Curricular consequences of programming without programming languages

Luděk Kučera
Charles University
Prague, Czech Republic

May 27, 2026

Abstract

The article examines the curricular consequences of an AI-supported programming method in which a programming project is developed z a human programmer into a complex structure of classes and objects with sufficiently simple and well-established class functions, whose bodies are described in human language. The functions are then translated into the target programming language by LLMs. The only difference with the pre-AI era is that AI took over the job of junior programmers. Our experience already comprises several thousands of lines of code and confirms the viability of the method.

The method is simple and natural, yet it has far-reaching implications. It is now possible to create complex projects without knowledge of programming languages. The common argument that a programmer must know the target programming language in order to evaluate, check, and verify the LLM output is not supported by our experience. Students who cheat on their homework assignments are also aware that AI-generated code tends to be correct.

The proposed method is *not* a vibe coding or NLP. A human programmer has full control of code, the method simply replaces the scheme ”senior:junior” by ”human:AI”. It can advantageously enter into synergy with approaches, where AI is used as a creative agent.

In a pedagogical experiment, undergraduate students without knowledge of Javascript but informad about the object oriented approach wrote their first HTML/-JavaScript visual application and found the work style quite natural and clear.

1 GenAI and Programming

There are many ways in which AI can support programming — generating complete programs, scripts, websites, or their parts (classes, functions, boilerplate code, etc.). See, for example, the fundamental articles [5], which presents benchmarks designed to measure the ability of LLMs to synthesize short Python programs from natural-language descriptions, and [16], which studies Python code-generation capabilities, as well as recent surveys of the field [20, 19], providing an overview of recent progress and numerous detailed references.

Another type of AI support involves explaining or commenting on generated code or the algorithms used, writing documentation, suggesting decompositions of a given problem, proposing solution strategies, or selecting suitable algorithms (which may, but need not, be accompanied by generation of the final code). Examples include [52], which proposes a program-synthesis language model that generates code progressively from coarse to fine in multiple passes, and [28], which constructs simpler subtasks for the LLM when a coarse prompt fails.

An important application of AI is debugging assistance (bug identification, crash explanation, stack-trace interpretation, and fix suggestion). Examples include [38, 10], which introduce debugging assistants collaborating with programmers; [39], which extends function-level code-repair capabilities to more complex repository-level scenarios; and the education-oriented AI debugger presented in [34], which actively engages students in the debugging process. The article [49] presents a comparative study of AI code-debugging assistants, while [26] attempts to align debugging procedures with the deductive processes of human developers.

AI can also be used to improve readability, modularity, naming conventions, and software architecture; see, for example, [12].

Another important use of AI is the evaluation of program properties such as computational complexity or security; see [29, 8].

AI agents can also act as interactive tutors. A comprehensive review of AI-based intelligent tutoring systems can be found in [51].

Last but not least, LLMs can be used as intelligent search tools querying large code bases for ready-to-use solutions to specific problems. One example will be presented later in Section 4.

The benefits of GenAI include speed, accessibility, rapid prototyping, learning support, and increased productivity, especially for repetitive tasks and boilerplate code.

However, several aspects of AI-generated code have been evaluated negatively. The article [47] argues that AI-generated code often exhibits poor non-functional quality characteristics (NFQC), such as reduced performance efficiency, code clarity, and security, making maintenance more difficult and increasing the qualification requirements for the human collaborator.

From an educational perspective, it is particularly interesting to study how programming students use generative AI. A growing body of literature investigates this issue.

The article [23] is a bridge between two parts of theis section by discussing educational aspects of GenAI viewed from the industry. There are some studies that simply observe behavior of students using AI when programming.

The study [21] investigates the dynamic interactions between students and ChatGPT during programming learning, and the article [41] argues that student behavior is better described by a Reverse Bloom Taxonomy: students begin with creation and only later move toward deeper understanding of the subject matter.

The authors of [31] found that students encountered significantly greater difficulty when correcting LLM-generated code than when solving traditional examination tasks, which is consistent with findings of [7] that programmers often face debugging an AI-generated code that they do not fully understand. In certain cases, sophisticated methods such as activity monitoring and eye tracking are used, see, for example, [42].

