foundation for the project by implementing the core aspects of the Entity-Component-System (ECS) model. It introduces the Scene Graph structure, which

is responsible for organizing and maintaining the various components within a 3D scene. The pyECSS library also employs a Geometric Algebra engine to handle complex transformations efficiently.

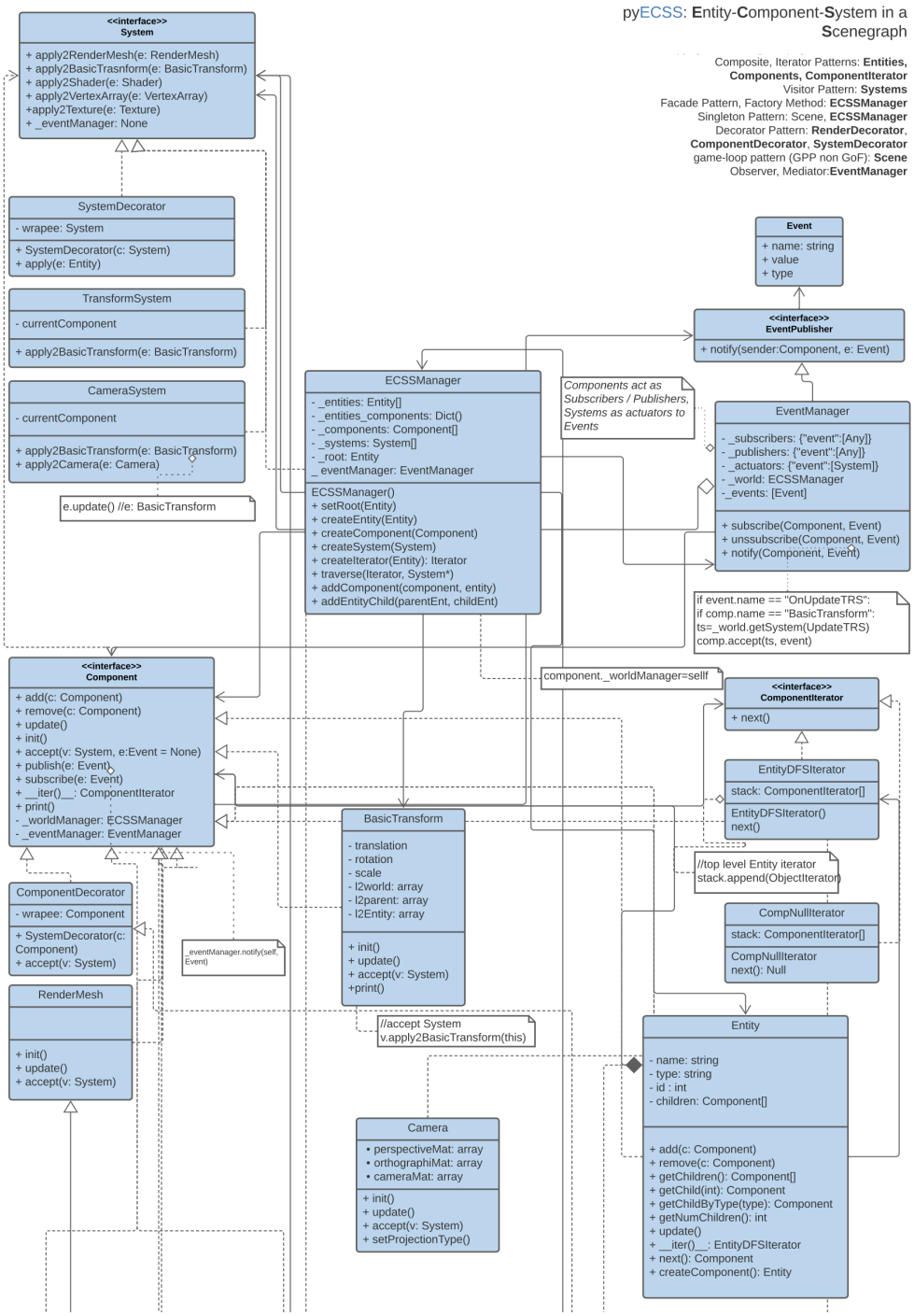


Figure 4: pyECSS class diagram Papagiannakis et al. (2023).

The pyGLV library complements pyECSS by offering graphical demonstrations of ECS within the Scene Graph. It contains numerous examples that illustrate how the ECS model can be applied to real-time rendering using OpenGL. This library is designed with a focus on cross-platform compatibility and applies sound software design principles. Through its integration with pyECSS, pyGLV showcases

how ECS can be utilized for both scientific visualization and advanced fields like geometric deep learning [5].

Within the pyECSS package, a variety of built-in components are provided to facilitate the construction of 3D scene graphs. Some of the key components [4] include:

BasicTransform: A component that manages an object's coordinate system relative to its parent entity. It combines translation, rotation, and scaling into a single matrix.

Camera: This component stores information about the camera's settings in the form of a view matrix. It can generate these matrices using orthogonal or perspective projection methods available in the pyECSS utilities.

RenderMesh: A component that handles the geometry of an entity, including vertex positions, face indices, and optionally, vertex colors and normals.

VertexArray: This component manages the vertex array object (VAO) and vertex buffer object (VBO), which are passed to the vertex shader for rendering.

Shader: A component that stores data required for OpenGL-GLSL shaders, including both vertex and fragment shaders.

In addition to these components, pyGLV includes several systems [5] that are crucial for handling different tasks in the computer graphics pipeline. These systems ensure the effective traversal and management of the scene graph, including the calculation of important transformation matrices such as model-to-world and root-to-camera matrices.

TransformSystem: This system traverses the scene graph and calculates the local-to-world matrix for each entity. It does this by multiplying the transformation matrices of all components in the scene graph, starting from individual entities and moving upward to the root node. This matrix multiplication order is consistent with what is typically taught in computer graphics courses.

CameraSystem: This system calculates the root-to-camera matrix, which is essential for the computer graphics rendering pipeline. It identifies the node with the camera component and returns the inverse of the model-to-world matrix for that entity.

InitGLShaderSystem: This system is responsible for initializing shader data and setting up the GPU for rendering, functioning outside of the main rendering loop.

RenderGLShaderSystem: This system handles the actual GPU rendering process. When an entity has both a VertexArray and Shader component, the system sends the vertex data to the GPU and renders it on the screen.

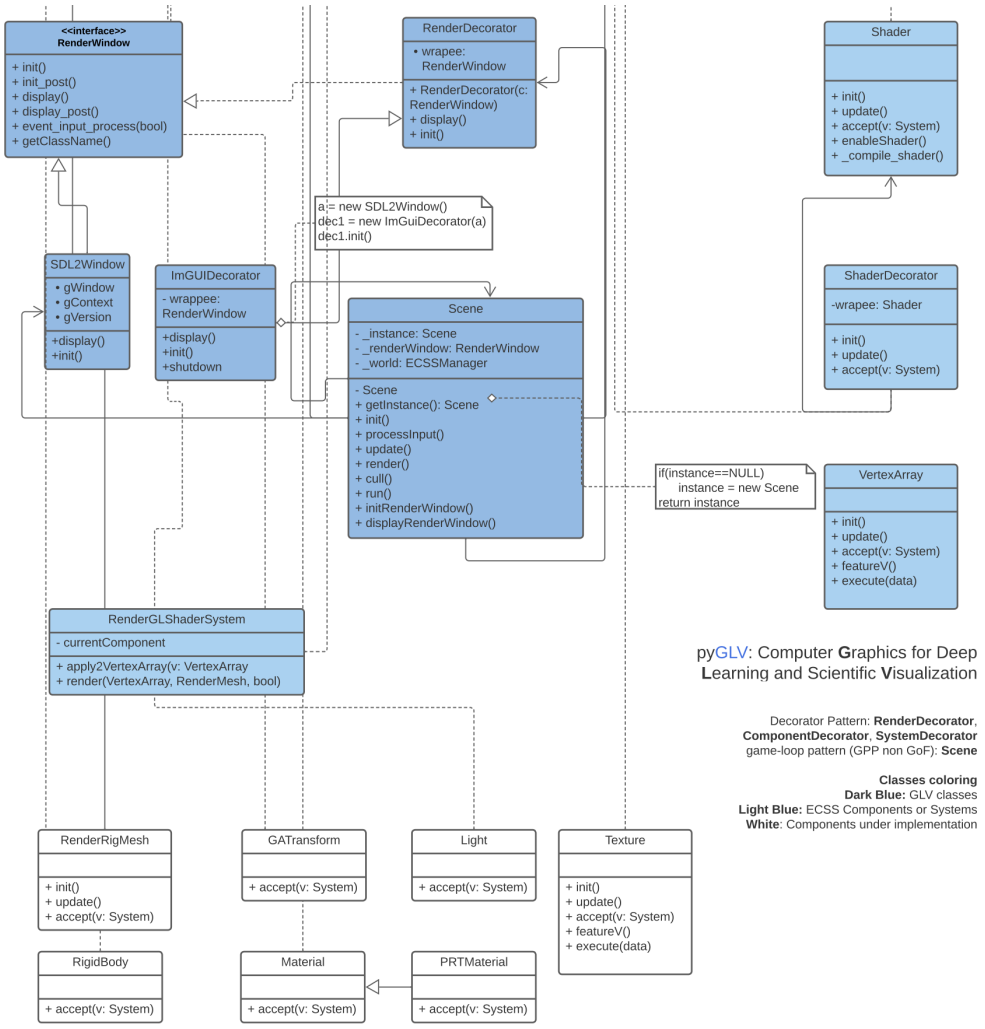


Figure 5: pyGLV class diagram Papagiannakis et al. (2023).

The pyEEL library promotes cutting-edge technology in neural networks, machine learning, and deep learning. It provides examples of how users can use ECS in a scene for deep learning.

In the thesis we present, we mainly use the pyECSS and pyGLV libraries as they contain the functionalities we need, such as ECS and various examples.

3.5 Singleton Design Pattern

The Singleton is a popular creational design pattern that ensures a class has only one instance, offering a global access point to it. It is used when just one instance of a class is needed to manage operations across a system. It achieves this by limiting the class to a single instantiation and providing a method for other parts of the application to access that instance universally. The key advantage of the pattern is that it maintains a single object instance, ensuring consistent behavior and efficient resource management throughout the system.

The Singleton pattern also facilitates the efficient management of shared resources. For example, in systems that require shared access to objects like

databases, configuration settings, or GPU resources, the Singleton pattern can centralize control, reduce memory usage, and prevent conflicting states from arising due to multiple instances of the same resource. Some of the key advantages of the Singleton pattern include global access, controlled resource management, and reduced memory footprint, especially in resource-intensive applications such as computer graphics or large-scale systems Gamma (1995).

```python
class Singleton:
    _instance = None  # Class attribute to store the single instance

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance

    def __init__(self):
        self.data = "This is a Singleton instance"

# Usage
singleton1 = Singleton()
singleton2 = Singleton()

print(singleton1.data)  # Output: This is a Singleton instance
print(singleton1 is singleton2)  # Output: True, both are the same instance
```

Listing 1: Python example of how a Singleton can be implemented.

In this [1] example, when the Singleton class is instantiated, it checks if an instance already exists. If not, it creates a new one; otherwise, it returns the existing instance. This pattern ensures that only one instance of the class can ever be created, regardless of how many times it's instantiated. This is highly effective for managing global states like resources or configuration settings in large systems, like the ones in the Elements framework.

