Dissertation/Report submitted to the

UNIVERSITY OF MOHAMED BOUDIAF - M'SILA, ALGERIA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

in partial fulfillment of the requirements for the degree of

Master's in Computer Science

Presented by

Wissal Titraoui

Entitled

An Interactive Computer Science Teaching Platform Using ManimGL

Under the supervision of

Abdessattar Ghemougui

Jury Members

Dr. Abdallah Tharafi	University of M'sila	President
Dr. Abdessattar Ghemougui	University of M'sila	Supervisor
Dr. Mohamed Chatra	University of M'sila	Examiner

June, 2026

Abstract

This thesis focuses on the design and development of an interactive learning framework that simplifies abstract and complex concepts in computer science and data structures. This work aims to move beyond static media and demonstrate the importance of visual simulation and immediate feedback. The proposed Framework is built using the ManimGL library and the Python programming language (NumPy, Pyglet, OpenGL), allowing learners to trace code execution and monitor memory changes in a synchronized manner. The project includes both a software engineering component and an evaluation component using a field survey to measure the proposed Framework's effectiveness in enhancing comprehension and fostering interaction.

Keywords: Interactive learning, ManimGL, computer science, visual simulation, data structures, OpenGL.

الملخص

تتمحور هذه المذكرة حول تصميم وتطوير إطار عمل تعليمي تفاعلي يعمل على تبسيط المفاهيم المجردة والمعقدة في علوم الحاسوب وهياكل البيانات، حيث جاء هذا العمل ليزيح الوسائط المجردة ويظهر أهمية المحاكاة البصرية والتحكم في الوقت الفعلي. تم بناء الإطار اعتمادا على مكتبة ManimGL ولغة Python (معتمد على OpenGL، Pyglet، NumPy) مما يسمح للمتعلم بتتبع التعليمات البرمجية داخل الكود وتغييرات الذاكرة في الزمن الحقيقي. يتضمن العمل جانبا هندسيا برمجيا وجانبا تقييميا من خلال الاستبيان الميداني لقياس مدى فاعلية الإطار في زيادة الاستيعاب وتعزيز التفاعل.

الكلمات المفتاحية: التعليم التفاعلي، ManimGL، علوم الحاسوب، محاكاة بصرية، هياكل البيانات، OpenGL.

إهداء

بعد مسيرة طويلة دامت سنوات حملت في طياتها الكثير من الصعوبات والمشقة والتعب، ها أنا أقف اليوم على عتبة تخرجي أقطف ثمار تعبي فخورة بنفسي، بهزائمي وانكساراتي، بخيباتي، بنوبات حزني واكتئابي التي لم أتقاسمها مع أحد، فخورة بالندوب التي لم يلحظها أحد، بتعثراتي وسقوطي، بتجاربي، بفشلي المتكرر، بمحاولاتي المتواصلة، بالأيام التي كانت قاسية علي ولكنني تخطيتها...

فخورة بكل المعارك التي خضتها، حتى تلك التي مزقت روحي إلى أشلاء، فخورة بصبري، بكل المشاعر التي أحملها داخلي، فخورة بابتسامتي التي كانت سلاحي الوحيد ضد كل المواقف.

فخورة بأحلامي، بأصغر انتصاراتي، وأبسط خطواتي بإصراري الذي يصل عنان السماء، فخورة بما وصلت إليه وما سأصل إليه. فخورة بكل تفاصيلي التي لا يعلم أحد عنها شيء. فاللهم لك الحمد قبل أن ترضى ولك الحمد إذا رضيت ولك الحمد بعد الرضا لأنك وفقتني على إتمام هذا العمل وتحقيق حلمي.

أهدي تخرجي هذا:

إلى الذي زين اسمي بأجمل الألقاب، من دعمني بلا حدود وأعطاني بلا مقابل.

إلى من علمني أن الدنيا كفاح سلاحها العلم والمعرفة، إلى من غرس في روحي مكارم الأخلاق داعمي الأول في مسيرتي وسندي وقوتي وملاذي بعد الله... إلى فخري واعتزازي **(والدي)**.

إلى نبع الحنان وقلب الدعاء الذي لم يغلق بابه يوماً، إلى من كانت عيونها تلاحق خطواتي خوفاً علي. يا أمان أيامي ويا هدايا الله في عمري. نجاحي هذا امتداد لدعائك، وهدية متواضعة لقلبك الكبير **(والدتي)**.

إلى شمعتا حياتي سندي الذي لا يميل، ورفاق عالمي الصغير، واليد التي تمتد لي قبل أن أطلب. لقد كنتم دائماً جزءاً من قوتي، وسبباً في أن أقف بثبات في أصعب الأيام **(أختاي)**.

إلى مؤنسي رحلتي من ساهموا في صنع ذكريات لا تنسى. وجودكم كان مساحة رحبة لقلبي وسنداً لتجاوز الصعوبات.

ها أنا اليوم أتممت أول ثمراته راجية من الله تعالى أن ينفعني بما علمني وأن يعلمني ما أجهل ويجعله حجة لي لا علي.

وصال تيطراوي

تشكرات

بسم الله الرحمن الرحيم

وَقُلِ اعْمَلُوا فَسَيَرَى اللَّهُ عَمَلَكُمْ وَرَسُولُهُ وَالْمُؤْمِنُونَ (التوبة: 105)

الحمد لله الذي بنعمته وتوفيقه تتم الصالحات، وبذكره تتنزل البركات، والحمد لله الذي أعانني ووفقني لإنجاز هذا العمل العلمي.

وامتداداً لقول رسول الله صلى الله عليه وسلم: **مَنْ صَنَعَ إِلَيْكُمْ مَعْرُوفًا فَكَافِئُوهُ**، وتطبيقاً لهذا الهدي النبوي الكريم، يسرني أن أتقدم بالشكر والتقدير إلى الأستاذ الدكتور **عبد الستار غموقي**، الذي اقترح موضوع هذا البحث وتولى مهمة الإشراف عليه، ومتابعته حتى تقديمه إلى لجنتكم الموقرة، فله مني وافر الاحترام والتقدير.

Contents

List of Figures

List of Tables

Introduction

The growing complexity of foundational computer science curricula particularly topics involving dynamic memory, recursive execution, and algorithmic state has intensified a well-documented problem in undergraduate instruction: the difficulty of making inherently non-visible computational processes observable to learners. While educational technology has expanded considerably in scope over the past two decades, most available tools address either visual quality or interactivity, but rarely both simultaneously.

The core problem addressed by this research lies in the inability of existing educational technologies to integrate these essential features. High-quality visual tools are often static and linear, while interactive systems lack graphical precision and mathematical accuracy. This limitation motivates the need for a unified interactive framework capable of combining these aspects effectively.

Based on this problem, this research aims to develop an interactive educational Framework built on the ManimGL engine, designed to allow instructors to create high-quality educational animations that can be modified dynamically during lectures.

The contribution of this work lies in proposing and implementing the Manim-CS Framework as an integrated environment that bridges the gap between visualization, interaction, and flexibility in educational content delivery.

To document the structure of this work, this thesis is organized into three main chapters: the first chapter presents the theoretical framework of educational methodologies and visual learning; the second chapter explores interactive graphics frameworks with particular emphasis on the Manim library; and the third chapter presents the design and implementation of the Manim-CS Framework as the proposed solution to the identified problem.

Theoretical Framework of Educational Methodologies and Visual Learning

1.1 Introduction

The field of education has witnessed a qualitative transformation driven by technological advancements, shifting the focus from traditional teacher-centered instruction to modern approaches that place the learner at the center of the educational process, fostering greater active participation. This chapter addresses the theoretical and pedagogical foundations of dynamic media, focusing on the role of animation in representing complex causal relationships. It also analyzes the technological gap between pre-recorded linear media, which impose a form of cognitive passivity, and interactive systems that allow for responsive control over variables.

1.2 Overview of Teaching Methodologies

Educational practices have undergone significant evolution over time, driven by technological advancements and changes in curriculum design. These practices have been divided into two main approaches: teacher-centered learning, in which the instructor serves as the primary source of knowledge, and learner-centered learning, in which the student becomes the foundation of the educational process through active engagement, with the instructor serving as a facilitator. This shift is fundamentally linked to the distinction between passive learning, which relies on the reception of information, and active learning, which requires the learner to process information critically and participate in constructing knowledge. Historically, the practical application of these methodologies has depended on educational tools that began with simple methods before technology entered the classroom.

1.3 Traditional Education and Teaching Aids

Traditional education refers to an educational model based on direct interaction between instructor and learner within a shared physical space. Historically, the educational process has relied on conventional tools such as textbooks and the blackboard, where knowledge is transmitted through lecturing and direct instruction. These methods grant instructors considerable flexibility to adapt their explanations based on student responses. While this direct interaction is effective, such tools have proven inadequate for representing complex or dynamic concepts, leading to an over-reliance on students' mental imagery and intuition.

Figure 1.1: Traditional blackboard and textbook-based educational learning.

1.3.1 Transition to Presentation Systems

To address the visual representation gap inherent in traditional education, slide-based presentation systems (such as PowerPoint [1] and Keynote [2]) emerged. These systems organize content into sequential units that follow a linear structure and a pre-programmed timeline, ensuring a standardized presentation of the material and facilitating digital archiving. The value of these tools lies in their capacity to integrate multimedia elements (images, videos, and animations), which aligns with the principles of visual learning by simplifying abstract concepts and reducing cognitive load through the distribution of information across auditory and visual channels.

However, complete reliance on these systems has introduced significant pedagogical and technical challenges. Technically, they are characterized by structural rigidity, making it difficult to modify content or alter the presentation path during instruction, thereby limiting the ability to respond to students' immediate questions. Pedagogically, these tools have often positioned learners as passive recipients with limited active participation. They also increase the risk of instructors becoming overly dependent on text-heavy slides and reading content verbatim, which may produce cognitive distraction caused by visual clutter or textual overload. This, in turn, burdens working memory and hinders the processing of essential information.