When evaluating use of AI by students of programming, studies like [50], arguing that using ChatGPT does not significantly affect students' learning performance in CS1 assignment-style tasks are less frequent than articles reporting negative aspects of AI use.

The authors of [18] found that, on examination questions taken from previous exams, Codex outperformed most students not only in CS1, but also in CS2.

A negative evaluation of using AI exhibits wide range of attitudes from expressing concern like [43], which warns about overreliance on generated outputs and the resulting decline in critical-thinking skills up to [14], which describes the misuse of AI by students for solving programming assignments with little or no effort as an "existential threat to current CS education models"..

A large group of studies reports a negative correlation between AI use and student performance.

The paper [37] reports that student performance negatively correlates with the reported frequency of AI-chatbot use, the authors of [35] found that higher confidence in GenAI is associated with lower levels of critical thinking and [25] reports a significant negative correlation between increased reliance on LLMs for critical-thinking-intensive tasks, such as code generation and debugging, and lower final grades, while [48] argues that overreliance on AI code completion may lead to only superficial understanding of programming concepts, diminished problem-solving skills, and reduced creativity.

However, the results cited in the previous paragraph are relativized by [15], which argues that studies examining correlations between ChatGPT use and learning performance often fail to establish causality, while [42] explicitly discusses an unfortunate divide between students who struggled and those who did not, with AI amplifying this difference.

The hard core of negatively oriented articles explicitly mentions cheating and plagiarism in connection with AI use.

The study [36] still presents "positive" findings: an anonymous survey suggests that cheating behavior among high-school students remained relatively stable after the introduction of the current generation of generative AI chatbots.

The anonymous study [36] further reports that the percentage of cheating in programming assignments increased even though most students did not consider the use of chatbots to complete entire assignments or papers acceptable, and according to [13], the emergence of ChatGPT significantly increases opportunities for undetectable AI-assisted academic dishonesty, requiring educational institutions to redesign assessment methods.

Several articles attempt to propose measures and tools addressing dishonesty and misuse. For example, [27] focuses on plagiarism and misuse of AI assistance in web programming and proposes the development of automated tools to help instructors identify both forms of misconduct. Similarly, [9] argues that the widespread deployment of LLMs may inadvertently foster academic dishonesty because of their easy accessibility, and proposes recommendations aimed at limiting excessive dependence on ChatGPT while promoting authentic competency development in programming.

While the previously cited education-relating papers were based on observation of students' behavior, several articles call for substantial changes in programming education.

The paper [32] presents four views of future curriculum development presented at a conference panel. The study [46] provides evidence that programming instructors must prepare for a world in which learners can use widely accessible AI technology to achieve passing scores on assessments that currently measure programming knowledge and skills, potentially with little effort.

The article [6] discusses ways of redesigning existing courses, reshaping assignments, and adapting pedagogy and the authors of [31] argue that programming education must increasingly focus on developing students' abilities to critically evaluate and correct AI-generated content.

The article [30], which resonates most strongly with the focus of our research, observes that "humans can now converse in their natural language, expecting machines to interpret and act. This shift has profound implications for computer science education. As educators, it is imperative to integrate this new dynamic into curricula."

A common characteristic of most of the educational papers cited above is that they observe and document how students use GenAI in ways developed independently by the students themselves or present just general and vague suggestions. In contrast, the present article attempts to provide students with a methodology that extends the observation from the previous quotation about "humans using their proper human language and machines interpreting and acting".