In the Elements framework, the Singleton pattern plays a vital role in managing GPU resources, such as shaders, textures, and models. A single instance of the Singleton manages these resources, ensuring that they are loaded and used consistently throughout the application. This guarantees that GPU memory is optimized, preventing duplication of resource data, which can cause performance bottlenecks. By using the Singleton pattern, the Elements framework avoids conflicts that might arise from multiple instances attempting to access or modify the same resources.

Moreover, the Singleton pattern in Elements is responsible for managing global settings, including lighting parameters, rendering options, and ECS configurations. These global settings are crucial for maintaining uniform behavior across all components and systems in the application. Having a centralized point of control for these parameters ensures consistency and simplifies the process of synchronizing updates, rendering operations, and other system-wide processes in a complex 3D scene.

The use of the Singleton pattern in Elements also brings additional benefits, such as easier debugging and maintenance. With a single point of control, managing configurations, logging, and resource usage becomes more straightforward, contributing to a more robust and stable application. By preventing the duplication of resources and offering a centralized interface for managing these assets, the Singleton pattern enhances both the performance and maintainability of the

Elements framework.

4 Theoretical Foundations of ECS-Based Animation and GPU Acceleration

This section explores the key theoretical concepts that underpin the design and implementation of this project. We will delve into the principles of skinning and rigging, which are essential for animating 3D models, followed by an examination of textures and lighting, which play a critical role in creating visually rich and realistic scenes. Finally, we will explore the concept of animation itself, detailing the methods used to bring 3D models to life within the Entity-Component-System (ECS) framework, utilizing GPU acceleration to optimize performance and enhance realism.

4.1 Understanding Skinning and Rigging

Rigging refers to the process of creating a digital skeleton (rig) for a 3D model. The purpose of the rig is to provide a structure to control the model's movement, particularly for animation. The rig consists of joints, also known as bones, which are connected to form a skeletal structure. Each joint is assigned specific characteristics, such as rotation limits and hierarchical relationships.

Here [6] we illustrate a rigged 3D hand model, where the digital skeleton, is visible through highlighted joint structures. These joints, marked with green lines and nodes, represent the bones of the hand model, showing the interconnected system that enables controlled movement and animation. The joints are responsible for limiting and guiding the range of motion for each finger, as demonstrated by their placement along the fingers and knuckles. This setup exemplifies how a rig allows for precise manipulation of individual parts of the model, crucial for realistic animation and character articulation.

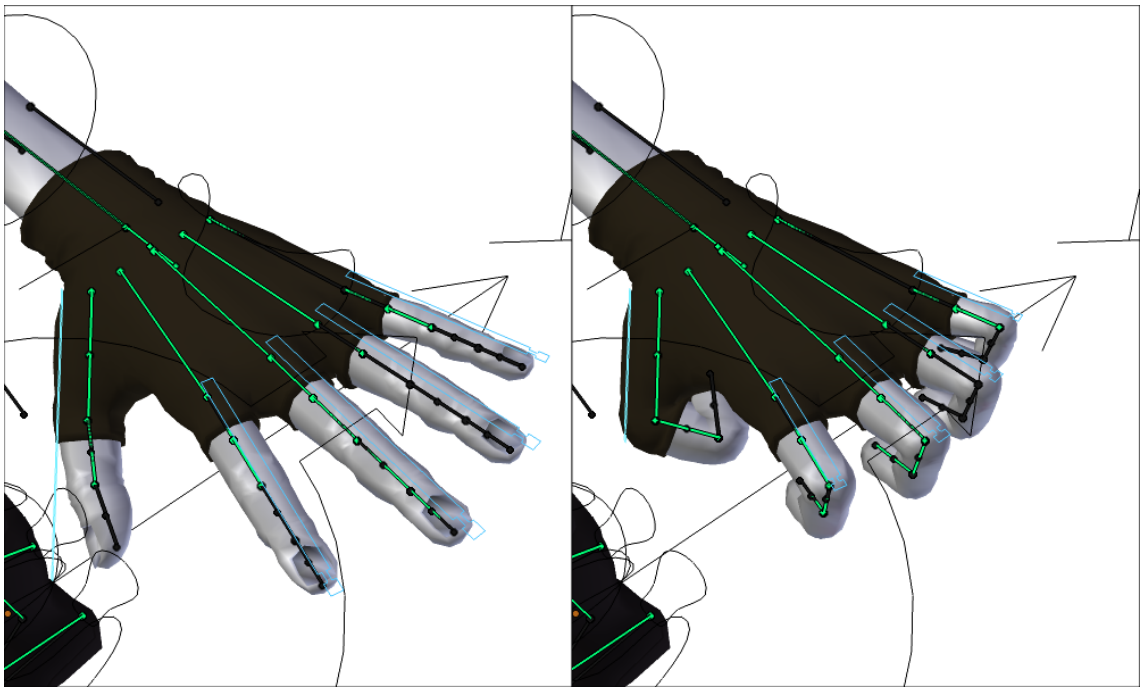


Figure 6: Rigged hand model Bundiuk (2023).

Skinning refers to the process of attaching a 3D model's mesh to a skeleton or set of bones. The purpose of skinning is to create realistic movement and deformation of the 3D model during animation. In this context, "skin" refers to the outer surface or mesh of the 3D model. The skinning process involves associating each vertex of the 3D mesh with one or more bones from the skeleton. This association is often expressed through a set of weights, which determine the influence of each bone on a particular vertex. When the skeleton moves, the vertices of the mesh move according to the transformations applied to the related bones, creating the illusion of realistic movement.

The image [7] demonstrates the process of skinning in a 3D model, where the mesh of lower body is visually attached to its underlying skeleton. The mesh, shown as a grid of green lines, represents the outer surface of the model. The bones, visible within the leg, are connected to the mesh vertices through weighted associations, which dictate how each section of the mesh will deform when the skeleton moves. As the bones shift, the corresponding parts of the mesh stretch and contract, simulating realistic movement. This depiction highlights how skinning links the visual mesh to the bones, creating dynamic and natural motion in animations.

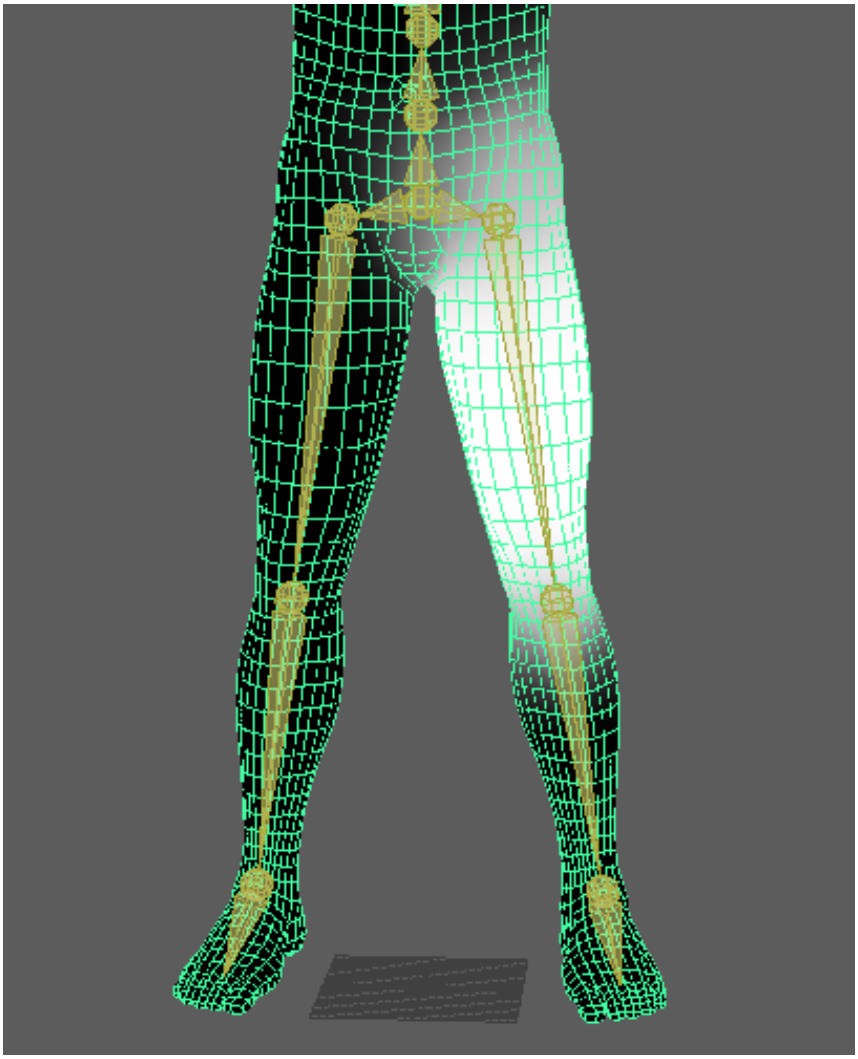


Figure 7: Low body with skin weights Insider (2019).

4.2 Textures

Texture refers to the visual appearance or surface quality of an object in a virtual 3D environment. Textures are used to add detail, realism, and complexity to the surfaces of 3D models. They simulate different materials when applied to objects.

Textures can have one to three dimensions, although two-dimensional textures are the most common. Textures are mapped onto the three-dimensional surfaces of objects. The process of applying these two-dimensional images to 3D models is known as texture mapping. The texture coordinates of the 3D model's surface are used to determine how the pixels of the texture image should be mapped onto the model.

The image [8] illustrates the process of texture mapping in 3D modeling, where a two-dimensional texture is applied to a 3D house model. In the top left, the model is depicted as a wireframe, showing its geometric structure without any textures. Next to it, the model is displayed in its basic, untextured form, demonstrating its plain appearance. Below, two-dimensional textures are shown as flat images, representing various surfaces (such as the roof and walls) that will be applied to the model. Finally, the fully textured 3D model is presented, where the 2D images wrap around the surfaces of the house, giving it a detailed, realistic look.

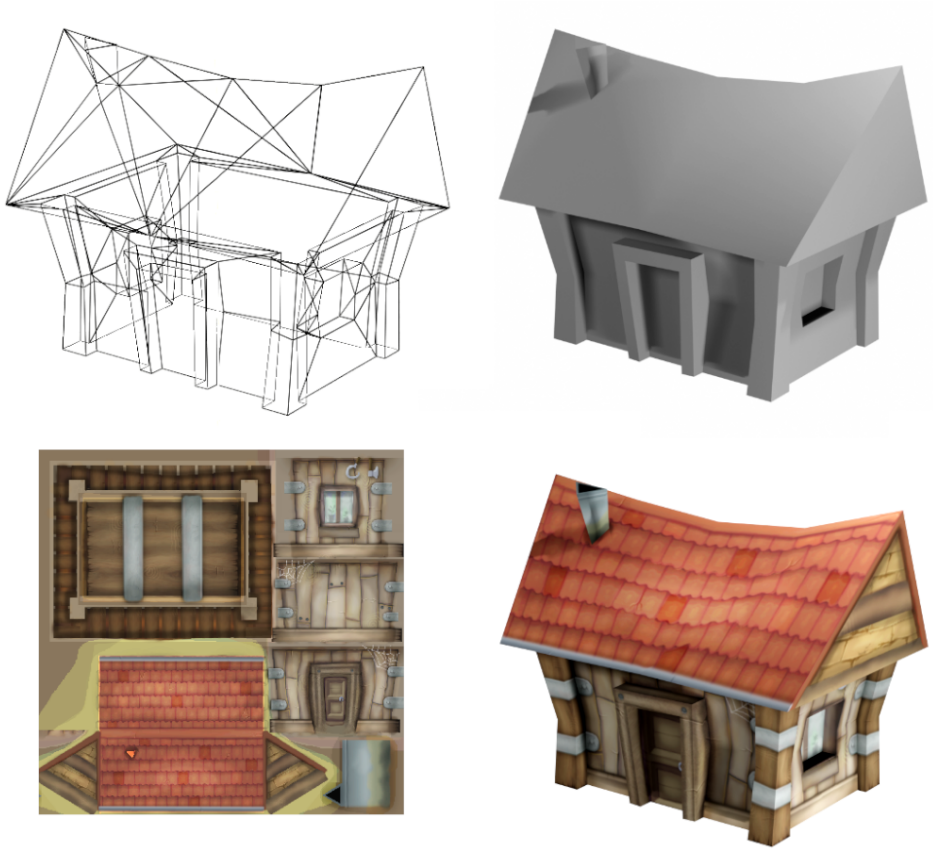


Figure 8: Mapping two-dimensional texture onto a 3D model of a house Wikipedia (2024b).