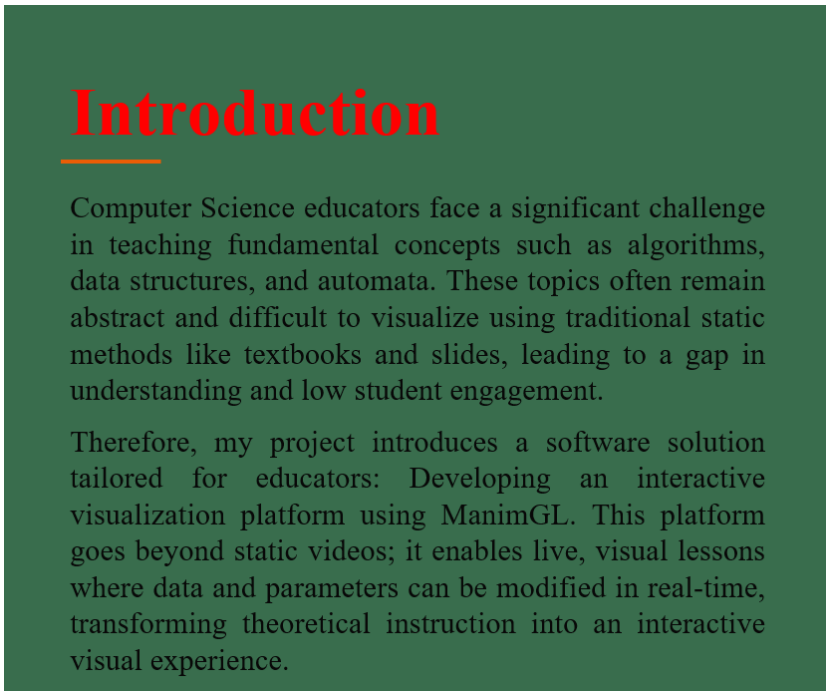

Figure 1.2: Cognitive overload in traditional presentation slides.

1.4 The Cognitive Foundations of Visual Learning

The effectiveness of visualization is rooted in cognitive principles that explain how the human brain processes information.

Allan Paivio [3] proposed the dual coding theory, positing that the brain operates through two systems: a verbal system (texts, words) and a visual system (images, diagrams). Learning through both words and images produces two distinct memory traces, resulting in more effective retention of information. This confirms that explanation supported by visual representation is more effective than reliance on text alone.

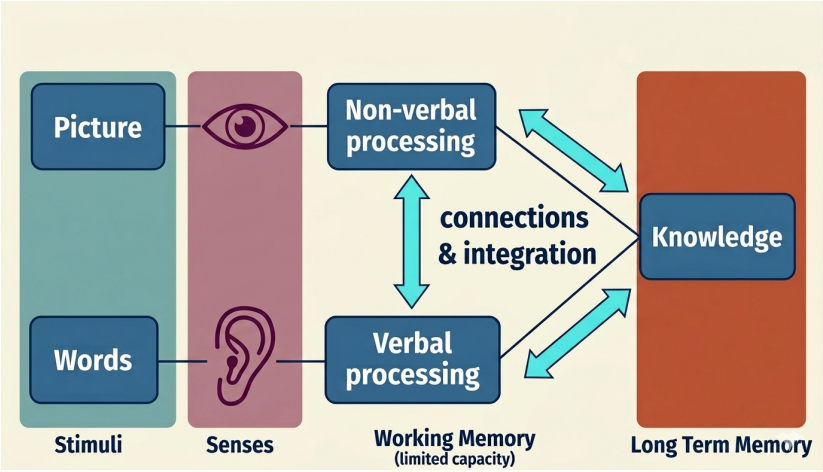

Figure 1.3: Allan Paivio's dual coding theory[3].

John Sweller [4], [5] developed cognitive load theory, distinguishing between three types of load: intrinsic load, which reflects the inherent difficulty of the subject matter; extraneous load, which relates to how information is presented; and germane load, which is responsible for building and automating schemas. The central argument is that poorly designed instruction causes the learner's cognitive resources to be spent on interpreting the presentation rather than on understanding the content itself. Accordingly, Sweller emphasizes that well-designed diagrams and charts reduce extraneous cognitive load, freeing the learner to focus on comprehension and knowledge construction.

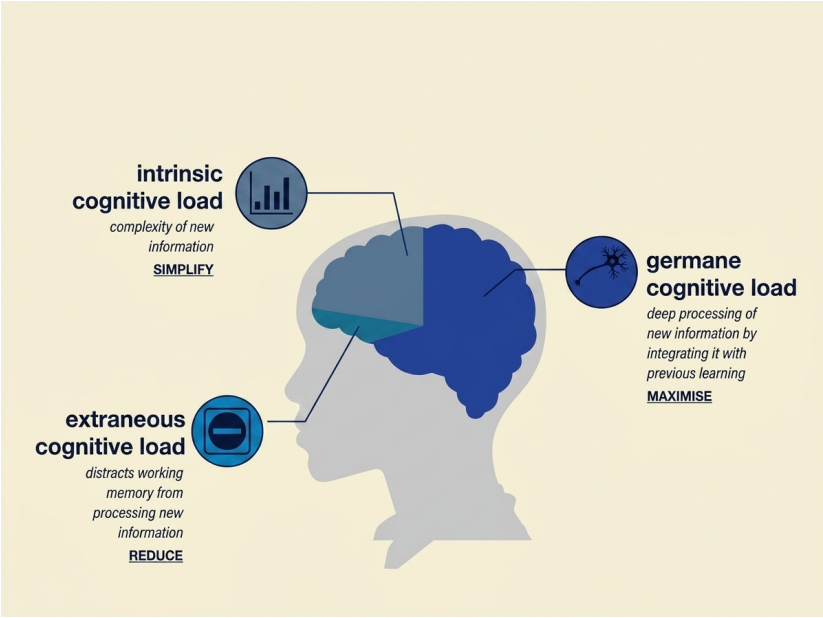

Figure 1.4: John Sweller's cognitive load theory[4], [5].

Richard Mayer's research [6] on the cognitive theory of multimedia learning similarly identifies two processing systems in the human brain: one verbal and one visual. The visual

system is capable of rapidly comprehending complex relationships and structures, owing to its capacity to process large amounts of information in parallel. Diagrams and charts are therefore an efficient means of explaining and simplifying complex concepts.

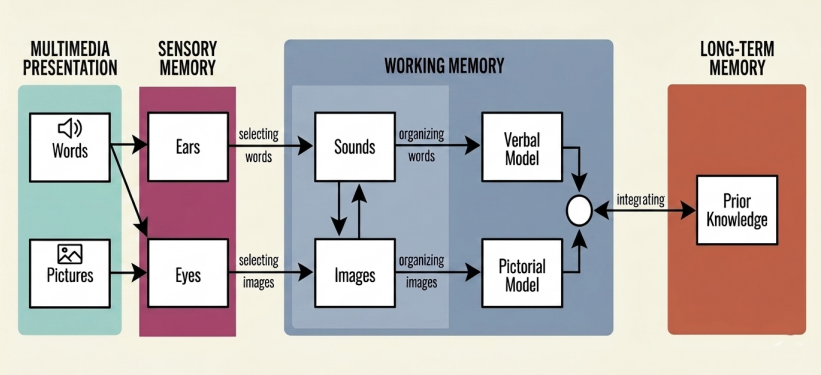

Figure 1.5: Cognitive theory of multimedia learning[6].

1.5 Dynamic Media

Animation is an effective educational tool owing to its ability to represent cause-and-effect relationships clearly and to demonstrate the step-by-step development of phenomena and processes. This helps students understand dynamic processes and complex systems, fostering deeper comprehension and active engagement. However, most animated media suffer from adaptive rigidity and cognitive inertia. Because they are pre-recorded, they cannot be modified during a lesson to explore different scenarios, leaving the student in a watching state rather than an experiencing one. This limitation necessitates a shift toward a higher level of interaction.

This deficiency also applies to educational videos, which present additional pedagogical challenges. Extended viewing periods often lead to distraction and decreased mental engagement, causing the learner to shift gradually from active attention to passive reception. The absence of any meaningful interaction with the system, and the inability of traditional videos to offer activities that allow for practice rather than observation, underscores the inherent passivity of these media. This highlights a pressing need to move beyond visual representation toward proactive interaction, where the system not only presents information but also enables the learner to become an active participant in constructing the learning experience.

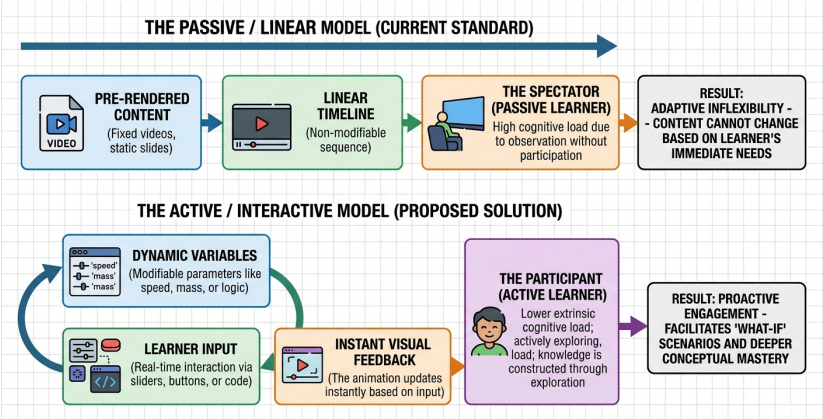

Figure 1.6: Information flow models.