2 Well-established functions

I had the following conversation with ChatGPT:
Question: Which important software systems do you know?
Answer of ChatGPT:

Networking Protocols: TCP/IP, UDP, HTTP, HTTPS, FTP, SFTP, SSH, Telnet, SMTP, IMAP, POP3, DNS, DHCP, SNMP, BGP, OSPF, MPLS, SIP, RTP, RTCP, WebSocket, QUIC, gRPC, NTP, ICMP, ARP, PPP, IPsec, TLS/SSL, Ethernet, 802.11 (Wi-Fi), Bluetooth, ZigBee, LoRaWAN, NFC, SOAP, REST, GraphQL,

Operating Systems: Linux, Windows, macOS, iOS, Android, FreeBSD, OpenBSD, NetBSD, Solaris, AIX, HP-UX, ChromeOS, Ubuntu, Debian, Fedora, CentOS / Rocky Linux, Red Hat Enterprise Linux (RHEL), Arch Linux, OpenSUSE, Gentoo, Kali Linux, Alpine Linux,

Programming Languages:, C, C++, Java, JavaScript, TypeScript, Python, Ruby, Go, Rust, Swift, Kotlin, PHP, Perl, Haskell, Scala, Elixir, Erlang, C#, F#, VB.NET, Lua, R, MATLAB, Fortran, Ada, Pascal, D, Objective-C, Julia, Bash, PowerShell,

Major Runtimes: Virtual Machines, Node.js, Deno, Bun, JVM (Java Virtual Machine), CLR / .NET Runtime, V8 (Google JavaScript Engine), Spider Monkey, We- bAssembly (WASM), GraalVM, CRuby / JRuby / Truffle Ruby, CPython, PyPy,

Databases: SQL + NoSQL + NewSQL, Oracle, MySQL, PostgreSQL, SQLite, Microsoft SQL Server, IBM DB2, Mari- aDB, MongoDB, Redis, Cassandra, CouchDB, Elasticsearch, DynamoDB, Neo4j, Couchbase, Aerospike, ClickHouse, Snowflake, BigQuery, InfluxDB, TimescaleDB, RethinkDB, RocksDB, LevelDB, FaunaDB, HBase **Web Servers**: Apache HTTP Server, Nginx, Lighttpd, Caddy, IIS, Tomcat, Jetty, JBoss / Wild- Fly, GlassFish, uWSGI, Gunicorn, FastAPI (runtime), Flask (WSGI), Express.js, (GUI Frameworks) Qt, GTK, wxWidgets, Electron, Tk / Tkinter, JavaFX, Swing, WPF, Avalonia, FLTK

Cloud Platforms: AWS, Azure, Google Cloud Platform (GCP), IBM Cloud, Oracle Cloud, DigitalOcean, Linode, Heroku, Render, Cloudflare, Netlify, Vercel

Container & Virtualization Systems: Docker, Kubernetes, K3s, OpenShift, Podman, LXC / LXD, Vagrant, VMware vSphere /ESXi, VirtualBox, QEMU, Hyper-V

Distributed Systems: Big Data, Hadoop, Spark, Hive, HBase, Kafka, Zookeeper, Flink, Storm, Presto, Druid, Beam, Airflow, NiFi, Kinesis, RabbitMQ, ActiveMQ, Pulsar

Build Systems: Make, CMake, Me- son, SCons, Gradle, Maven, Ant, npm, Yarn, pnpm, pip, conda, cargo, go modules, bundler, composer, vcpkg, NuGet

Version Control: Git, Subversion (SVN), Mercurial (hg), GitHub, GitLab, Bitbucket, Jenkins, Travis CI, CircleCI, GitHub Actions, Ansible, Puppet, Chef, Terraform, Packer, SaltStack

Cryptography: OpenSSL, LibreSSL, GnuPG, HashiCorp Vault, Kerberos, OAuth, OpenID Connect, SAML, JWT, WireGuard, OpenVPN

Scientific: NumPy, SciPy, Pandas, Matplotlib, Seaborn, TensorFlow, PyTorch, Keras, OpenCV, scikit-learn, JAX, Theano, Caffe, MXNet (Game Engines) Unity, Unreal Engine, Godot, CryEngine, Source Engine, Frostbite, GameMaker Studio, Cocos2d, Panda3D, Ren'Py

Embedded Arduino, Raspberry Pi OS, Zephyr, FreeRTOS, Mbed OS, Yocto, OpenWRT, Home Assistant, ESP-IDF