4.3 Phong Lighting Algorithm

Lighting refers to the simulation of how light interacts with objects in a virtual environment. Proper lighting is essential for creating realistic and visually appealing images. Lighting affects how colors, shadows, and highlights appear on the surfaces of 3D models. It plays a crucial role in conveying the shape, depth, and texture of objects in a scene.

In our code, we use the Phong lighting algorithm, which consists of three main components: ambient, diffuse, and specular reflection.

Ambient reflection represents the light that is scattered in all directions, providing a base level of illumination to a surface regardless of its orientation. It is a constant term applied uniformly to all points on a surface, independent of the direction of incoming light.

Diffuse reflection simulates the matte, non-glossy appearance of surfaces. It depends on the angle between the incoming light and the surface normal. The intensity of diffuse reflection is calculated using Lambert's cosine law, which states that the intensity is proportional to the cosine of the angle between the light direction and the normal vector.

Specular reflection represents the shininess or glossiness of a surface. It depends on the angle between the reflected light direction and the viewer's line of sight. The intensity of specular reflection is higher when the viewer is aligned with the reflection direction, creating highlights.

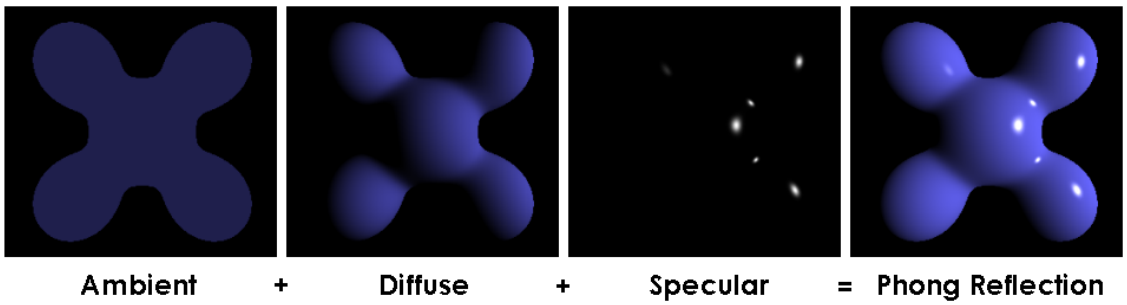


Figure 9: Visual representation of the Phong equation Wikipedia (2024a).

Specifically, our work uses the following Phong algorithm:

$$\text{Ambient} = \text{ambientStr} \times \text{ambientColor} \tag{1}$$

$$\text{Diffuse} = \max(\text{dot}(\text{norm}, \text{lightDir}), 0.0) \times \text{lightColor} \tag{2}$$

$$\text{Specular} = \text{shininess} \times (\max(\text{dot}(\text{viewDir}, \text{reflectDir}), 0.0))^{32} \times \text{tex.xyz} \tag{3}$$

$$\text{OutputColor} = (\text{Ambient} + (\text{Diffuse} + \text{Specular}) \times \text{lightIntensity}) \times \text{tex.xyz} \tag{4}$$

Where:

- ambientStr: Intensity of ambient light.
- ambientColor: Color of ambient light.
- norm: Normalized normal vector.
- lightDir: Normalized vector for the direction of the light.
- viewDir: Normalized vector for the viewing direction.
- reflectDir: Normalized reflection direction vector.
- lightPos: Position of the light source.
- lightColor: Color of the light source.
- lightIntensity: Intensity of the light source.
- shininess: Material shininess.
- tex: Texture.

4.4 Animation

Animation is the process of creating the illusion of motion by displaying a sequence of images or frames. In computer graphics, this involves manipulating objects, characters, or scenes to create movement or change over time. Animation is crucial in fields such as film, video games, simulations, and education, enhancing interactive experiences, visual storytelling, and real-time simulations.

Key principles, initially developed for traditional animation, ensure that movements appear natural and believable, and they have been adapted for digital media, including 3D animation. For example, "squash and stretch" gives objects flexibility and weight, making motion dynamic. Anticipation sets up actions by preparing the audience for movement (e.g., winding up before a jump). Timing and spacing control movement speed, while follow-through and overlapping action account for inertia, ensuring realistic physics. Exaggeration highlights key actions, making them more dramatic or noticeable.

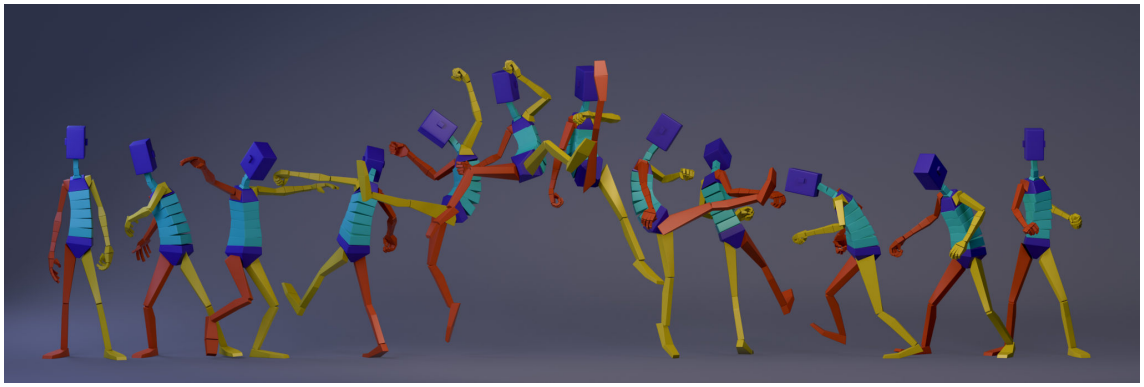


Figure 10: Keyframes of animation Academy (2021).

In keyframe animation, keyframes [10] define the object's position, rotation, and scale at specific points in time. Between these, smooth transitions are achieved using interpolation techniques like linear interpolation (LERP) and spherical linear interpolation (SLERP).

LERP is a method for linear transitions between two points, calculated as:

$$L(t) = (1 - t) \cdot A + t \cdot B \tag{5}$$

where AA and BB are the start and end points, and tt is the interpolation factor between 0 and 1. It is ideal for moving objects along straight paths or gradually changing values like position or scale.

SLERP is used to interpolate between rotations, especially useful for smooth transitions in 3D space. Given two orientations represented as quaternions q0q0 and q1q1, the SLERP equation is:

$$S(t) = \frac{\sin{(1 - t)\theta}}{\sin{\theta}} \cdot q_0 + \frac{\sin{t\theta}}{\sin{\theta}} \cdot q_1 \tag{6}$$

where θ is the angle between the quaternions. SLERP ensures smooth and realistic rotational movements.

In 3D computer graphics, animation is achieved through transformations—translation, rotation, and scaling. Each object in a scene has a transformation matrix that encapsulates these operations:

$$T = T_{\text{translation}} \times T_{\text{rotation}} \times T_{\text{scaling}} \tag{7}$$

During animation, this matrix changes over time, resulting in the object's movement or deformation. The Entity-Component-System (ECS) architecture used in this project helps manage these transformations efficiently by separating the data (e.g., position, velocity) from the logic (e.g., how keyframes are updated).

With modern 3D animation's increasing complexity, GPU acceleration is essential for real-time performance. By offloading heavy computations, like rendering and physics simulations, to the GPU, the system can handle complex scenes while maintaining smooth animations and high visual fidelity. This is crucial in applications like video games and simulations, where performance and visual quality must be balanced.

5 Project Implementation

In this chapter, we will discuss how we designed and implemented the thesis programmatically. We will explore how we created new functions, classes, and new components, entities, and systems, and how we combined them with the existing Elements objects to produce a scene with an animated, lit, and textured model.

In more details, the main focus of the project is on animating two 3D models: a robotic arm and an AstroBoy figure, both of which are rigged to allow for realistic movement. The animation is controlled through a sequence of keyframes, each representing a snapshot of the model's position, rotation, and scaling at a particular point in time. By interpolating between these keyframes using techniques like Linear Interpolation (LERP) and Spherical Linear Interpolation (SLERP), the system produces smooth transitions, resulting in lifelike motion for the models. These animations are further enhanced by the use of Phong lighting, a shading model that combines ambient, diffuse, and specular reflection to simulate realistic lighting effects.

Additionally, the project makes extensive use of GPU acceleration to handle the computationally expensive tasks of rendering and animation. By offloading these tasks to the GPU, the system is able to maintain high frame rates and visual fidelity, even when animating complex scenes with detailed textures and lighting effects. This is particularly important in real-time applications, where any drop in performance could negatively impact the user experience. The ECS framework's inherent ability to separate data (components), behavior (systems), and entities (identity) allows for easier optimization and parallelization of these processes.

The integration of these technologies—ECS architecture, GPU acceleration, and the Elements toolkit—demonstrates a modern approach to 3D animation systems. The project provides valuable insights into how modular design and efficient resource management can be used to create scalable, flexible, and performant systems. More importantly, it underscores the potential of the Elements framework as both an educational platform and a tool for rapid prototyping, giving students and developers the opportunity to explore advanced computer graphics concepts in a practical and interactive way.

5.1 Project overview

This section describes the specific implementation aspects of the project that we developed. The project is based on the Entity-Component-System (ECS) framework, but focuses primarily on the coding and integration of various components within a graphical environment. The main goal was to create a functional scene that showcases different graphical elements, including a rigged model, camera, lighting, and a variety of objects.

In the scene, the user can interact with the models, specifically the robotic arm or the Astroboy figure, by modifying their Translation, Rotation, and Scaling (TRS) properties or altering their animation sequences. Additionally, the user can interact with various elements of the environment, such as the floor, camera, skybox,

and lighting. This level of control allows users to experiment and gain a deeper understanding of how graphics operate, and more specifically, how animation functions within the context of this project. All of these interactions are facilitated through a graphical user interface (GUI).