1.6 Interaction Paradigms in Learning

Interaction models are central to the design of contemporary educational systems. They go beyond content delivery to actively promote learners' intellectual and emotional engagement. This is reflected in a hierarchy of interaction levels. The lowest is the passive level, characterized by a one-way flow of information. The intermediate level is interactive, where learners respond to prompts and activities. The highest is the proactive level, which allows learners to initiate exploration and make autonomous choices, leading to deeper understanding and knowledge construction [7].

The development of these interaction levels is grounded in well-established educational frameworks that draw on constructivist theory [8] and active learning [9]. Constructivism holds that knowledge is not merely transmitted but is independently developed through experience and engagement. As Michael Prince (2004) [10] demonstrated, active learning strategies improve students' academic performance, problem-solving abilities, and capacity for self-directed study. Interaction thus serves as the bridge between educational theory and practical application, transforming the learning process from passive reception to active construction.

With the rapid advancement of educational technology, these concepts have been realized in digital tools that support dynamic learning environments. Simulation platforms such as Physics Education Technology (PhET) [11] enable the immediate testing of variables. Interactive media tools such as ThingLink [12] combine digital content with interactive elements. Computational environments such as Desmos [13] and Jupyter [14] Notebook [15] provide learners with digital laboratories for hands-on practice in programming and mathematics. These developments support the contemporary view that the learner is both the starting point and the driving force of the educational process.

1.7 Gap Analysis

Despite the sophistication of current educational tools, a significant functional gap remains. Presentation systems are structured but rigid and linear; educational videos and animations are visually rich but promote passive reception and lack adaptive flexibility; while simulations offer interaction but often lack the structured narrative flow of a coherent lesson. As a result, most existing educational technologies address only certain aspects of the learning process while neglecting others, creating a separation between visual representation, interaction, and pedagogical flexibility. No existing unified system effectively combines high-fidelity animation, flexible presentation structures, and proactive interaction.

Learning Method	**Visual Quality**	**Interactivity**	**Structural Flexibility**
Presentation Systems	Moderate	Low	Low
Educational Videos	High	Very Low	Low
Interactive Simulations	Moderate to High	High	Limited
Traditional Teaching	Low	Moderate	Moderate
Proposed Interactive framework	High	High	High

Table 1.1: Comparison of existing educational approaches.

1.8 Conclusion

It follows from the preceding discussion that the shift toward interactive learning requires overcoming the limitations of traditional linear media by addressing the functional gap between structured presentation and responsive engagement. Building on these theoretical foundations, the next chapter moves to the technical aspects, presenting the selected software library and development tools used to implement the proposed Framework.

Introduction to Computer Graphics and the Manim Frameworks

2.1 Introduction

Computer graphics technologies have become an essential component of modern educational visualization systems, enabling the creation of dynamic and interactive learning environments. Among existing visualization frameworks, Manim has emerged as a capable tool for producing high-quality mathematical and scientific animations through programmable vector graphics. This chapter presents the technical foundations related to computer graphics rendering and visualization frameworks, with particular focus on the architecture and capabilities of the Manim framework. It also discusses the current limitations of Manim in the context of computer science education, which motivates the development of the proposed Manim-CS extension presented in the next chapter.

2.2 Fundamentals of Computer Graphics Rendering

This section introduces the fundamental principles of computer graphics rendering, focusing on the distinction between CPU-based and GPU-based approaches and their respective roles in modern visualization systems.

2.2.1 The Graphics Rendering Process

The graphics rendering process is the sequence of steps used to convert a digital scene into a final image displayed on screen. It forms the core pipeline of any computer graphics system.

The process begins with a virtual scene composed of geometric objects, transformations, and camera settings. These elements are then processed through a rendering pipeline that includes geometric processing, rasterization, and fragment processing.

Geometric processing applies transformations such as translation, rotation, and scaling to position objects in space. Rasterization converts these objects into screen-space pixels, while fragment processing applies color, lighting, and texture to produce the final image.

2.2.2 Central Processing Unit Rendering

Software-based rendering principles

CPU-based rendering refers to a software-based approach in which all graphical computations are handled by the CPU. It is commonly used for vector graphics rendering and prioritizes accuracy and image quality over speed.

Limitations in interactive applications

This approach is computationally expensive and considerably slower than GPU rendering, making it unsuitable for applications requiring interactive or high frame-rate output.

2.2.3 Graphics Processing Unit Rendering

GPU acceleration principles

GPU rendering relies on the graphics processing unit to perform parallel computations for image generation. It uses hardware acceleration to handle complex graphical tasks efficiently.

Advantages for interactive visualization

This method provides high-speed rendering and supports responsive interaction, making it well-suited for animations, simulations, and interactive visual systems.

2.3 Low-Level Graphics Libraries

This section presents the main low-level graphics libraries used for rendering, highlighting their architectural differences and their role in providing direct access to hardware-accelerated graphics processing.

2.3.1 OpenGL

OpenGL is a cross-framework graphics Application Programming Interface (API) used for rendering 2D and 3D vector graphics. It provides direct communication with the GPU, enabling high-performance graphical applications.

2.3.2 DirectX

DirectX[16] is a collection of APIs developed by Microsoft for handling multimedia and graphics tasks, particularly in game development and Windows-based applications.

2.3.3 Vulkan

Vulkan[17] is a modern low-level graphics Application Programming Interface (API) that provides efficient, high-performance access to GPU hardware with reduced CPU overhead and greater control over rendering pipelines.

2.4 High-Level Visualization Frameworks

2.4.1 Game Engines

A game engine is a software framework that supports the development of video games. It typically integrates components such as graphics rendering, physics simulation, and asset management.

2.5 The Manim Framework

Manim emerged as a specialized software tool for creating mathematical animations. Grant Sanderson developed it for producing educational videos for his YouTube channel 3Blue1Brown [21]. The primary goal of Manim was to provide a Python[18]-based environment capable of generating precise, high-quality animations with full programmatic control over visual elements, timing, and motion. As its use expanded beyond the original video productions, the need arose to restructure the project so that it could evolve from a personal tool into a well-managed framework suitable for academic use and community-driven development. This evolution led to the emergence of two main versions: ManimGL and Manim Community Edition [22].

2.5.1 Manim Design Goals

- **Precise Programmatic Control**: Manim enables developers and researchers to create animations using Python[18], ensuring high mathematical accuracy and reproducibility. Rather than relying on manual animation tools that are prone to imprecision, each visual element is generated based on mathematical equations and algorithms that guarantee correspondence between the visual scene and the scientific content.
- **Conceptual Visualization**: The library's principal goal is to foster a deeper understanding of complex topics. By progressively animating algorithms, data structures, geometric transformations, and mathematical equations, Manim bridges the gap between theoretical abstraction and visual perception, facilitating the comprehension of dynamic processes that are difficult to convey through static media.

- **Automation and Scalability**: This flexibility is demonstrated by the ability to employ parameterization to modify values and inputs, enabling the efficient and automatic generation of varied visual models. This programmatic approach also supports reusability, as pre-built components can be integrated into broader educational systems or modified without requiring a complete redesign. Full automation enables the production of large and complex visual content with significantly reduced effort and resources.

2.5.2 Manim Architecture and Core Components

The Manim library relies on an integrated software architecture[19] comprising several essential elements that work in concert to produce scientific animations. These elements are:

Scene

The scene is the fundamental structure upon which all animations are built. It represents the digital stage on which elements are displayed. The scene organizes the chronological sequence of events and manages all operations related to visual objects. It contains the core `construct` function, which is used to define the animation logic, while maintaining a precise record of all objects present in the display frame.

Mobjects

Mobjects (mobile objects) are the library's basic visual building blocks, encompassing everything visible on screen. These objects are highly flexible, supporting various geometric transformations such as movement, displacement, and scaling, as well as dynamic control over color and position. They include geometric shapes such as circles and squares, mathematical text, graphs, arrows, and external images.

Animation

In Manim, an animation is defined as a transformation applied to a single object or a group of objects over a defined time interval. The nature of this change is governed by parameters including duration, easing curve, and delay intervals. Animations can be combined into groups (`AnimationGroup`) or sequenced to produce complex animated scenes.

Rendering Engine

The rendering engine converts the code and mathematical definitions of a scene into frames and subsequently into a final video output. The underlying technology varies by library version: ManimGL relies directly on OpenGL, while the Community Edition uses

either Cairo or OpenGL. This engine handles frame generation, camera perspective, lighting in 3D scenes, frame rate configuration (FPS), and video encoding.

Text Engine

The text engine is responsible for producing animated and transformable text objects. It relies primarily on the LaTeX[20] framework for rendering complex mathematical expressions. The engine provides complete control over font type, size, color, and alignment, converting plain text into `Text` objects and mathematical notation into `MathText` objects.

Camera

The camera determines the viewpoint of the scene and controls the final frame presented to the viewer. It supports dynamic operations such as zoom in, zoom out, and translation. In 3D scenes, it governs the pivot point, scaling, and depth angles through rotation parameters (phi, theta).

Window

The window is the live preview interface that appears to the developer during the construction process. It provides an immediate interactive view of the scene, allowing the programmer to inspect the output and make corrections before initiating the final video export.