Package Formats: XML, JSON, YAML, Protocol Buffers, Avro, Thrift, BSON, CSV

Question: What is the level of your knowledge? **Answer of ChatGPT**: Expert

Question: Which areas of mathematics do you know? **Answer**:

Elementary and Secondary Mathematics: Arithmetic, Algebra, Geometry, Trigonometry, Analytic geometry, Combinatorics, Probability, Mathematical induction

Calculus and Classical Analysis: Differential calculus, Integral calculus, Multivariable calculus, Vector calculus, Ordinary differential equations, Partial differential equations (many standard classes), Real analysis, Complex analysis, Fourier analysis

Linear and Discrete Mathematics: Linear algebra, Matrix theory, Graph theory, Discrete mathematics, Recurrence relations, Generating functions, Boolean algebra

Abstract and Higher Algebra: Group theory, Ring theory, Field theory, Galois theory (standard topics), Representation theory (basic to intermediate), Category theory (basic to moderate)

Geometry and Topology: Differential geometry (basic/intermediate), Projective geometry, Algebraic topology (introductory/intermediate), General topology

Number Theory: Elementary number theory, Modular arithmetic, Diophantine equations, Analytic number theory (selected topics), Cryptographic mathematics

Probability and Statistics: Probability theory, Statistical inference, Regression, Bayesian methods, Markov chains, Stochastic processes (standard models)

Applied Mathematics: Numerical analysis, Optimization, Operations research, Dynamical systems, Control theory (basic/intermediate), Information theory, Signal processing

Computer-Science-Related Mathematics: Algorithms and complexity, Automata theory, Formal languages, Logic, Computability, Cryptography, Machine learning mathematics

Mathematical Physics: Classical mechanics, Special relativity, Quantum mechanics mathematics, Tensor calculus (moderate level)

My limitations are:	**I am usually strongest when**:
I can make mistakes in long or highly intricate proofs.	Deriving formulas, constructing proofs, checking reasoning
Frontier research problems may exceed my reliability.	Explaining concepts.
Deep theorem proving sometimes requires verification.	Generating examples/counterexamples.
Very computation-heavy symbolic tasks may need external CAS tools.	*Translating mathematics into algorithms or code.*

My experience is that ChatGPT is right - any direct question to both systems and mathematics is answered at the expert level and any function, implementing a boilerplate system operation or a fundamental mathematical operation is translated correctly into a selected target programming language.

This is why such functions will be called *well-established functions*.

3 Programming without programming languages

3.1 A decomposition into well-established functions

There are many ways in which AI is used to help writing programs. In most cases, the reason for using AI support is increased productivity and time efficiency. The goal of this article is to describe another method with a different goal. In its strictest form, it brings only a low or moderate increase in productivity, but it offers another very important benefit: the way a human programmer communicates with a machine is elevated to the level of human thinking and communication, without using programming languages such as C++, Java, and Python.

It might sometimes seem that programming courses are primarily places where students learn the syntax of programming languages. However, it now appears very clearly that the main knowledge and competencies students acquire are the general principles of programming common to programmers in any language, while programming languages and their rigid syntax merely represented a syntactic vehicle that was necessary in the pre-AI era to deliver code to a machine.

Structuring control flow using constructs such as if-then-else branches, case statements, and loops of the form for-all, while-do, and do-until originates from human language. The human way of using such constructs is mostly ambiguous, because a spoken sentence is always interpreted within a context that determines its exact meaning. Their exaggerated use in programs is dictated by the requirement of absolute precision, and it is therefore perceived as being of human nature.

Consider texts produced by lawyers, who aim to express themselves in a way that has a unique meaning in all contexts while using a “natural language”. The structure of their documents often appears even more unnatural than the logical structure of programs.

The object-oriented approach (OOA), pioneered by Simula 67, was gradually introduced primarily to make software structure correspond more naturally to human conceptual models rather than to requirements of machine-level execution.

The use of strict typing is another feature that corresponds naturally to human thinking. Consider the books of John R. Tolkien, full of fictitious creatures, where each individual being is consistently described as a hobbit, elf, troll, dragon, etc., in much the same way programmers describe their variables as integers, mouse events, sockets, or objects of a defined class.