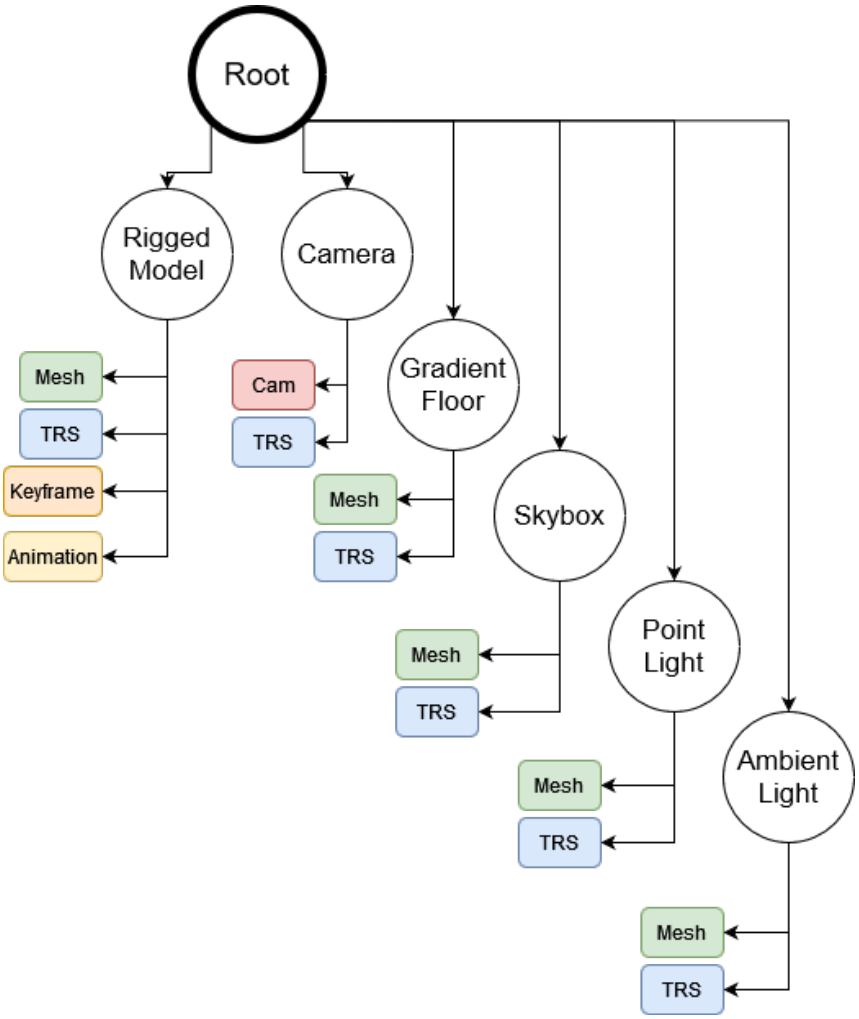


Figure 11: ECS Project graphical enviroment.

In the diagram [11], the hierarchical structure of entities and their associated components is shown. Each entity, represented by circles (such as the Camera, Gradient Floor, Skybox, Point Light, and Ambient Light), is composed of different components. These components, displayed as colored rectangles, include TRS (Transformation), Mesh, Camera settings (Cam), Keyframes, and Animation. The Root entity serves as the parent node, with its child entities like the Rigged Model, Camera, and various lights arranged in a hierarchical manner. This setup demonstrates how the scene graph is managed, and how different graphical components interact to form the overall visual representation of the project.

The key part of my project was to code the functionalities of these entities, ensuring that they are properly rendered and animated within the ECS framework. By using GPU acceleration, I was able to optimize the rendering pipeline, allowing for smooth real-time visual feedback. The rigged model, for instance, is animated by applying keyframe transformations to its skeletal structure, while the

lighting entities (Point Light and Ambient Light) dynamically illuminate the scene based on their positions and properties.

5.2 Installation of Elements and Prerequisites

For the proper functioning of Elements and its libraries, the installation of certain specific applications is required.

It is important to note that this entire thesis was conducted on a Windows 10 environment. Although we have not performed tests on any Linux distributions or macOS, all the packages and libraries utilized in this thesis are compatible with these operating systems and can be installed accordingly.

We need to install the programming language Python on which the Elements code is based. Next, we need Anaconda as Elements uses several of its libraries. For editing, writing, and general management of our code, we use Visual Studio Code as it is modular and open-source.

Once the prerequisites have been installed, we can proceed with the installation of Elements. Detailed installation instructions for Elements can be found here.

5.2.1 Installation of pyassimp

For the installation of pyassimp, we must first have installed the prerequisites and Elements. Once these steps are completed, we should be in the anaconda environment we created—how to create and open an anaconda environment is detailed in the Elements installation instructions. Then, we need to run the command `"pip install pyassimp==4.1.3"` in the terminal.

We use version 4.1.3 because it has been extensively tested during the development of the thesis and works without errors.

Next, we need to check if the versions of assimp and numpy are compatible with pyassimp. Running the command `"conda list"` in the terminal will display all installed packages in the current environment. The versions should be assimp = 4.1.0 and numpy = 1.24.4. If they are not, we need to install the correct versions with the commands `"pip install numpy==1.24.4"` and `"conda install assimp=4.1.0=h0536686_2"`

5.3 Model Data Extraction

In our application, we utilize two models: the AstroBoy model, which we did not design, and a robotic arm model, which we designed ourselves. Using pyassimp, we extract the data (mesh, vertices, indices, bones) from both models.

As shown in the code snippet below[2], in the first line, we use the loadfunction, which is a pyassimp function that extracts all the data from the file containing our model. The str(file) essentially represents the location of our file in the computer's storage.

In the fifth line, we load the model into the variable mesh. The `mesh_id` is needed to extract the correct model from the file, as there may be other models in the

same file. The remaining lines are used to extract the vertices, faces/indices, and bones of the model.

```
    figure = load(str(file))

    mesh_id = 3

    mesh = figure.meshes[mesh_id]
    v = mesh.vertices
    f = mesh.faces
    b = mesh.bones
```

Listing 2: Code for extracting data from our model.

In the following code snippet[3] we initialize the vertex_weight class, which is a class from the Elements library, in order to distribute the vertex weights to the bones.

Then, from lines six to eleven, we change the color of the vertices so that their color changes based on their y-coordinate.

In line thirteen, we flatten the array f to make it easier to use later. The transform variable is used later and is always set to True, as it selects between different codes in Elements.

```
    vw = vertex_weight(len(v))
    vw.populate(b)

    v2 = np.concatenate((v, np.ones((v.shape[0], 1))), axis=1)

    c = []
    min_y = min(v, key=lambda v: v[1])[1]
    max_y = max(v, key=lambda v: v[1])[1]
    for i in range(len(v)):
        color_y = (v[i][1] - min_y) / (max_y - min_y)
        c.append([0, color_y, 1-color_y , 1])

    f2 = f.flatten()

    transform = True
```

Listing 3: Code for extracting data from our model.

In the code snippet[4], we have the initialize_M function, which initializes a list of 4x4 identity matrices. The length of this list is the number of bones in our model. In lines 8, 17, 18, 28, and 29, we initialize the initial poses that each keyframe will have.

In line 37, we initialize BB, which consists of 4x4 matrices, where each 4x4 matrix has the initial Translation Rotation Scale (TRS) from each bone.

In line 41, we reshape BB so that it can be used later.

The remaining lines of code will be analyzed in the appropriate chapter.

```
    M = initialize_M(b)

    WW = np.array([[np.eye(4)
        for _ in range(len(M))]
            for _ in range(3)]);

    WW_9 = [[[0] * 6 + [1, 1, 1]
        for _ in range(len(M))]
            for _ in range(3)];

    #Initialising first keyframe
```

```
    M[1] = np.dot(np.diag([1,1,1,1]),M[1]);

    keyframe1.array_MM.append(read_tree(figure,mesh_id,M,transform));

    for i in range(0,len(WW_9[0])):
        WW_9[0][i][:3] = translation(keyframe1.array_MM[0][i])
        WW_9[0][i][3:6] = (rotationEulerAngles(keyframe1.array_MM[0][i])/180)*np.pi
        WW_9[0][i][6:9] = scale(keyframe1.array_MM[0][i])

    #Initialising second keyframe
    M[1][0:3,0:3] = eulerAnglesToRotationMatrix([0.3,0.3,0.4])
    M[1][0:3,3] = [0.5,0.5,0.5]

    keyframe2.array_MM.append(read_tree(figure,mesh_id,M,transform))
    for i in range(0,len(WW_9[0])):
        WW_9[1][i][:3] = translation(keyframe2.array_MM[0][i])
        WW_9[1][i][3:6] = (rotationEulerAngles(keyframe2.array_MM[0][i])/180)*np.pi
        WW_9[1][i][6:9] = scale(keyframe2.array_MM[0][i])


    if keyframe3 != None:
        M[1][0:3,0:3] = eulerAnglesToRotationMatrix([-0.5,0.3,0.4])
        M[1][0:3,3] = [0.5,0.5,0.5]
        keyframe3.array_MM.append(read_tree(figure,mesh_id,M,transform))

        for i in range(0,len(WW_9[0])):
            WW_9[2][i][:3] = translation(keyframe3.array_MM[0][i])
            WW_9[2][i][3:6] = (rotationEulerAngles(keyframe3.array_MM[0][i])/180)*np.pi
            WW_9[2][i][6:9] = scale(keyframe3.array_MM[0][i])

    #Initialising BB array
    BB = [b[i].offsetmatrix for i in range(len(b))]

    # Flattening BB array to pass as uniform variable
    ac.bones.append(np.array(BB, dtype=np.float32).reshape((len(BB), 16)))
```

Listing 4: Code for extracting data from our model.

5.4 Implementation of Keyframe Component

For the implementation of animation, we will use the ECS architecture and the Elements library extensively, as well as create new components and systems.

Starting with the keyframe component [5], this is where the 4x4 matrices defining the TRS of each bone will be stored.

```
class Keyframe(Component):

    def __init__(self, name=None, type=None, id=None, array_MM=None):
        super().__init__(name, type, id)

        self._parent = self
        if not array_MM:
            self._array_MM = []
        else:
            self._array_MM = array_MM

    @property
    def array_MM(self):
        return self._array_MM

    @array_MM.setter
    def array_MM(self, value):
        self._array_MM = value
```

Listing 5: Code for the keyframe component.

5.5 Implementation of Animation Component

In the following code, we have the AnimationComponents component class[6], where we initialize the attributes of the animation. For keyframe and bones, we pass the corresponding matrices. The MM is the final matrix that we will later pass as a uniform variable to the shaders. The alpha is the interpolation factor ranging from 0 to 1. The tempo dictates the rate of change of the alpha. The time_add is for storing the current time. The animation_start dictates whether the model starts/stops moving. The time is for the timeline of the keyframes. The flag is used to have repetitive animation, and inter is the switch between SLERP or LERP.

```
class AnimationComponents(Component):

    def __init__(
                self,
                name=None,
                type=None,
                id=None,
                keyframe=None,
                bones=None,
                MM=None,
                alpha=0,
                tempo=2,
                time_add=0,
                animation_start = True,
                anim_keys = 2,
                time = [0, 100, 200],
                flag = True,
                inter = 'SLERP'):

        super().__init__(name, type, id)
        self._parent = self

        self.alpha = alpha
        self.tempo = tempo
        self.time_add = time_add
        self.anition_start = animation_start
        self.animKeys = anim_keys
        self.inter = inter
        self.time = time
        self.flag = flag

        self.MM = []

        if not keyframe:
            self._keyframe = []
        else:
            self._keyframe = keyframe

        if not bones:
            self._bones = []
        else:
            self._bones = bones

    @property
    def bones(self):
        return self._bones

    @bones.setter
    def bones(self, value):
        self._bones = value
```

Listing 6: Code for initializing attributes of the animation component.

Initially, the animation_loop[7] initializes a self.MM array with identity 4x4 matrices, where the intermediate frame generated by interpolation will be stored.

Then, it evaluates whether the motion should continue, taking into account the current time (time_add) and the intervals between keyframes. It calls the animation_for_loop method, facilitating interpolation between keyframes and updating transformation matrices for each bone or joint in the animation.

The function handles timing issues and updates a flag for animation repetition. If the animation is set to repeat, time progresses at a specified rate; otherwise, it remains constant. This dynamic timing mechanism ensures smooth transitions between key frames.