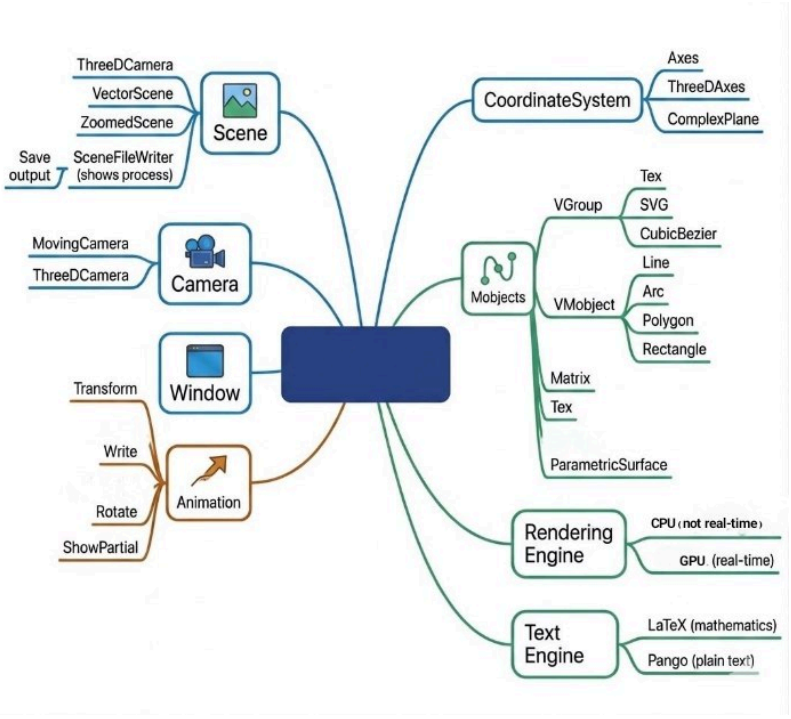

Figure 2.1: Architecture and core components of the Manim framework[19].

2.5.3 Manim Versions

The project has evolved into two distinct branches that serve different development needs.

ManimGL

ManimGL[21] is the version developed and maintained by Grant Sanderson. It uses the OpenGL interface exclusively for all graphics management and rendering. Unlike the Community Edition[22], it employs a more flexible object-oriented architecture that gives developers direct control over the way elements are displayed while providing an interactive environment for manipulating mathematical objects and animations. The key technical characteristics of this version are:

- **Interactive Preview Window:** ManimGL[21] uses OpenGL to provide a live preview window in which changes are reflected immediately. The developer can pan, zoom, and navigate within the scene during construction.
- **GPU-Accelerated Rendering:** ManimGL uses the GPU to ensure smooth and efficient rendering, even for mathematically complex scenes.
- **Professional Mathematical Typesetting:** ManimGL[21] integrates LaTeX[20] natively, enabling precise and professional rendering of mathematical formulae.
- **Extensive Geometric Library:** ManimGL[21] provides a comprehensive set of built-in tools for constructing and transforming geometric shapes, well-suited to scientific visualization.
- **Programmatic Animation Logic:** As ManimGL[21] is Python[18]-based, animations are defined entirely through code, enabling full integration with Python[18] libraries such as NumPy.

ManimGL[21] incorporates updater functions and event listeners. Updaters allow Mobjects to be modified on every rendered frame through variable manipulation, while event listeners enable interaction with objects via mouse or keyboard input.

ManimCE

Manim Community Edition[22] (ManimCE) is a fork of the original Manim library, maintained by a distributed group of contributors. Its primary objectives are to improve the library's stability, scalability, and accessibility for a wider developer audience.

ManimCE is built around a modular structure in which core subsystems the rendering engine, scene management, graphical objects, and the command-line interface are decoupled

from one another. This separation simplifies maintenance and facilitates integration with external software.

The key characteristics of ManimCE[22] are:

- **Stability and versioning:** ManimCE adheres to semantic versioning, ensuring that releases are stable and backward-compatible with existing projects.
- **Rendering flexibility:** ManimCE primarily uses the Cairo rendering backend for high-quality output, while progressively expanding its OpenGL support, providing compatibility across different operating systems.
- **Documentation and community support:** As a community-maintained project, ManimCE benefits from extensive documentation and a plugin ecosystem, making it a practical choice for educational and research applications.

Performance Efficiency

A significant technical distinction between the two Manim versions lies in their use of hardware resources. While ManimCE[22] relies on CPU-based calculations through the Cairo backend, ManimGL[21] delegates rendering to the Graphics Processing Unit via OpenGL. Rather than performing computationally intensive mathematical operations on the Central Processing Unit, ManimGL offloads fragment and vertex processing to the GPU, resulting in substantially faster rendering and higher frame rates. The GPU's capacity to process large numbers of vectors in parallel makes interactive preview viable.

Table 2.1 presents a comparative analysis of ManimGL[21] and ManimCE[22], highlighting their key technical differences:

Criterion	ManimCE (Community Edition)	ManimGL
Main Developer	Community (fork since 2020)	Grant Sanderson personally
Stability	Very high; careful versioning and backward compatibility	Experimental; frequent changes, occasionally breaking
Documentation	Comprehensive, organized, and regularly updated	Limited and irregularly maintained
Default Rendering Engine	Cairo (stable) with increasing OpenGL support	Primarily OpenGL
Compatibility & Installation	Straightforward and reliable across platforms (Windows, macOS, Linux)	More complex; common OpenGL configuration issues
Renderer	Cairo / OpenGL (gradual transition toward OpenGL)	OpenGL (GPU)
Live Preview	Relatively limited	Available; immediate and efficient
Animation Quality	High	High; motion can be smoother
Community Support	Large and highly active	Moderate (centred on Grant and a small group of contributors)

Table 2.1: Comparison between ManimCE and ManimGL[23].

2.5.4 Taxonomy of Manim Objects

`VMobject` in Manim refers to all graphical elements created using vector graphics. All shapes in Manim are `VMobject` instances because Manim does not use rasterized images; instead, it employs a mathematical approach to construct geometric objects.

Mathematically-Based Geometric Structure

Every shape, from a basic circle to a complex custom polygon, is defined using a set of coordinates and control points connected mathematically through Bézier curves [24], specified by anchor and handle coordinates. Rather than storing the color of each pixel, the system computes the path of these points.

2.5.5 Advantages of Vectorized Geometries

This geometric representation offers several key benefits in scientific visualization:

- **Vector-Based and Scalable:** The mathematical definition of shapes ensures perfectly crisp edges at any resolution, allowing infinite scaling without quality degradation an

important property for educational visualizations where zooming is required.

- **Compositional Foundation:** `VMobjects` serve as the fundamental building blocks from which all complex animations are constructed, including vector arrows, coordinate axes, and function graphs.
- **Dynamically Configurable Properties:** Defining shapes in Python[18] gives the developer full programmatic control over stroke, fill, and other visual properties, enabling their modification throughout the animation process.
- **Geometric Transformation:** Because shapes are represented as sets of discrete points, Manim can interpolate between point configurations to animate the morphing of one geometric figure into another.

2.5.6 Common Shapes in Manim

The Manim library provides a broad set of ready-made classes for representing mathematical and geometric objects, enabling developers to construct complex scenes with minimal code. Table 2.2 lists the most frequently used shapes along with a brief description of each:

Class	**Description**	**Example Usage**
Circle	Represents a circle object	`Circle(radius=2, color=BLUE)`
Square	Represents a square shape	`Square(side_length=3)`
Rectangle	Standard rectangular object	`Rectangle(width=4, height=2)`
RoundedRectangle	Rectangle with rounded corners	Suitable for cards and interface elements
Ellipse	Elliptical shape	`Ellipse(width=4, height=2)`
Annulus	Ring-shaped object	`Annulus(inner_radius=1, outer_radius=2)`
Triangle	Equilateral triangle	`Triangle()`
Polygon	Arbitrary polygon shape	`Polygon([p1,p2,p3,p4])`
RegularPolygon	Polygon with equal sides	`RegularPolygon(n=6)`
Star	Star-shaped object	`Star(n_points=5)`
Line	Straight line segment	`Line(start=LEFT,end=RIGHT)`
Arrow	Directed line with arrowhead	`Arrow(start=LEFT,end=RIGHT)`
CurvedArrow	Curved directional arrow	Suitable for diagrams and flow transitions
DashedLine	Dashed line segment	`DashedLine()`

Table 2.2: Common shapes provided by Manim.

2.6 Limitations of the Manim Framework

Manim is a promising framework for creating educational content; however, as it was originally designed for mathematics education, it lacks support for computer science-specific visual elements such as data structure representations. Furthermore, it provides no built-in support for interactive behavior.

2.7 Conclusion

This chapter presented the technical foundations underlying modern educational visualization systems, with particular focus on the Manim animation framework. The discussion covered the fundamentals of computer graphics rendering, low-level graphics libraries, and high-level visualization frameworks before examining the architecture and design philosophy of Manim and its OpenGL-based variant, ManimGL.

Although Manim provides a capable environment for mathematical animation and scientific visualization, several limitations remain with regard to interactivity and support for computer science educational content. These limitations motivate the development of the proposed Manim-CS extension, which is presented in the next chapter.

System Design & Implementation

3.1 Introduction

This chapter explains the Framework's architecture and operating logic. This interactive learning framework is designed to bridge the gap between abstract computer science concepts and students' mental representations. The design relies on synchronized visual feedback, focusing on the key themes encountered in computer science instruction.

3.2 Framework Design Objectives and Requirements

The development of Manim-CS followed a set of educational and technical objectives derived directly from the functional gaps analyzed in Chapter I. The requirements fall into two categories: functional requirements that define what the Framework must do, and non-functional requirements that constrain how it must perform.

3.2.1 Functional Requirements

The following functional requirements were defined to address the limitations identified in the review of existing educational tools:

- **Interactive, dynamic visualization** of algorithms and data structures, allowing instructors to modify algorithm parameters and observe the effects immediately during live lectures directly addressing the adaptive rigidity of pre-recorded educational videos and presentations.
- **Responsive manipulation of graphical elements** through mouse and keyboard events, enabling instructors to insert, delete, and reorder data structure elements during demonstrations.
- **Simulation of memory operations**, pointer manipulation, and recursive execution with synchronized visual feedback, transforming abstract low-level concepts into observable graphical events.
- **Synchronization between algorithmic execution steps and graphical animations**, ensuring that each computational operation maps to an explicit visual state change.
- **Modular and reusable educational component design**, allowing lesson modules to be composed, extended, and integrated into new educational contexts without significant

rework.