Control-flow structures, the object-oriented approach, and typed objects represent the most important knowledge students learn in programming courses, and all these ingredients are preserved as the fundamental building blocks of the proposed method.

The citations presented in the state-of-the-art section (Section 1) of this paper, as well as the practical example of the proposed method presented in Section 4, suggest that human programmers still significantly outperform AI agents in analyzing problem specifications and decomposing them into simple units, while AI excels at solving simple low-level problems, for example those denoted by the term “boilerplate”, mathematical operations or typical homework assignments from programming courses.

For this reason, the proposed method assumes a very strict division of roles between a human programmer and an AI agent. A human programmer decomposes the initial program specification into a collection of class definitions with their constructors, variables, functions, and derived objects. The description produced by a human is written in natural language, without using the special syntax of any established programming language (although this is not excluded).

There is no problem in presenting declarations of variables and function headers (i.e., what appears in standard .h header files in C/C++) to an AI agent.

Regardless of whether a declaration is written in natural language (“n is an integer”, “f receives an integer parameter n and returns a character”) or in the syntax of a programming language (“Int n;” or “Char f(Int n)”), AI agents have no difficulty understanding such declarations correctly.

In its strictest embodiment, the proposed method assumes that the functions used in the final iteration of the human decomposition of the program are as simple as possible, or are combinations of such functions constructed in a way that does not admit ambiguity.

The term “as simple as possible” refers to the impossibility of decomposing a function into a composition or cooperation of even simpler functions.

The term includes everything that can be described as “boilerplate”, but it is broader and also involves operations known in mathematics, computer science, physics, electrical engineering, and related fields.

The study by Chen et al. [16] introduces the HumanEval benchmark, consisting of Python function-generation tasks, many of which are mathematically straightforward algorithmic functions.

The results show that AI agents perform particularly well on boilerplate code generation and on implementations that directly follow well-defined mathematical or algorithmic specifications. We will call all such functions *well-established*.

The results of the study [16] are confirmed by subsequent articles, for example [22, 17].

As an example, the tetrahedron drawing presented in Section 4 is implemented using two kinds of functions: standard boilerplate functions that listen to mouse events and paint 2D geometric objects on the screen, on the one hand, and functions performing rotations of 3D space about its axes and determining whether the perimeter of a triangle XYZ is traversed clockwise, on the other hand.

Let us note that certain features of programming languages considered important and useful are, in fact, agreements between a human programmer and an AI agent, for example "private" and "public" objects or operator overloading. In the pre-AI era, a fixed set of such conventions with rigid syntax formed part of the definition of a language.

Now, a human programmer can simply and flexibly insert a directive into a prompt, for example, "a variable declared as 'private' must not be used outside this class", or "if '*' appears between two objects of the class *ComplexNumber*, translate the expression as ..."

In summary, the method proposed in this article follows a general goal formulated in [30]: "Humans can now converse in their natural language, expecting machines to interpret and act."

3.2 Debugging of AI-generated code

In the ideal case, the translated code in the target programming language, prepared by the proposed method, is immediately compiled and linked so that it can be executed without being inspected by the human programmers who prepared the prompt(s) (and who perhaps do not know the target language at all).

In articles describing AI-supported programming, it is usually assumed that AI-generated code in the target programming language must be "read, reviewed, debugged, and tweaked" (a quotation from [24]).

We use the proposed method to create our visual educational programs (already comprising several thousand lines of code), and in all cases the AI agent correctly translated well-established functions into the target language (JavaScript).

However, there were numerous bugs introduced by human programmers, which were present in the prompts for the AI agent and were translated literally, making debugging necessary.

A code written in a programming language (PL) is usually compiled and linked to produce machine code. When debugging PL-level code, we do not debug it at the level of machine code, but at the PL level.

Similarly, when debugging code prepared at the level of well-established functions described in human language, we do not need to read, review, debug, and tweak the translated code in the target programming language. Instead, we read, review, and tweak the description written in human language, and debugging focuses on tracing values defined in the human-readable description.