In conclusion, the function finalizes by flattening the self.MM array, converting it into a format suitable for passing to a uniform variable. The animation_loop plays a central role in orchestrating the animation process, enabling dynamic and visually appealing sequences.

```
    def animation_loop(self):
            #Filling MM with 4x4 identity matrices
            self.MM = [np.eye(4)
                for _ in self.keyframe[0]]

            if (self.time_add >= self.time[1]
                and self.keyframe[2] is None)
                or (self.time_add >= self.time[2]):
                    self.flag = False

            elif self.time_add <= self.time[0]:
                self.flag = True


            if self.time_add >= self.time[0]
                and self.time_add <= self.time[1]:
                    self.animation_for_loop(self.keyframe[0],
                                            self.keyframe[1],
                                            self.time[0],
                                            self.time[1])

            elif self.time_add > self.time[1]
                and self.time_add <= self.time[2]
                and self.keyframe[2] is not None:
                    self.animation_for_loop(self.keyframe[1],
                                            self.keyframe[2],
                                            self.time[1],
                                            self.time[2])


            #So we can have repeating animation
            if self.flag == True:
                if self.anition_start == True:
                    self.time_add += self.tempo
                else:
                    self.time_add = self.time_add
            else:
                if self.anition_start == True:
                    self.time_add -= self.tempo
                else:
                    self.time_add = self.time_add

            # Flattening MM1 array to pass as uniform variable
            self.MM =  np.array(self.MM, dtype=np.float32).reshape((len(self.MM), 16))

            return self.MM
```

Listing 7: The function is responsible for managing the animation loop. It updates the animation state, interpolates between keyframes and returns a flattened array of 4x4 matrices representing the transformation for each bone in the animation.

The function[11] calculates the alpha, taking into account the current time relative to the intervals between keyframes (8).

$$\text{self.alpha} = \frac{\text{self.time\_add} - t_0}{|t_1 - t_0|} \tag{8}$$

Then, the algorithm extracts translation [8], rotation[9], and scale[10] elements from the provided keyframes.

```
def translation(matrix):
    return matrix[:3,3];
```

Listing 8: Code that extracts translation.

```
def rotationEulerAngles(matrix):
    # First get rotation matrix from trs. Divide by scale
    rotationMatrix = matrix.copy();
    sc = scale(matrix);
    rotationMatrix = rotationMatrix @ util.scale(1/sc[0], 1/sc[1], 1/sc[2])
    myR = rotationMatrix[:3,:3]
    if myR[2,0] not in [-1,1]:
        y = -np.arcsin(myR[2,0]);
        x = np.arctan2(myR[2,1]/np.cos(y), myR[2,2]/np.cos(y));
        z = np.arctan2(myR[1,0]/np.cos(y), myR[0,0]/np.cos(y));
    else:
        z = 0;
        if myR[2,0] == -1:
            y = np.pi/2;
            x = z + np.arctan2(myR[0,1], myR[0,2]);
        else:
            y = -np.pi/2;
            x = -z + np.arctan2(-myR[0,1], -myR[0,2]);
    return np.array([x,y,z])*180/np.pi;
```

Listing 9: Code that extracts rotation.

```
def scale(matrix):
    m = matrix.copy()[:3,:3];
    A = m.transpose() @ m
    sx = np.sqrt(A[0,0])
    sy = np.sqrt(A[1,1])
    sz = np.sqrt(A[2,2])
    return numpy.array([sx, sy, sz])
```

Listing 10: Code that extracts scale.

5.5.1 Scale

The extraction of scale can be mathematically represented as follows:

We are given a matrix **M** where the top-left is a 3×3 submatrix **m**,

$$\mathbf{m} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}, \tag{9}$$

Then we have the matrix **A**, which is calculated as follows:

$$\mathbf{A} = \mathbf{m}^T\mathbf{m}. \tag{10}$$

The matrix **m** is the Rotation(R) matrix multiplied with the Scale(S) matrix.

$$(RS)^T(RS) = S^2 \tag{11}$$

We get $\mathbf{S}^2$ because the R is orthonormal matrix and the S is a diagonal matrix.

The scaling factors s_x, s_y, and s_z are then determined as:

$$s_x = \sqrt{A_{11}}, \quad s_y = \sqrt{A_{22}}, \quad s_z = \sqrt{A_{33}}. \tag{12}$$

The function returns a matrix containing these scaling factors:

$$\text{scale}(\mathbf{M}) = \begin{bmatrix} s_x \\ s_y \\ s_z \end{bmatrix}. \tag{13}$$

5.5.2 Rotation

The rotation matrix in Euler angles can be mathematically represented as follows.

Given a rotation matrix **R**, the function calculates the Euler angles x, y, and z as follows:

1. We normalize the rotation matrix:

$$\begin{aligned} \text{rotationMatrix} &= \mathbf{R} \\ \text{sc} &= \text{scale}(\mathbf{R}) \\ \text{rotationMatrix} &= \text{rotationMatrix} \times \text{util.scale}\left(\frac{1}{\text{sc}[0]}, \frac{1}{\text{sc}[1]}, \frac{1}{\text{sc}[2]}\right) \end{aligned}$$

2. We extract the rotation submatrix:

$$\text{myR} = \text{rotationMatrix}[:3, :3]$$

3. We calculate the Euler angles:

$$\begin{aligned} y &= -\arcsin(\text{myR}[2,0]) \\ x &= \arctan\left(\frac{\text{myR}[2,1]}{\cos(y)}, \frac{\text{myR}[2,2]}{\cos(y)}\right) \\ z &= \arctan\left(\frac{\text{myR}[1,0]}{\cos(y)}, \frac{\text{myR}[0,0]}{\cos(y)}\right) \end{aligned}$$

4. Conversion to degrees:

$$\text{Euler angles} = [x, y, z] \times \frac{180}{\pi}$$

5.5.3 Euler Angles

The Euler angles to quaternions conversion can be mathematically represented as follows:

Given the Euler angles roll, pitch, and yaw, the function calculates the corresponding quaternion (q_x, q_y, q_z, q_w) as:

$$
\begin{aligned}
\text{roll} &\leftarrow \frac{\text{roll}}{2.0} \\
\text{pitch} &\leftarrow \frac{\text{pitch}}{2.0} \\
\text{yaw} &\leftarrow \frac{\text{yaw}}{2.0} \\
cy &\leftarrow \cos(\text{yaw}) \\
sy &\leftarrow \sin(\text{yaw}) \\
cp &\leftarrow \cos(\text{pitch}) \\
sp &\leftarrow \sin(\text{pitch}) \\
cr &\leftarrow \cos(\text{roll}) \\
sr &\leftarrow \sin(\text{roll}) \\
qw &\leftarrow cr \cdot cp \cdot cy + sr \cdot sp \cdot sy \\
qx &\leftarrow sr \cdot cp \cdot cy - cr \cdot sp \cdot sy \\
qy &\leftarrow cr \cdot sp \cdot cy + sr \cdot cp \cdot sy \\
qz &\leftarrow cr \cdot cp \cdot sy - sr \cdot sp \cdot cy
\end{aligned}
$$

The resulting quaternion is (q_x, q_y, q_z, q_w).

5.5.4 Translation

The extraction of translation can be mathematically represented as follows:

Given a transformation matrix **M**, the function calculates the translation vector (t_x, t_y, t_z) as

$$
\text{translation}(\mathbf{M}) = \mathbf{M}[:3, 3] = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}
$$

5.5.5 SLERP - Quaternions

For interpolation between rotations, quaternion interpolation is used (either LERP(15) or SLERP(14))

$$
\text{quaternion\_slerp}(q_1, q_2, t) = \begin{cases} q_1 & t = 0.0 \\ q_2 & t = 1.0 \\ q_1 \cos(\theta) + \frac{q_2 - q_1 \cos(\theta)}{\sin(\theta)} \sin(\theta t) & \end{cases} \tag{14}
$$

Here, q_1 and q_2 are unit quaternions, θ is the angle between q_1 and q_2, t is the interpolation fraction ($t \in [0, 1]$), $\cos(\theta)$ is the dot product of q_1 and q_2, $\sin(\theta)$ is the sine of the angle between q_1 and q_2, $q_1 \cos(\theta)$ represents the component of q_1 parallel to q_2, $q_2 - q_1 \cos(\theta)$ represents the component of q_2 perpendicular to q_1, $\frac{q_2 - q_1 \cos(\theta)}{\sin(\theta)}$ is the normalized perpendicular component, and $\sin(\theta t)$ represents the rotation around the axis defined by the normalized perpendicular component.

5.5.6 LERP

While linear interpolation (15) is used for translation and scale.

$$\text{lerp}(a, b, t) = a + t \cdot (b - a) \tag{15}$$

Then, the algorithm proceeds to update the transformation matrix for each bone by incorporating the interpolated values. The resulting transformation matrices (16)(17) are stored in the self.MM array

Matrix transformation:

$$\text{transformed\_matrix} = \text{scale\_matrix} \times \text{rotation\_matrix} \tag{16}$$

where the rotation matrix comes from the interpolated quaternion.

$$\text{transformed\_matrix}[:3, 3] = \text{interpolated\_translation} \tag{17}$$

```
    def animation_for_loop(self, k_1, k_2, t0, t1):
        self.alpha = (self.time_add - t0) / abs(t1 - t0)

        keyframe1 = Keyframe(array_MM=[k_1])
        keyframe2 = Keyframe(array_MM=[k_2])

        # print(keyframe1.array_MM[0][1])
        for i in range(len(k_1)):

            translation_1 = translation(keyframe1.array_MM[0][i])
            translation_2 = translation(keyframe2.array_MM[0][i])

            rotation_1 = rotationEulerAngles(keyframe1.array_MM[0][i])
            rotation_2 = rotationEulerAngles(keyframe2.array_MM[0][i])

            rq1 = quat.Quaternion.
                  euler_to_quaternion(math.radians(rotation_1[0]),
                                      math.radians(rotation_1[1]),
                                      math.radians(rotation_1[2]))

            rq2 = quat.Quaternion.
                  euler_to_quaternion(math.radians(rotation_2[0]),
                                      math.radians(rotation_2[1]),
                                      math.radians(rotation_2[2]))

            rq1 = rq1 / rq1.norm()
            rq2 = rq2 / rq2.norm()

            scale_1 = scale(keyframe1.array_MM[0][i])
            scale_2 = scale(keyframe2.array_MM[0][i])


            if(self.inter == "LERP"):
                rl = quat.quaternion_lerp(rq1, rq2, self.alpha)
            else:
                rl = quat.quaternion_slerp(rq1, rq2, self.alpha)

            sl = self.lerp(scale_1, scale_2, self.alpha)
            sc = util.scale(sl[0],sl[1],sl[2])
            self.MM[i][:3, :3] = sc[:3, :3] @ quat.Quaternion.to_rotation_matrix(rl)

            self.MM[i][:3, 3] = self.lerp(translation_1, translation_2, self.alpha)
```

Listing 11: The function performs interpolation between two keyframes for each bone in the animation

To execute the animation_loop function, we need to invoke it through a system; otherwise, we would violate the principles of ECS.

In the following code snippet [12], we see the implementation of the system.

```
    class SkinnedAnimationSystem(System):
    def __init__(self, name=None, type=None, id=None):
        super().__init__(name, type, id)

    def apply2AnimationComponents(self, animationComponents: Elements.
                                  features.
                                  SkinnedAnimation.
                                  SkinnedAnimation.AnimationComponents):

        animation_data = animationComponents.animation_loop()

        return animation_data
```

Listing 12: Class for the animation system.

Below is the code[13] for the GUI that will appear in our program.

The variables WW and WW_9 were initialized in the code [4] on lines 3 and 5, essentially used to store the changes we make to translation, rotation, and scale through the GUI.

On line 6, a slider button labeled ”Alpha Tempo” is created in the GUI. The value of self.tempo is updated based on the user interaction, controlling the animation speed.