- **Support for interactive educational demonstrations** during live lectures and independent student study sessions.

3.2.2 Non-Functional Requirements

Beyond its functional capabilities, the Framework was also required to satisfy several non-functional properties:

- **High rendering performance** through GPU acceleration, ensuring smooth animation at interactive frame rates.
- **Low-latency user interaction**, with visual feedback occurring within a single rendering frame of any user input event.
- **Modular and maintainable software architecture**, enabling independent development, testing, and extension of individual components.
- **Scalability for future educational content**, achieved through the reusable component design described in Section 3.8.
- **Cross-platform compatibility** across standard desktop operating systems.
- **Accurate rendering of mathematical expressions** and support for multilingual text, including Arabic right-to-left layouts.

3.3 Framework Architecture

The Manim-CS Framework follows a multi-layered software architecture in which each layer provides well-defined services to the layers above it while maintaining logical independence from higher-level concerns. This separation of responsibilities facilitates maintenance, simplifies testing of individual layers, and allows the instructional content layer to evolve without requiring changes to the underlying rendering infrastructure.

The architecture consists of four layers: the top-level instructional scenarios layer, which contains all interactive lesson modules; the Manim-CS Framework layer, which coordinates the instructional logic and visual presentation behavior; the ManimGL animation engine layer, which manages scene rendering and event processing; and three parallel low-level subsystems: OpenGL for GPU rendering, Pango[25] for text rendering, and LaTeX[20] for mathematical typesetting. The following figure illustrates the overall organization of the Manim-CS Framework architecture.

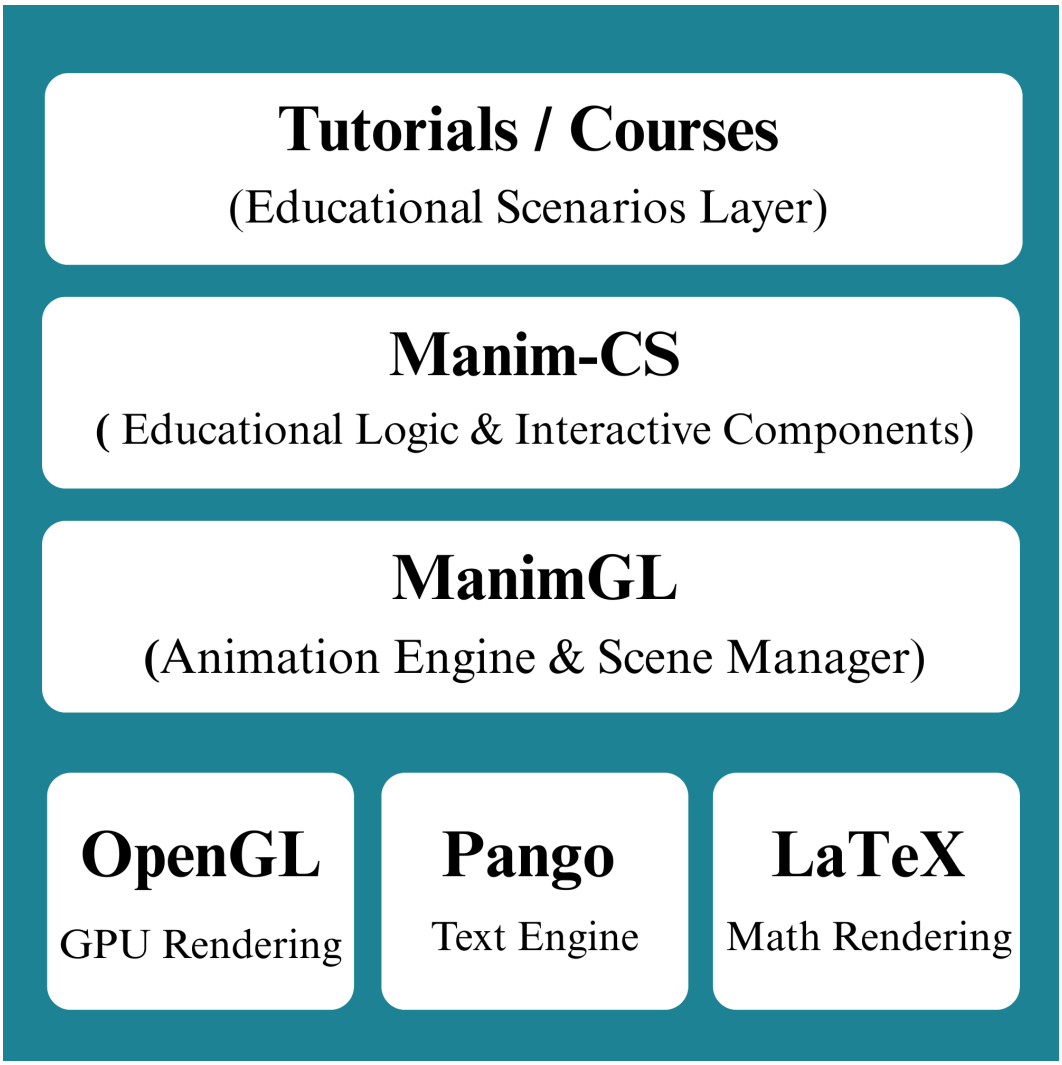

Figure 3.1: Multi-layered architecture of the Manim-CS Framework.

3.3.1 Educational Scenarios Layer

This layer represents the educational interface of the Framework. It contains all interactive lesson modules and algorithm visualization scenarios designed for computer science instruction, including data structure simulations, memory representation modules, and algorithm visualizations. It transforms theoretical concepts into dynamic visual experiences by composing the reusable components provided by the Manim-CS Framework layer.

3.3.2 Manim-CS Framework Layer

The Manim-CS layer is the educational core of the Framework. It coordinates interaction logic, manages educational state transitions, synchronizes animations with algorithmic execution steps, and provides the reusable visualization components such as `InteractiveList`, `MemSys`, and `CallTree` used by all lesson modules. This layer acts as the bridge between

educational logic and the graphical rendering engine, ensuring that any change in algorithmic state is immediately reflected in the visual representation.

3.3.3 ManimGL Animation Engine Layer

ManimGL[21] serves as the primary animation engine, managing the scene graph, object sequencing, and the interactive OpenGL window. The engine continuously updates graphical objects during execution, enabling smooth animation playback and responsive handling of user input events.

3.3.4 Lower-Level Rendering Subsystems

Three specialized subsystems operate at the lowest architectural level:

- **OpenGL** provides GPU-accelerated rendering of all graphical objects, converting vector-based geometric data into pixel output through the graphics hardware pipeline.
- **Pango** handles multilingual text rendering, including Unicode support, right-to-left Arabic text processing, and dynamic interface labels.
- **LaTeX** provides high-precision mathematical typesetting, converting mathematical expressions into scalable vector graphics through a DVI-to-SVG pipeline, which are then integrated into the Manim scene as `VMobject` instances.

3.4 Technology Selection and Design Rationale

The core technologies used in Manim-CS were not chosen arbitrarily. Each decision was made after evaluating alternatives against the Framework's specific requirements particularly the need for interactive rendering, mathematical precision, and programmatic control over animation logic. This section documents the reasoning behind the three most significant architectural decisions.

3.4.1 ManimGL vs Alternative Animation Frameworks

Several candidate frameworks were considered for the animation and rendering engine, including Unity, WebGL [26] /Three.js [27], and Manim Community Edition (ManimCE). Table 3.1 presents a comparative analysis of these alternatives against the specific requirements of Manim-CS.

Criterion	**Unity**	**WebGL / Three.js**	**ManimCE**	**ManimGL**
Primary purpose	Game engine; not designed for mathematical visualization	General-purpose 3D web rendering	Mathematical animation for video export	Mathematical animation with an interactive window
Real-time interaction	Supported, but requires extensive scripting setup	Supported via JavaScript event model	Limited; primarily designed for offline video rendering	Native; built-in event loop
Mathematical precision	Not native; requires custom implementation	Not native; requires additional libraries	High; LaTeX-based typesetting	High; LaTeX[20] and Pango[25] built in
Programmatic control	Requires C# scripting	Requires JavaScript callbacks	Python[18]-based; high control	Python[18]-based; full control
GPU rendering	Yes, via DirectX/ Vulkan/OpenGL	Yes, via WebGL	Partial; primarily CPU-based	Yes; direct OpenGL via ModernGL
Python[18] integration	No; language boundary	No; language boundary	Yes; full Python ecosystem	Yes; full Python ecosystem
Desktop deployment	Requires Unity Editor	Requires browser and server	Yes	Yes; single Python environment
Learning curve	High; requires C# and game development concepts	High; requires JavaScript and web technologies	Moderate	Moderate; standard Python

Table 3.1: Comparative analysis of animation framework alternatives.

Based on this analysis, ManimGL was selected for three primary reasons. First, it is the only framework in this comparison that provides a native interactive window with built-in keyboard and mouse event handling a fundamental requirement for live classroom use. ManimCE, despite its stronger community support and documentation, is designed primarily for offline video rendering and does not expose an interactive event loop suitable for dynamic manipulation during a session. Second, ManimGL natively integrates LaTeX[20] and Pango[25], ensuring the mathematical precision required for computer science educational content without any custom text-rendering work. Unity and WebGL do not provide this out of the box. Third, ManimGL operates entirely within the Python[18] ecosystem, allowing direct integration with NumPy for numerical operations and JSON for lesson state

persistence, without the language-boundary overhead introduced by Unity (C#) or WebGL (JavaScript). Choosing ManimGL over Unity and WebGL therefore represents a deliberate trade-off: lower visual production value and no web-based deployment, in exchange for native interactivity, mathematical precision, and programmatic control within a unified Python environment.

3.4.2 OpenGL vs Higher-Level Rendering Engines

The decision to use OpenGL directly accessed through the ModernGL Python[18] interface rather than a higher-level abstraction such as Pygame[28], PyGame Zero[29], or a browser-based canvas, was driven by the Framework's performance requirements. Higher-level rendering engines typically introduce an abstraction layer that buffers drawing operations and manages its own update cycle. While this simplifies development for general-purpose applications, it conflicts with the fine-grained frame control required for educational animation: in Manim-CS, every graphical object must be redrawn within a single frame in response to a user event, without waiting for the engine's own refresh cycle.

Using OpenGL directly provides three specific advantages in this context:

- **Frame-level control:** each call to `update_frame()` directly triggers a GPU draw cycle, ensuring that visual changes from user interaction appear in the immediately following frame with no intermediate buffering delay.
- **Vertex buffer object management:** ManimGL renders each graphical object as a VBO within the OpenGL context. Adding or removing an object from the scene takes effect in the next frame with no scene-graph traversal overhead a property essential for smooth animated insertion and deletion of data structure nodes.
- **Headless rendering support:** by substituting the EGL backend for the default X11 windowing system, OpenGL rendering can operate on Linux servers without a graphical display, enabling background rendering and potential future server-side deployment.

The use of OpenGL therefore reflects the need for deterministic, low-latency frame control that higher-level rendering abstractions do not reliably provide.

3.4.3 Event-Driven Architecture vs Alternative Interaction Models

The Manim-CS interaction model is built on an event-driven architecture in which user inputs keyboard presses, mouse movements, mouse button clicks, and scroll wheel actions are dispatched as discrete events to registered handler functions. This model was chosen over two main alternatives: a polling-based model and a command-based model.

In a polling-based model, the application continuously checks the state of input devices at every frame, regardless of whether any input has occurred. While straightforward to implement, this introduces unnecessary CPU overhead during periods of inactivity and couples input-handling frequency to the frame rate a problematic dependency in an educational context where the instructor may pause the visualization for extended periods while explaining a concept.

In a command-based model, user interactions are encapsulated as discrete command objects that are queued and executed sequentially. Although this model supports undo/redo functionality well, it introduces latency between a user's action and the visual response, and requires a command dispatcher infrastructure that adds architectural complexity without contributing to the Framework's educational objectives.

The event-driven model, as implemented through ManimGL's `on_key_press()`, `on_mouse_drag()`, and related handlers, provides three properties well suited to an educational context:

- **Immediate response:** events are processed synchronously within the rendering loop, ensuring that the visual consequence of each user action appears in the immediately following frame.
- **Decoupled interaction:** each event handler is independent of the others, allowing lesson modules to register only the events relevant to their scenario without interference from unrelated input channels.
- **Natural fit for demonstration pedagogy:** the event-driven model mirrors the structure of a live educational demonstration, where the instructor performs a discrete action (e.g., pressing `H` to insert a node) and the Framework responds with a corresponding animation a direct mapping between instructor action and Framework response that supports effective pedagogical communication.

3.5 Event-Driven Interaction Model

The event-driven interaction architecture of Manim-CS is implemented through a set of event handlers integrated directly into the ManimGL scene environment. Unlike static animation systems that execute a fixed sequence, Manim-CS continuously processes user input during execution, allowing the instructor to intervene, modify, and redirect the demonstration at any point.

3.5.1 Keyboard Interaction

Keyboard events are dispatched through the `on_key_press()` and `on_key_release()` handlers, which are overridden in each lesson scene to register the shortcuts relevant to that module. Keyboard interaction controls animation playback, triggers data structure operations (insertion, deletion, traversal), navigates between algorithmic steps, and switches between visualization modes.

3.5.2 Mouse Interaction

Mouse interaction is managed through a three-phase protocol press, drag, release implemented via `on_mouse_press()`, `on_mouse_drag()`, and `on_mouse_release()`. These handlers enable direct manipulation of graphical elements: node selection by click, spatial repositioning by drag, value editing by double-click, and viewport navigation (panning and zooming) through the `CameraController` subsystem.

3.5.3 Synchronized Frame Updates

The `update_frame()` method is invoked at every rendering cycle and acts as the primary synchronization point between the application state and the visual output. At each cycle, object positions are recalculated, `DynamicArrow` connectors are refreshed to follow their target objects, animation sequences are advanced, and interface states are updated. This continuous synchronization ensures that the visual representation remains consistent with the underlying data model at all times, regardless of the speed or sequence of user interactions.

3.6 Rendering and Visualization Pipeline

The rendering pipeline transforms high-level Python[18] object definitions into the interactive graphical output displayed in the ManimGL window. The pipeline operates across three functional stages.

3.6.1 GPU-Accelerated Rendering

All graphical objects in Manim-CS are rendered using the OpenGL graphics pipeline, accessed through the ModernGL interface. Each `VMobject` is converted into geometric vertex data and stored as a vertex buffer object (VBO) on the GPU. The rendering engine processes these buffers at every frame, producing the final pixel output shown in the interactive window. This hardware-accelerated approach enables smooth animation at interactive frame rates, even for scenes containing multiple concurrently animated data structures.

On Linux systems without a graphical display, the OpenGL context is initialized using the EGL backend rather than the default X11 windowing system, allowing rendering to proceed without a monitor or desktop environment.

3.6.2 Mathematical Expression Rendering

Mathematical expressions and code annotations are processed through a LaTeX[20]-based rendering pipeline. The source expression is first compiled into Device Independent (DVI) format. The DVI output is then converted into Scalable Vector Graphics (SVG) using the `dvisvgm` tool, producing resolution-independent vector paths. These paths are loaded into the Manim scene as `VMobject` instances via the `tex()` and `mathtext()` classes, ensuring that all mathematical annotations remain sharp and geometrically precise at any zoom level.

3.6.3 Multilingual Text Rendering

General-purpose text content interface labels, node value annotations, array indices, and editor content is rendered using the Pango[25] text engine via the `text()` class. Pango provides full Unicode support, correct handling of right-to-left Arabic text, and variable-width font rendering. The selection rule applied throughout the Framework is straightforward: mathematical content uses `tex()` (the LaTeX[20] pipeline); all other text uses `text()` (the Pango[25] pipeline).

3.6.4 Vector-Based Object Representation

All graphical objects in Manim-CS are represented as `VMobjects` vectorized mathematical objects defined by sets of anchor and control points connected by Bézier curves. This representation provides resolution-independent scalability, smooth interpolation during animation, and dynamic geometric transformation all essential for educational visualizations where zoom, pan, and animated morphing must preserve visual quality.

3.7 Development Environment and Tools

The Manim-CS Framework is implemented in Python[18] as its primary language, integrated with the following specialized libraries and development tools:

Tool	Technical Function
Python	Core programming language responsible for scene logic, data structure management, and event-driven control flow.
ManimGL	Primary animation engine managing GPU-accelerated vector graphics generation, scene management, and animation sequencing.
Pyglet	Manages the interactive window lifecycle and dispatches low-level mouse and keyboard events to the application layer.
NumPy	Provides high-performance numerical operations and three-dimensional coordinate transformations.
JSON	Used to store and retrieve persistent lesson data, including code editor content and session notes.
Visual Studio Code	Primary integrated development environment with Python[18] support.
Git	Version control system used for tracking incremental changes across the development lifecycle.
Venv	Virtual environment manager used to isolate project dependencies from system-level Python[18] installations.

Table 3.2: Development tools used in the Manim-CS Framework.

3.8 Core Software Components

The Manim-CS Framework is organized as a collection of reusable Python classes grouped into two subsystems: a shared infrastructure layer providing cross-lesson services, and a component layer providing the pedagogically oriented visualization and simulation classes used by the lesson modules.

3.8.1 Shared Infrastructure

Color Management This module serves as the single authoritative source for all color constants across the Framework, organized into functional groups: a general UI palette (background, highlight, neutral), node and indicator colors (selection, traversal, found state), notepad theme colors, operation button colors, and comparative visualization colors for the Array vs. Linked List scenario.

Utility Classes Three reusable utility classes and one function are available to all lesson modules:

- `sym_to_char`: a keyboard input parser that converts Pyglet key codes and bitmask modifiers into printable Unicode characters, supporting all printable ASCII characters.
- `CameraController`: wraps the ManimGL camera frame to provide interactive panning

via middle-mouse-button drag and zooming via scroll wheel. Camera behavior is configurable through `pan_speed`, `zoom_speed`, `min_zoom`, and `max_zoom` parameters.

- `DynamicArrow`: maintains a live graphical arrow between two dynamically positioned points. Both endpoints are specified as lambda functions, ensuring the arrow automatically follows moving objects. The arrow is recomputed on every call to `refresh()`, maintaining persistent visual connectivity between nodes during drag operations.

Help Window and Integrated Notepad A floating keyboard-shortcut overlay provides context-sensitive help within each lesson scene. The integrated notepad allows instructors to annotate lessons during live sessions, with persistent JSON-based storage, scene-specific note isolation, undo/redo stack support, automatic text wrapping, and clipboard integration.

3.8.2 Visualization SubFramework

Linked List Component Three classes implement the linked list visualization. `DraggableNode` represents a single node with a value rectangle, a next-pointer field, and an index label; it supports direct value editing, traversal highlighting, spatial displacement, and mouse hit-testing. `InteractiveList` manages the full node collection, routing mouse events to individual nodes and providing `add_node()`, `remove_node()`, `traverse_animation()`, and `search_animation()` operations. `ValueBox` provides a modal input widget for capturing new values during interactive operations.

Array Component The array component maintains three parallel lists of graphical objects (cell rectangles, value labels, index labels) and exposes a clean interface for cell selection, value updates, full reconstruction, and mouse-based cell identification through bounding-box hit-testing.