On line 7, there is a checkbox labeled ”Animation” that allows us to pause and resume the animation.

The purpose of the loops is to create collapsible tree nodes for each keyframe and articulation.

For each articulation, slider buttons are created for properties such as translation, rotation, and scale. The values are updated based on user interaction.

On line 30, if any articulation is modified, the code updates the WW variable, then calls read_tree to make the necessary changes not only to the articulation we’ve modified but also to its children from that articulation.

```
def drawSelfGui(self, imgui):
        global WW
        global WW_9

        imgui.begin("Animation", True)
        _, self.tempo = imgui.drag_float("Alpha Tempo", self.tempo, 0.01, 0, 5)

        _, self.anition_start = imgui.checkbox("Animation", self.anition_start)

        i = 0
        for k in self.keyframe:
            if imgui.tree_node("Keyframe " + str(i)):
                j = 0
                for mm in k:
                    if imgui.tree_node("Joint " + str(j)):

                        imgui.text("My Value: {}".format(mm))

                        tran_x, WW_9[i][j][0] = imgui.drag_float("Translate X",
                                                                 WW_9[i][j][0],
                                                                 0.1,
                                                                 -10,
                                                                 10)
```

```
                        tran_y, WW_9[i][j][1] = imgui.drag_float("Translate Y",
                                                                 WW_9[i][j][1],
                                                                 0.1,
                                                                 -10,
                                                                 10)
                        tran_z, WW_9[i][j][2] = imgui.drag_float("Translate Z",
                                                                 WW_9[i][j][2],
                                                                 0.1,
                                                                 -10,
                                                                 10)

                        rot_x,  WW_9[i][j][3] = imgui.drag_float("Rotate X",
                                                                 WW_9[i][j][3],
                                                                 0.1,
                                                                 -3,
                                                                 3)
                        rot_y,  WW_9[i][j][4] = imgui.drag_float("Rotate Y",
                                                                 WW_9[i][j][4],
                                                                 0.1,
                                                                 -3,
                                                                 3)
                        rot_z,  WW_9[i][j][5] = imgui.drag_float("Rotate Z",
                                                                 WW_9[i][j][5],
                                                                 0.1,
                                                                 -3,
                                                                 3)

                        sc_x,  WW_9[i][j][6] = imgui.drag_float("Scale X",
                                                                WW_9[i][j][6],
                                                                0.01,
                                                                0.1,
                                                                1)
                        sc_y,  WW_9[i][j][7] = imgui.drag_float("Scale Y",
                                                                WW_9[i][j][7],
                                                                0.01,
                                                                0.1,
                                                                1)
                        sc_z,  WW_9[i][j][8] = imgui.drag_float("Scale Z",
                                                                WW_9[i][j][8],
                                                                0.01,
                                                                0.1,
                                                                1)

                        if tran_x or tran_y or tran_z
                           or rot_x or rot_y or rot_z
                           or sc_x or sc_y or sc_z:
                            temp = util.scale(WW_9[i][j][6],
                                              WW_9[i][j][7],
                                              WW_9[i][j][8])
                                              @ eulerAnglesToRotationMatrix4(
                                                WW_9[i][j][3],
                                                WW_9[i][j][4],
                                                WW_9[i][j][5])
                            temp[0][3] = WW_9[i][j][0]
                            temp[1][3] = WW_9[i][j][1]
                            temp[2][3] = WW_9[i][j][2]

                            WW[i][j] = temp
                            self.keyframe[i] = read_tree(figure,3,WW[i],True)

                        imgui.tree_pop()
                    j += 1
                imgui.tree_pop()
            i += 1
        imgui.end()
```

Listing 13: Function for displaying GUI, with this GUI we adjust the rotation, translation, and scale of our model for each keyframe and each bone

When running the above code, the result will look like the image shown in figure[12].

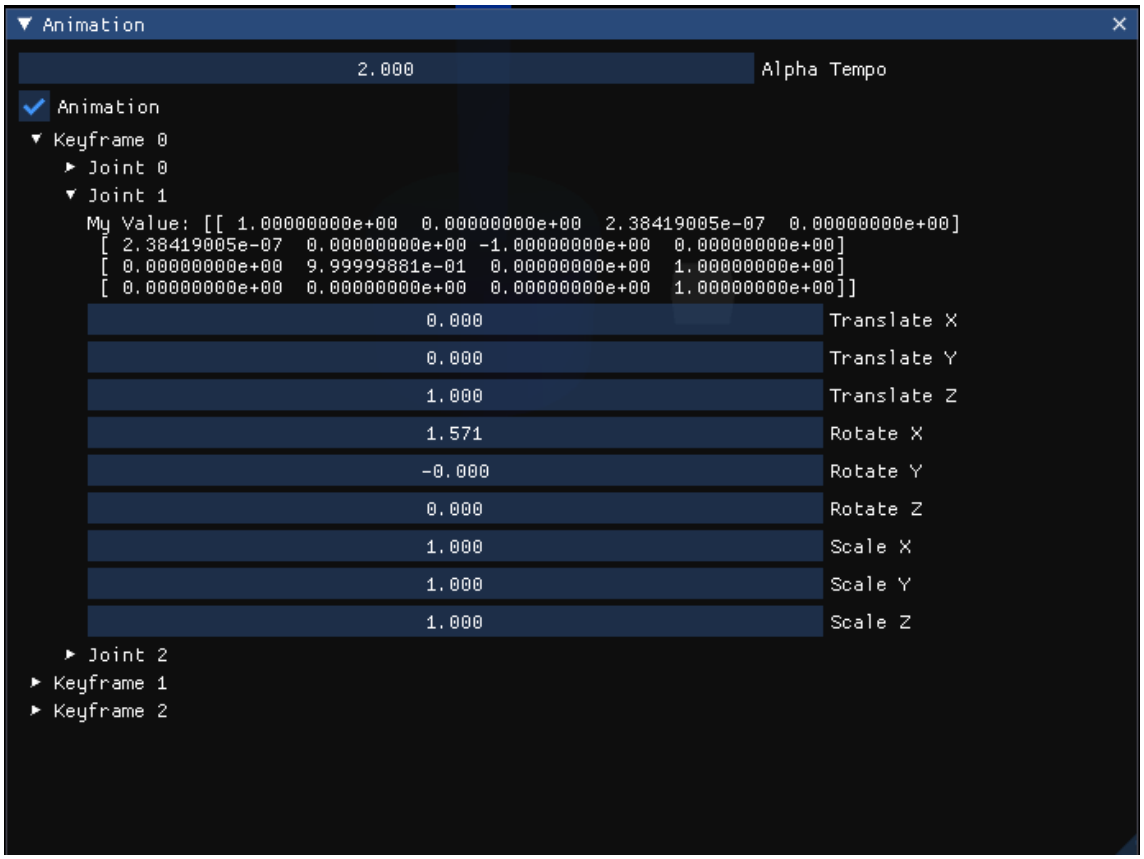


Figure 12: The GUI when the program is executed.

5.6 GPU Shaders

In the following shader, we have vPosition, vColor, vNormal, vWeight, and vWID, which are the corresponding variables from the code[21],Then, we define the constants MAX_BONES and MAX_BONES_INF. BB, MM, modelViewProj, and model are the variables from the code [22]. pos, color, and normal are the results produced by the shader to pass to the Phong algorithm.

For the main part of the code, we start by initializing the vector newv with zero values. Then, we have the loop that iterates with a maximum of MAX_BONES_INF(4) bones. Essentially, this means that a movement made to one bone can influence up to 4 parent bones. The if statement checks if vWID for the current bone is greater than or equal to zero. If true, the code inside the block is executed.

The mat is the transformation matrix for the current bone, which is obtained by multiplying the bone's matrix BB and the MM matrix.

The temp is the result of multiplying the initial vertex position vPosition by the mat.

The newv is updated by adding the weighted contribution of the transformed position, where the weight is determined by vWeight.

The Normal is calculated by multiplying the normal vector by the inverse transpose of the total model transformation matrix and the mat matrix.

In pos, the transformed position is assigned by applying the model matrix to the transformed new position of the vertex.

The color is defined as the original color vColor of the vertex.

It should be noted that to apply texture, we follow the same steps, just replacing vColor with vTexCoord and color with fragmentTexCoord.

```
    VERT_ANIMATION = """
        #version 410

        layout (location=0) in vec4 vPosition;
        layout (location=1) in vec4 vColor;
        layout (location=2) in vec4 vNormal;
        layout (location=3) in vec4 vWeight;
        layout (location=4) in vec4 vWID;

        const int MAX_BONES = 100;
        const int MAX_BONES_INF = 4;

        uniform mat4 BB[MAX_BONES];
        uniform mat4 MM[MAX_BONES];

        uniform mat4 modelViewProj;
        uniform mat4 model;

        out     vec4 pos;
        out     vec4 color;
        out     vec3 normal;

        void main()
        {
            vec4 newv = vec4(0.0f);

            for (int i = 0; i < MAX_BONES_INF; i++)
            {
                if(int(vWID[i]) >= 0)
                {
                    mat4 mat = BB[int(vWID[i])] * MM[int(vWID[i])] ;
                    vec4 temp = vPosition * mat;
                    newv += vWeight[i] * temp;
                    normal = mat3(transpose(inverse(model)))*mat3(mat) * vNormal.xyz;
                }
            }

            gl_Position = modelViewProj * newv;
            pos = model * newv;
            color = vColor;
        }
    """
```

Listing 14: Code for the Shader.

5.7 Main program implementation

For the implementation of the program, we have a main function where we call the necessary functions from the elements and the ones we have created.

Starting with the analysis of the main function, we have the variables Mshininess and Mcolor with which we adjust the color and shininess of the material from our model, where Mshininess takes values between 0 and 1, just like Mcolor.

```
    #Material
    Mshininess = 0.4
    Mcolor = util.vec(0.8, 0.0, 0.8)
```

Listing 15: Variables for the color and shininess of the model's material.

Next, we initialize the systems of the elements and the scene graph.

```
    # Initialize Systems used for this script
    transUpdate = scene.world.createSystem(TransformSystem("transUpdate",
                                                           "TransformSystem",
                                                           "001"))
    camUpdate = scene.world.createSystem(CameraSystem("camUpdate",
                                                      "CameraUpdate",
                                                      "200"))
    renderUpdate = scene.world.createSystem(RenderGLShaderSystem())
    initUpdate = scene.world.createSystem(InitGLShaderSystem())

    # Scenegraph with Entities, Components
    rootEntity = scene.world.createEntity(Entity(name = "Root"))
```

Listing 16: Initialization of the systems from the elements and the scene graph.

Next, we initialize the position of the camera in our scene.

```
    # Spawn Camera
    mainCamera = SimpleCamera("Simple Camera");
    # Camera Settings
    mainCamera.trans2.trs = util.translate(0, 0, 8) # VIEW
    mainCamera.trans1.trs = util.rotate((1, 0, 0), -45);
```

Listing 17: Initialization of camera parameters.

Below, we create the environmental and point light, set an initial intensity for the lighting, and adjust the light's position in space using util.translate and its size using util.scale.

```
    # Spawn Light
    ambientLight = Light("Ambient Light");
    ambientLight.intensity = 0.1;
    scene.world.addEntityChild(rootEntity, ambientLight);
    pointLight = PointLight();
    pointLight.trans.trs = util.translate(0.8, 1, 1) @ util.scale(0.2)
    scene.world.addEntityChild(rootEntity, pointLight);
```

Listing 18: Initialization of lighting and ambient light.