3.8.3 Memory Simulation SubFramework

Four classes implement the RAM visualization model. `MemCell` represents an individual memory cell with variable name, address, and value fields, supporting dynamic value modification, flash alerts for read/write events, and state transitions between allocated and freed states. `Cellgrid` manages the two-dimensional spatial layout of `MemCell` objects. `PtrArrow` maintains a curved graphical arrow between a pointer cell and its target, with adaptive dynamic re-rendering that keeps the arrow accurate during interactive manipulation. `MemSys` provides the high-level API: `update()` modifies stored values, `free()` releases allocated memory, and `hook_text()` binds code-editor lines to memory cell operations enabling the live code-to-memory synchronization that is central to the Pointers module.

3.8.4 Recursive Tree Component

The recursive visualization system comprises five main elements: a trace engine that intercepts all call and return events; a zigzag layout algorithm that distributes tree nodes to conserve vertical space; `DraggableNode` instances configured as interactive call-frame representations with color-coded status (pending, active, returned, base case); `DynamicArrow` connectors with adaptive Bézier curves; and a `CallTree` manager that coordinates push/pop operations with the call stack.

3.9 Examples of Interactive Learning Scenarios

The five interactive lesson modules of Manim-CS share a common pedagogical architecture: each scene is initialized with a well-defined visual state, processes user interactions through the event-handling pipeline, and dynamically reconstructs its interface to reflect each operation. Rather than providing an exhaustive walkthrough of every module, this section presents a selection of representative examples that illustrate how the Framework behaves in practice.

3.9.1 Binary Search Module

The Binary Search module pre-computes all algorithm states as an ordered sequence of snapshots using `compute_snapshot()`. Each snapshot captures the complete array state — cell colors, border states, and boundary indicator (`LOW`, `MID`, `HIGH`) positions. The visualization can apply any snapshot on demand via `get_to_snapshot(i)`, enabling constant-time navigation between algorithm steps without re-executing the algorithm. This stateless rendering architecture separates algorithm logic from visualization state a design pattern that generalizes well to any deterministic algorithm visualization.

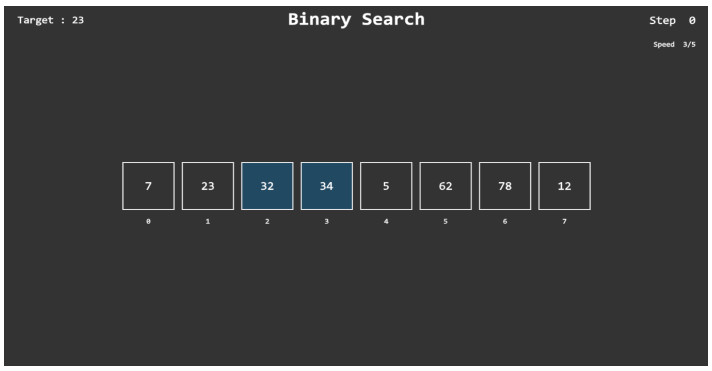

Figure 3.2: Sorting the array in ascending order before starting the algorithm.

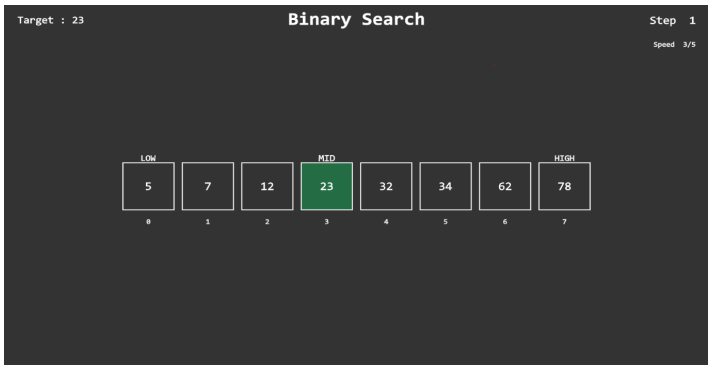

Figure 3.3: Matching the element and highlighting it in green when the target is found.

3.9.2 Recursion Module

The Recursion module renders a live call tree for recursive function execution. In step-by-step mode, the `CallTree` manager pushes a new `DraggableNode` for each recursive call and connects it to its parent with a `DynamicArrow`. When a base case is detected, the node turns pink and the explanation panel displays the resolution message. During the unwinding phase, return values propagate upward through the tree and appear within each parent node, providing a spatial representation of the call stack that reveals the accumulation structure invisible to static code inspection.

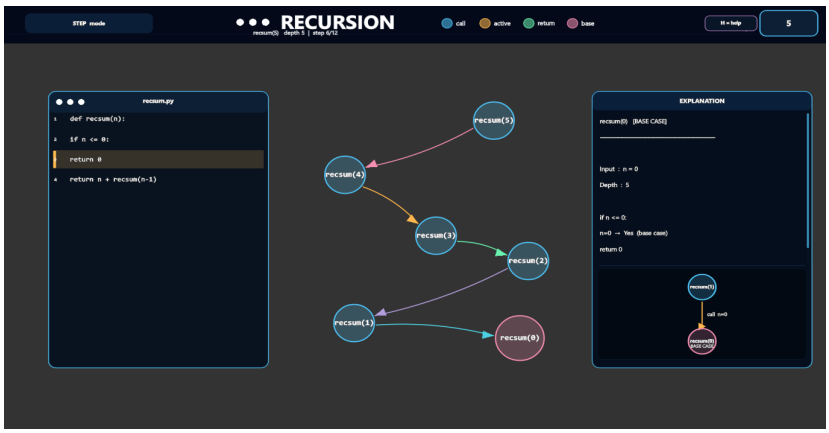

Figure 3.4: Constructing the recursive tree downwards until the base case is reached.

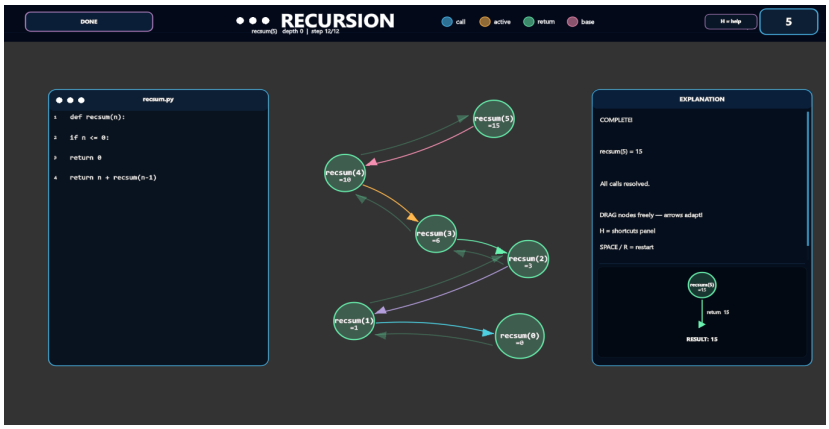

Figure 3.5: The resulting tree after computing the final return value.

3.10 Survey Results and Analysis

A structured questionnaire was distributed to computer science students across five academic levels, as illustrated in the figure below:

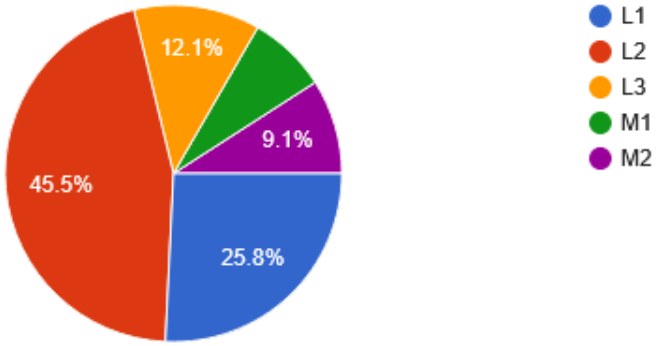

Figure 3.6: A total of 66 valid responses were collected.

The questionnaire evaluated three main dimensions: the perceived educational value of the Framework, the quality of its visual output, and students' acceptance of its use as a supplementary tool in algorithm courses. Table 3.3 summarizes the main findings.

Survey Item	Results
Preference for integrating the Framework into algorithms courses	97.0% supportive (64 out of 66)
Level of agreement on integrating the Framework	53.0% strongly agree; 37.9% agree; 9.1% neutral
Perceived reduction in time and effort for understanding complex algorithms	63.6% completely; 30.3% partially; 6.1% no
Ease of understanding through visual presentation (1--5 scale)	Average = 3.86 / 5
Clarity of visual output (arrow movement, memory values)	47.0% excellent; 40.9% good; 12.1% average
Effectiveness of code-memory synchronization in understanding pointer logic	53.0% significant; 42.4% slight; 4.5% no difference
Frequency of difficulty in tracking memory data without visual support	53.0% always; 39.4% sometimes; 7.6% rarely
Difficulty of abstract concepts using traditional methods (1--5 scale)	Average = 3.55 / 5

Table 3.3: Summary of student survey results.

The overall reception of the Framework was highly positive. Nearly all participants expressed a preference for integrating it into their coursework, and a large majority either agreed or strongly agreed with its adoption.

In terms of learning efficiency, students reported that Manim-CS significantly reduced the cognitive effort required to understand complex algorithms. The visual output was generally well received, with most participants rating it as good or excellent. The code-to-memory synchronization feature, in particular, proved highly effective in helping students follow pointer-based logic.

These findings are further reinforced by the baseline difficulty reported for traditional learning methods. Many students indicated that tracking memory operations without visual support is a recurring challenge, and the perceived difficulty of abstract concepts in traditional instruction remains relatively high.

Overall, the results suggest that Manim-CS effectively addresses a clear gap in algorithm education by improving conceptual understanding and increasing student engagement.