In the following code, we start by creating the entity for our model. Then, we add it to the scene graph. After that, we add components for transformation, keyframes, and animation.

```

    node4 = scene.world.createEntity(Entity(name="Object"))
    scene.world.addEntityChild(rootEntity, node4)
    trans4 = scene.world.addComponent(node4,
                                      BasicTransform(name="Object_TRS",
                                                     trs=util.scale(0.5)
                                                     @util.rotate((1,0,0),-90)
                                                     @util.translate(0,0,0)))
    mesh4 = scene.world.addComponent(node4, RenderMesh(name="Object_mesh"))
    key1 = scene.world.addComponent(node4, Keyframe(name="Object_key_1"))
    key2 = scene.world.addComponent(node4, Keyframe(name="Object_key_2"))
    key3 = scene.world.addComponent(node4, Keyframe(name="Object_key_3"))

    ac = scene.world.addComponent(node4,
                                  AnimationComponents(name="Animation_Components"))
```

Listing 19: Initialization of the model.

We declare the address of our model in the variable and then pass it to animation_initialize along with keyframe and animation component. The animation_initialize is the code we analyzed in the Model Data Extraction chapter [2][3][4]. We initialize the system for the animation, then place the keyframes

in the animation component so that the appropriate calculations can be made in the animation_loop. Finally, with generateNormals, we produce the necessary normals for our model.

```
    obj_to_import = MODEL_DIR / "astroBoy_walk.dae"
    vertices, colors, boneWeight, boneID, faces = animation_initialize(obj_to_import,
                                                                       ac,
                                                                       key1,
                                                                       key2,
                                                                       key3)

    testAnim = SkinnedAnimationSystem()

    ac.keyframe = [key1.array_MM[0], key2.array_MM[0], key3.array_MM[0]]

    normals = generateNormals(vertices, faces)
```

Listing 20: Model address on the computer model import function initialization of the animation system and creation of normal vectors

We import the vertices, colors, normals, boneWeight, and boneID obtained from the code above[20] into the shaders. Similarly, when we want to use textures, we import the texture coordinates we want instead of colors.

```
    #Passing vertices, colors, normals, bone weights, bone ids to the Shader
    mesh4.vertex_attributes.append(vertices)
    #mesh4.vertex_attributes.append(TEX_COORDINATES*int(len(i)/6))colors
    mesh4.vertex_attributes.append(colors)
    mesh4.vertex_attributes.append(normals)
    mesh4.vertex_attributes.append(boneWeight)
    mesh4.vertex_attributes.append(boneID)
    mesh4.vertex_index.append(faces)
    vArray4 = scene.world.addComponent(node4, VertexArray())
    #shaderDec4 = scene.world.addComponent(node4, ShaderGLDecorator(Shader(vertex_source
     = Shader.ANIMATION_SIMPLE_TEXTURE_PHONG_VERT, fragment_source=Shader.
    SIMPLE_TEXTURE_PHONG_FRAG)))
    shaderDec4 = scene.world.addComponent(node4,
                                          ShaderGLDecorator(
                                          Shader(vertex_source = Shader.VERT_ANIMATION,
                                          fragment_source=Shader.FRAG_PHONG)))
```

Listing 21: Code to pass basic variables to the shader.

Finally, we have the code with which we import the variables ac.bones[0], MM, trans4.l2cam, trans4.l2world as uniform variables into the shader[14]. Since ac.bones[0] does not change throughout the execution of the code, we leave it outside the loop. The other uniform variables are for Phong lighting. We call the system which gives us the result MM.

```
    shaderDec4.setUniformVariable(key='BB', value=ac.bones[0], arraymat4=True)
    shaderDec4.setUniformVariable(key='ambientColor',
                                  value=ambientLight.color,
                                  float3=True);
    shaderDec4.setUniformVariable(key='ambientStr',
                                  value=ambientLight.intensity,
                                  float1=True);

    shaderDec4.setUniformVariable(key='lightColor',
                                  value=np.array(pointLight.color),
                                  float3=True);
    shaderDec4.setUniformVariable(key='lightIntensity',
                                  value=pointLight.intensity,
                                  float1=True);

    shaderDec4.setUniformVariable(key='shininess',value=Mshininess,float1=True)
    shaderDec4.setUniformVariable(key='matColor',value=Mcolor,float3=True)
```

Listing 22: Final loop for rendering the scene.

```
    while running:

        scene.world.traverse_visit(transUpdate, scene.world.root)
        scene.world.traverse_visit_pre_camera(camUpdate, mainCamera.camera)
        scene.world.traverse_visit(camUpdate, scene.world.root)
        viewPos = mainCamera.trans2.l2world[:3, 3].tolist();
        lightPos = pointLight.trans.l2world[:3, 3].tolist();
        pointLight.shaderDec.setUniformVariable(key='modelViewProj',
                                                value=pointLight.trans.l2cam,
                                                mat4=True)

        MM = testAnim.apply2AnimationComponents(ac)

        shaderDec4.setUniformVariable(key='modelViewProj',
                                      value=trans4.l2cam,
                                      mat4=True);
        shaderDec4.setUniformVariable(key='model',value=trans4.l2world,mat4=True)

        shaderDec4.setUniformVariable(key='MM', value=MM, arraymat4=True)

        shaderDec4.setUniformVariable(key='viewPos', value=viewPos, float3=True);
        shaderDec4.setUniformVariable(key='lightPos', value=lightPos, float3=True);

        # call SDLWindow/ImGUI display() and ImGUI event input process
        running = scene.render()

        # call the GL State render System
        scene.world.traverse_visit(renderUpdate, scene.world.root)

        # ImGUI post-display calls and SDLWindow swap
        scene.render_post()

    scene.shutdown()
```

Listing 23: Final loop for rendering the scene.

Note that to add texture to our model, we need to add the following code at the beginning of the code[22]

```
    texturePath = TEXTURE_DIR / "dark_wood_texture.jpg"
    texture = Texture(texturePath)
    shaderDec4.setUniformVariable(key='ImageTexture', value=texture, texture=True)
```

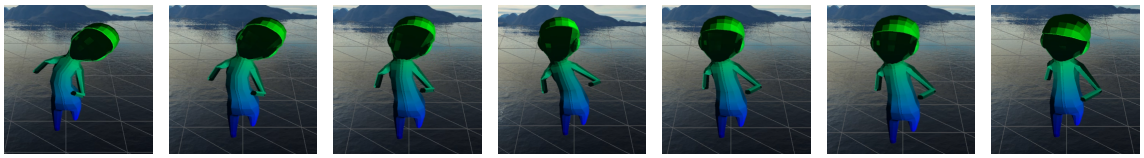
Listing 24: Adding texture.

5.8 Results

This chapter presents the results produced by the animation project, showcasing the performance and visual outcomes achieved through the implemented system. The focus was on the animation of two models: the Astroboy and robotic arm model. The figures [13][14] illustrate keyframes and interpolated frames from these animations. Although, despite the inherent complexity of working with rigged models—due to the need for incorporating skeletal structures—the use of GPU acceleration significantly improved performance. Specifically, the use of a for loop [14] for processing each index in conjunction with the GPU allowed for increased control over the models and enhanced overall system performance.

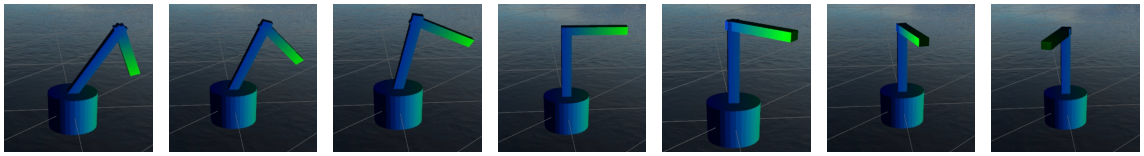
The system's efficiency was evident through the frame rate achieved. On a Nvidia GTX 1050ti GPU, the system maintained an average frame rate of 74–75 fps, matching the maximum refresh rate of the monitor used for testing[15]. This result is particularly noteworthy given that Python, a relatively slower programming language, was used in the development. This consistent performance demon-

strates the effectiveness of the GPU-enabled architecture in handling the computational demands of real-time animation.



(a) (b) (c) (d) (e) (f) (g)

Figure 13: Astroboy animation keyframes (a)(d)(g) and in between frames (b)(c)(e)(f).



(a) (b) (c) (d) (e) (f) (g)

Figure 14: Robot arm animation keyframes (a)(d)(g) and in between frames (b)(c)(e)(f).

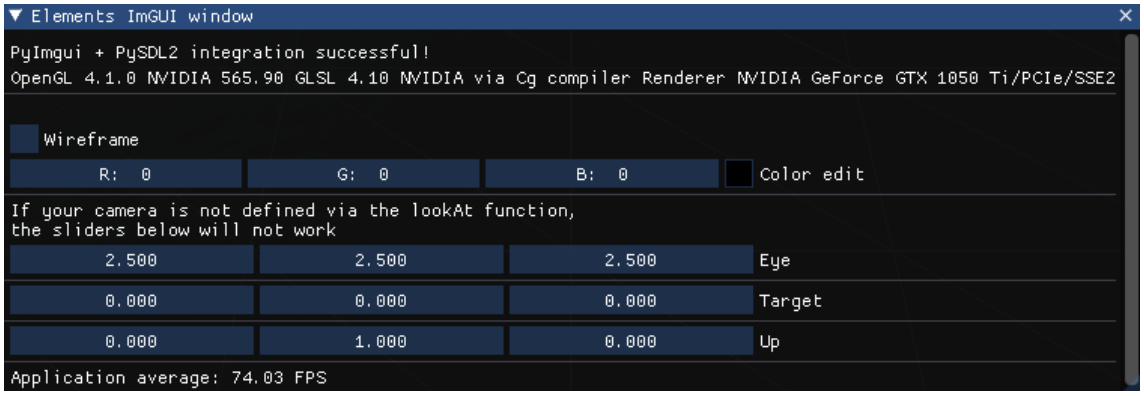
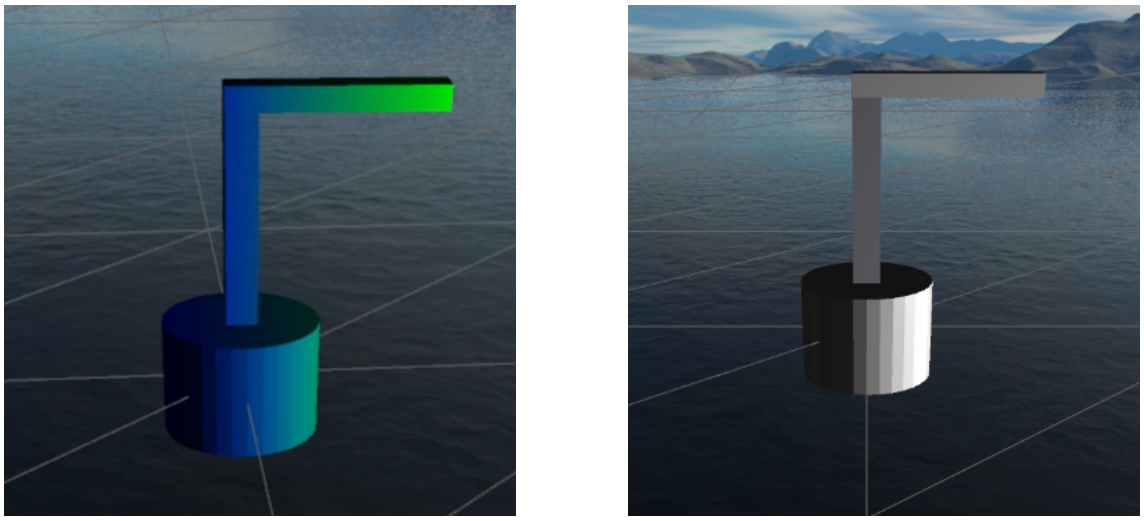


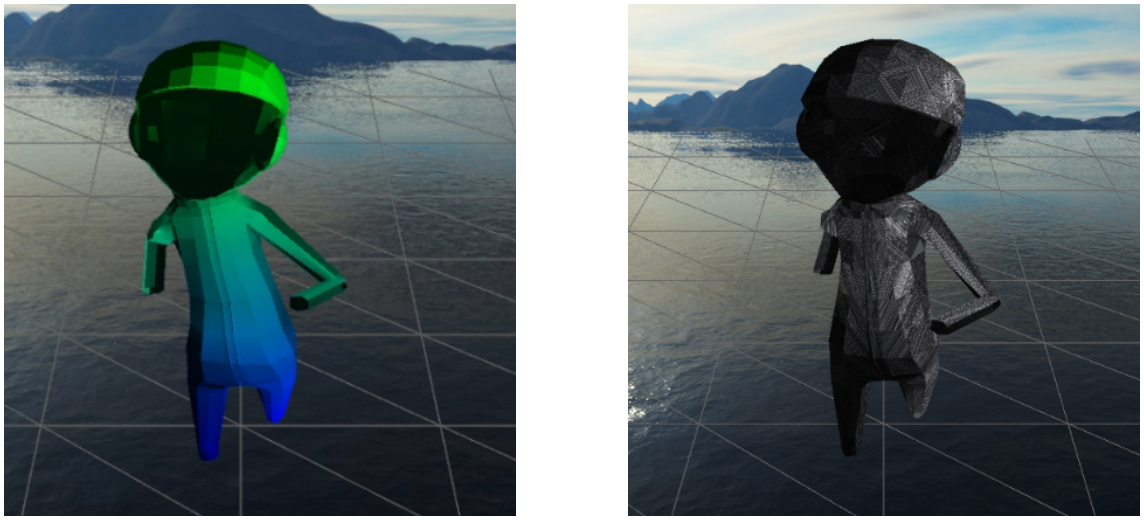
Figure 15: Framerate and GPU used of the project.

In addition to performance optimization, efforts were made to improve the visual fidelity of the models by incorporating textures. For the Astroboy model, the texture was successfully applied, resulting in a more realistic metal appearance due to the appropriate distribution of polygons across the model [18]. However, for the robotic arm, the applied texture did not have as pronounced an effect. This discrepancy is attributed to the stretched nature of the polygons in the model, which caused the texture to lose its distinctive characteristics [16].



(a) (b)

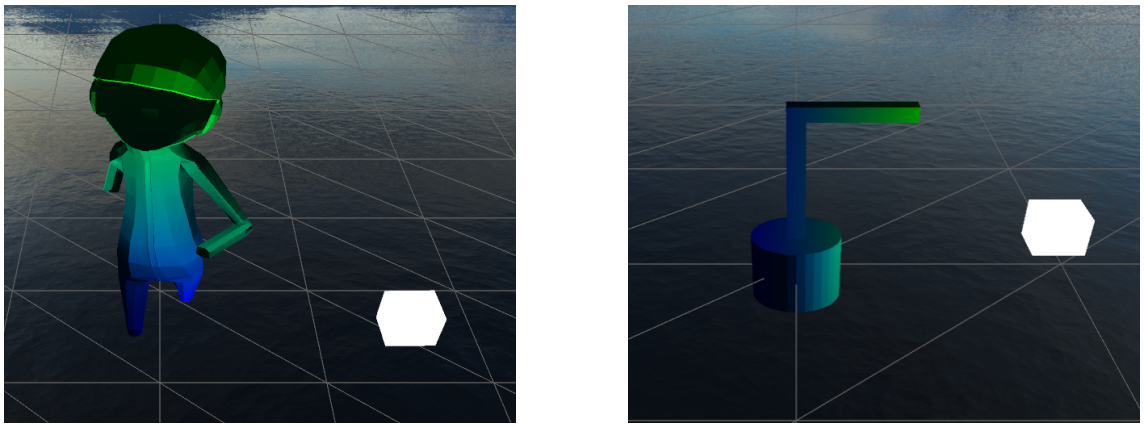
Figure 16: (a) gradient color, (b) paper texture.



(a) (b)

Figure 17: (a) gradient color, (b) metal texture.

Lighting effects were also tested, with a light source placed to the right of the models. As shown in figure [18], the lighting behaved as expected, illuminating the objects appropriately based on the position of the light source. These results validate the integration of Phong lighting within the system and demonstrate the overall effectiveness of the lighting model used in this project.



(a) (b)

Figure 18: (a) Astroboy lighting, (b) Robot arm lighting.

The results of this project demonstrate that the integration of GPU acceleration in an Entity-Component-System (ECS) framework significantly improves the performance and flexibility of real-time 3D animation. The system achieved high frame rates, even with the added complexity of rigged models and realistic lighting. While some limitations were observed with texture mapping on certain models, the overall visual outcomes and system performance underscore the advantages of using GPU-accelerated ECS frameworks for animation. These results set a foundation for future work, which could involve further optimization of the model importation process and the development of more interactive elements within the graphical user interface. Future enhancements may also explore the application of this framework in more complex simulations and virtual reality environments.

6 Summary and Future Extensions

In this work, the animation of the AstroBoy model and a robotic arm model was implemented, along with the application of Phong lighting in combination with texture. To achieve the animations, three keyframes were used. The majority of the code runs on the CPU, except for the shaders and lighting, which run on the GPU and constitute the heaviest workload.

Looking ahead, there are several opportunities to extend this research. One promising direction involves refining the model import process. While the current implementation manages to import models such as the AstroBoy figure and a robotic arm, more sophisticated mechanisms could be developed to automate and streamline the process. By improving the adaptability of the system to different file formats and structures, the system could become even more versatile and user-friendly.

Another area worth exploring is the enhancement of the graphical user interface (GUI). The current version allows for animation through fixed keyframes, but future versions could benefit from interactive elements that enable users to dynamically manipulate keyframes, adjust textures, or even replace models on the fly. These enhancements would make the system more intuitive, offering a more interactive and flexible environment for real-time animation and 3D rendering.

Additionally, there is significant potential to further optimize the system for more computationally demanding tasks. Expanding the ECS framework to handle more complex physical simulations, such as soft-body dynamics or fluid simulations, would push the boundaries of what is possible in real-time graphics. This could involve integrating machine learning algorithms to predict object behavior or further developing GPU-accelerated techniques to maintain performance even as scene complexity grows.

Moreover, this research opens up exciting possibilities for integration into virtual and augmented reality (VR/AR) environments. By adapting the ECS framework to these immersive platforms, the system could be used in interactive simulations, gaming, and educational applications, offering users highly responsive and realistic experiences. This would require refining object interaction capabilities and ensuring that the system can handle the added complexity of real-time user input and feedback.

From an educational standpoint, the project also lays the groundwork for further development within the Elements toolkit. As the system continues to evolve, it could serve as an invaluable teaching tool, helping students to understand both GPU programming and real-time graphics. Expanding the toolkit with additional learning modules and practical exercises would bridge the gap between theoretical knowledge and hands-on experience, empowering students to experiment with advanced graphics concepts in a structured and accessible way.

In conclusion, while the current project has successfully demonstrated the power and flexibility of GPU-enabled ECS frameworks for real-time animation, it also highlights several avenues for future research and development. By improving model import processes, enhancing the GUI, optimizing for more complex simu-

lations, and exploring VR/AR applications, this system has the potential to push the boundaries of 3D animation and real-time graphics even further.

References

Academy, C. (2021), 'Cave academy'.
URL: https://caveacademy.com/wiki/post-production-shots/animation/animation-training/introduction-to-animation-course/animate-an-action-move-the-tornado-kick/

Andújar Gran, C. A., Chica Calaf, A., Fairén González, M. & Vinacua Pla, Á. (2018), Gl-socket: A cg plugin-based framework for teaching and assessment, in 'EG 2018: education papers', European Association for Computer Graphics (Eurographics), pp. 25–32.

Bundiuk, V. (2023), 'What is rigging in 3d animation and what does it mean?'.
URL: https://aaagameartstudio.com/blog/what-is-rigging-in-3d-animation/

docs.unity3d.com (2024), 'Ecs concepts | entities | 0.17.0-preview.42', `https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/ecs_core.html`. Accessed: 24 September 2024.

Gamma, E. (1995), 'Design patterns: elements of reusable object-oriented software'.

Härkönen, T. (2019), 'Advantages and implementation of entity-component-systems'.

Insider, G. D. (2019), 'Rigging and skinning'.
URL: https://gamedevinsider.com/making-games/rigging-and-skinning/

Lin, W. (2020), 'Entity component system for unity: Getting started', `https://www.kodeco.com/7630142-entity-component-system-for-unity-getting-started`. Accessed: 29 September 2024.

Miller, J. R. (2014), Using a software framework to enhance online teaching of shader-based opengl, in 'Proceedings of the 45th ACM technical symposium on Computer science education', pp. 603–608.

Papagiannakis, G., Kamarianakis, M., Protopsaltis, A., Angelis, D. & Zikas, P. (2023), Project elements: A computational entity-component-system in a scene-graph pythonic framework, for a neural, geometric computer graphics curriculum, in A. Magana & J. Zara, eds, 'Eurographics 2023 - Education Papers', The Eurographics Association.

Papagiannakis, G., Papanikolaou, P., Greassidou, E. & Trahanias, P. E. (2014), glga: an opengl geometric application framework for a modern, shader-based computer graphics curriculum., in 'Eurographics (Education Papers)', pp. 9–16.

Petropoulos, J., Kamarianakis, M., Protopsaltis, A. & Papagiannakis, G. (2024), 'pygandalf–an open-source, geometric, animation, directed, algorithmic, learning framework for computer graphics', arXiv preprint arXiv:2409.16724 .

Phong, B. T. (1998), Illumination for computer generated pictures, in 'Seminal graphics: pioneering efforts that shaped the field', pp. 95–101.

Product, D. (1993), 'Phong shading reformulation for hardware renderer simplification'.

Theoharis, T., Papaioannou, G., Platis, N. & Patrikalakis, N. M. (2008), Graphics and visualization: principles & algorithms, CrC Press.

Toisoul, A., Rueckert, D. & Kainz, B. (2017), Accessible glsl shader programming., in 'Eurographics (Education Papers)', pp. 35–42.

Vitsas, N., Gkaravelis, A., Vasilakis, A.-A., Vardis, K. & Papaioannou, G. (2020), Rayground: An online educational tool for ray tracing., in 'Eurographics (Education Papers)', pp. 1–8.

Webster, C. (n.d.), 'Entity component systems usefulness'.

Wikipedia (2024a), 'Phong reflection model — wikipedia, the free encyclopedia', `http://en.wikipedia.org/w/index.php?title=Phong%20reflection%20model&oldid=1133079828`. [Online; accessed 04-February-2024].

Wikipedia (2024b), 'Texture mapping'.
URL: https://en.wikipedia.org/wiki/Texture$_{m}apping$

Wünsche, B. C., Huang, E., Shaw, L., Suselo, T., Leung, K.-C., Dimalen, D., van der Mark, W., Luxton-Reilly, A. & Lobb, R. (2019), Coderunnergl-an interactive web-based tool for computer graphics teaching and assessment, in '2019 International Conference on Electronics, Information, and Communication (ICEIC)', IEEE, pp. 1–7.

Abbreviations - Acronyms

ECS	Entity Component System
GLSL	OpenGL Shading Language
OpenGl	Open Graphics Library
CPU	Central Processing Unit
GPU	Graphics Processing Unit
3D	Three Dimensional