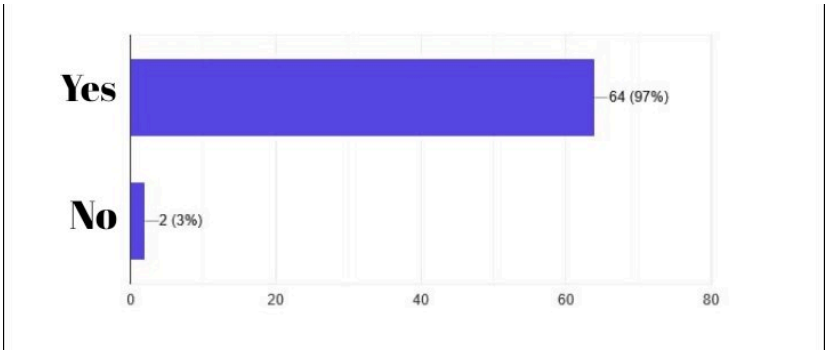

Figure 3.7: The general level of acceptance for integrating the Framework into the Algorithms course.

3.11 Discussion

The design and implementation of Manim-CS demonstrates several analytically significant properties worth examining beyond their functional description.

The synchronization between the code editor, the memory panel, and the graphical structure display as implemented in the Pointers module via `MemSys.hook_text()` addresses a fundamental limitation of static educational media: the inability to make the causal relationship between a line of source code and its effect on program state directly observable. By coupling the step controller to both the editor and the memory visualization layer, the Framework makes this relationship explicit and traceable at the granularity of individual instructions.

The snapshot-based architecture of the Binary Search module illustrates a broadly applicable design pattern: decoupling algorithm logic from visualization state. Pre-computing all states enables $O(1)$ navigation between steps and eliminates the need to re-execute the algorithm during backward traversal, removing a common limitation of sequential animation-based algorithm visualizers.

The parallel animation architecture of the Array vs. Linked List scenario demonstrates that simultaneously visualizing two competing algorithmic behaviors within a single interactive session is more pedagogically effective than comparing them sequentially. Viewing both side by side allows the student to perceive the structural difference as a spatial and temporal phenomenon rather than as an abstract memorized fact.

Finally, the modular component architecture in which `MemSys`, `DynamicArrow`, `InteractiveList`, and related classes are reusable across all lesson modules ensures that the Framework's extensibility rests on composing existing components into new educational contexts rather than duplicating infrastructure.

3.12 Limitations and Future Work

Any research or development project encounters a range of technical and practical challenges during the design and implementation phases. This section outlines the current limitations of the Manim-CS Framework and the prospects for its future development.

3.12.1 Current Limitations

- The development of the tool was constrained by limited hardware resources, particularly the reliance on integrated graphics processors rather than dedicated graphics cards during development.
- Manim-CS requires a local installation of Python[18], ManimGL[21], ModernGL, and LaTeX[20], and does not have a web-based deployment option.
- Integration with the Pango engine[25] remains partially limited.
- The current code editor supports only two built-in languages: C and Python. Supporting additional programming languages requires the implementation of a dedicated language parser and syntax engine.
- The Framework is designed to serve a single instructor or student at a time.
- The number of scenes is fixed at compile time. Adding or removing scenes requires direct modification of the source code.
- The Framework is comparatively more complex than simpler Python-based teaching tools and may present a barrier for instructors unfamiliar with Python.

3.12.2 Future Work

- Package the Manim-CS application as WebAssembly using Pyodide, or migrate the rendering pipeline to a WebGL [26]/Three.js [27] architecture, enabling in-browser operation and integration with learning management systems (LMS) such as Moodle [30].
- Improve the integration of the Pango engine [25] to ensure native Arabic language processing, accurate bidirectional text flow, and full compatibility with right-to-left layouts across all interactive editors.
- Set up a WebSockets server to stream the instructor's scene state to students' devices, enabling synchronized read-only viewing on student screens while the instructor retains full control of the main view.

- Develop a dual-render control system that separates the interface into an instructor-specific dashboard (containing hidden variables and controls) and a student-facing display.
- Support higher-performance development environments to increase production efficiency and enable the creation of more complex instructional modules.
- Expand the curriculum to include AVL tree visualization, graph traversal algorithms (BFS/DFS), hash tables, and dynamic programming models, by reusing core infrastructure components such as `MemSys`, `DynamicArrow`, and `InteractiveList`.
- Introduce high-contrast color themes to improve accessibility for visually impaired learners.
- Extend the Framework to support Android devices, enabling mobile learning and broadening accessibility for students beyond desktop environments.
- Add support for touchscreen interaction and stylus input, allowing more intuitive manipulation of diagrams and animations, particularly in tablet-based learning scenarios.

3.13 Conclusion

This chapter presents the complete design and implementation of the Manim-CS Framework, from its multi-layered architecture and the rationale for technology selection, to its core software components and interactive learning scenarios. It also discusses the Framework's current limitations and future development prospects, with the goal of establishing it as an interactive tool that supports the practical teaching of computer science.

Conclusion

This thesis represents the culmination of research and development efforts aimed at transforming abstract programming concepts into a flexible, interactive, and visually engaging educational Framework. This work resulted in the design and implementation of Manim-CS, a computer science teaching tool that transcends the limitations of traditional passive-learning methods and static instructional videos.

The reliance on the ManimGL library and its integration with an interactive event system proved to be a viable technical foundation. It facilitated the development of a clearly separated architecture between lesson logic and graphical infrastructure, ensuring the Framework's maintainability and enabling the future integration of more complex lessons with minimal additional effort. Beyond its technical aspects, the Framework underwent evaluation by computer science students. The results highlighted a significant positive impact on the understanding of complex concepts such as pointers, loops, and searching algorithms, attributable to the Framework's support for free navigation and systematic exploration. Despite current technical constraints from its restriction to desktop environments to the limitations of the hardware used during development Manim-CS opens promising directions in the field of educational technology. The proposed migration to web-based environments remains a qualitative step forward, one that would extend its reach from a laboratory tool to a broadly accessible resource for anyone engaged in the field.

References

[1] Microsoft, *Powerpoint*, 2026. [Online]. Available: https://www.microsoft.com/en-us/microsoft-365/powerpoint

[2] A. Inc, *Keynote for mac*, 2026. [Online]. Available: https://www.apple.com/keynote

[3] A. Paivio, *Mental Representations: A Dual Coding Approach*. New York: Oxford University Press, 1986.

[4] J. Sweller, "Cognitive load during problem solving: Effects on learning," *Cognitive Science*, vol. 12, no. 2, pp. 257–285, 1988.

[5] J. Sweller, P. Ayres, and S. Kalyuga, *Cognitive Load Theory*. New York, NY, USA: Springer, 2011.

[6] R. E. Mayer, *Multimedia Learning*. Cambridge, U.K: Cambridge University Press, 2020.

[7] R. A. Schwier and E. R. Misanchuk, *Interactive Multimedia Instruction*. 1993.

[8] E. Learning, *Constructivist learning theory*, https://elmlearning.com/hub/learning-theories/constructivism, Accessed: 26 March 2026, 2026.

[9] C. Bonwell and J. A. Eison, *Active Learning: Creating Excitement in the Classroom*. Washington, DC: The George Washington University, 1991.

[10] M. Prince, "Does active learning work? a review of the research," *Journal of Engineering Education*, vol. 93, no. 3, pp. 223–231, 2004.

[11] C. E. Wieman, "Phet: Simulations that enhance learning," *Science*, vol. 322, pp. 682–683, 2008.

[12] ThingLink, *Thinglink*, 2026. [Online]. Available: https://www.thinglink.com/fr/

[13] Desmos, *Desmos graphing calculator*, 2026. [Online]. Available: https://www.desmos.com/?lang=fr

[14] P. Jupyter, *Project jupyter*, 2026. [Online]. Available: https://jupyter.org

[15] Google, *Notebooklm*, 2026. [Online]. Available: https://notebooklm.google.com

[16] Microsoft Corporation, *Directx 12 documentation*, 2026. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/direct3d12/

[17] Khronos Group, *Vulkan - the essential guide*, 2026. [Online]. Available: https://www.vulkan.org/

[18] P. S. Foundation, *Python official website*, 2026. [Online]. Available: https://www.python.org/

[19] 3Blue1Brown, *Manim's structure*, 2026. [Online]. Available: https://3b1b.github.io/manim/getting_started/structure.html

[20] The LaTeX Project, *Latex/tex engines: A technical overview*, 2026. [Online]. Available: https://www.latex-project.org/help/documentation/

[21] 3Blue1Brown, *Youtube channel*, 2026. [Online]. Available: https://www.youtube.com/@3blue1brown

[22] M. C. Developers, *Faq: Installation*, 2026. [Online]. Available: https://docs.manim.community/en/stable/faq/installation.html

[23] T. of Beethoven, *Manimce vs manimgl*, 2022. [Online]. Available: https://www.youtube.com/watch?v=1tqtgnawBts

[24] P. Bézier, *The Mathematical Basis of the UNISURF System*. London: John Wiley & Sons, 1972.

[25] The GNOME Project, *Pango: Internationalized text layout and rendering engine*, 2026. [Online]. Available: https://pango.gnome.org/

[26] Khronos Group, *Webgl specification*, 2026. [Online]. Available: https://www.khronos.org/webgl/

[27] R. Cabello, *Three.js – javascript 3d library*, 2026. [Online]. Available: https://threejs.org/

[28] Pygame Community, *Pygame: Python game development*, 2026. [Online]. Available: https://www.pygame.org/

[29] D. Pope, *Pygame zero: A zero-boilerplate game development framework*, 2026. [Online]. Available: https://pygame-zero.readthedocs.io/

[30] Moodle Pty Ltd, *Moodle - open-source learning platform*, 2026. [Online]. Available: https://moodle.org/