Currently, there are no development tools that support debugging at the level of well-established functions, written in human language. It can be expected that this situation will improve rapidly.

At present, we may ask the AI agent to give a code producing the computing log ("compile each function to sends the values of input parameters to the console after being called, and the return value just before returning"), which is then interactively analyzed using a DIY tool (our was written from scratch by ChatGPT).

Let us note that [7] reports that programmers currently often have to "debug code they didn't write", which means inspecting and validating AI-generated code that they do not fully understand.

This complication cannot occur when programming using the method proposed in the present article.

3.3 An existential threat to current education models

The article [14] describes the misuse of AI by students solving programming assignments as an "existential threat to current CS education models", and many other articles cited in Section 1 express similar concerns.

The article [14] is correct in stating that such student behavior is unethical, because students are violating an agreement under which they receive credits or grades for solutions obtained *by themselves* (and not by AI).

On the other hand, the part of current CS education models in which students are asked to implement programming assignments at the level of well-established functions (or their simple combinations) in a given programming language (C++, Java, Python, ...) is indeed disappearing, if not already dead, because this skill will no longer be needed in practice; asking AI for a solution will often be preferable even to simply minimize the number of keystrokes.

Let us recall that the most difficult part of programming is the analysis of complex problems and their decomposition into well-established functions, a task that AI agents still do not master well.

For this reason, we are convinced that even if well-established functions are implemented by AI, students will still have many opportunities to improve their understanding of programming concepts, problem-solving skills, and creativity.

3.4 Extension of the proposed method

In its strict form, the proposed programming method does not bring a substantial increase in productivity; its main advantage lies in eliminating the need to use programming languages directly.

However, it can be combined with any strategy cited in Section 1. One possibility is, during decomposition of the original problem, to ask AI to describe (in human language) a method for decomposing a complex function into well-established functions, or to translate the function into the target programming language *together* with an explanation of the procedure.

If an AI agent is able to perform a correct translation, it is also able to provide a explanation of the process.

Requesting a detailed explanation of how to determine the orientation of a triangle (used in Section 4) produced a two-page article, far more extensive than necessary, while the final code consisted of a single line computing a simple expression.

This approach largely eliminates the problem of "debugging code that we did not write".

3.5 Saint-Exupéry and progress

The great French aviator and writer Antoine de Saint-Exupéry wrote [44]: *Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.* ("It seems that perfection is achieved not when there is nothing more to add, but when there is nothing left to take away.") This famous quotation continues with a less well-known sentence: *Au terme de son évolution, la machine se dissimule.* ("At the end of its evolution, the machine conceals itself.") Both translations are by Lewis Galantière [45].

For Saint-Exupéry, *la machine* referred to an airplane. Let us note that the machine is still present in his statement, but the user may ignore its presence.

This perfectly captures the present state of programming languages, which are gradually disappearing from the sight and direct use of human programmers while still remaining present as an intermediate step in the conversion of human prompt-based descriptions into final executable code.

4 A pedagogical experiment

4.1 Participants

In order to get support for the results of this article, we asked 26 students of our university to prepare a HTML/JavaScript program solving a problem of medium complexity, even though a substantial part of them did not know JavaScript (they have a course of programming in Python).

Due to a highly non-standard nature of our experiment, it was not possible to include it into the main courses on algorithms or programming languages at our university. The experiment war run in an optional seminar for undergraduate students of computer science in the second semester of their study that explains the algorithms from the main DSA course using visualizations and introduces additional important algorithms that are not covered in the main course.

The seminar was offered to CS undergraduates under the thesis that "in the near future, AI will make writing the final code in particular programming languages much less important, but ability to design the architectore of programs, which includes knowledge of standard and advanced algorithms, will be the main skill of a computer scientist" and, despite heavy load of our undegraduates in their first year of study, the enrollement of the seminar is about 25 % of CS majors.

We decided to use a program with a visual output, because it is easier to check that the program works correctly. The goal was
Create a program in HTML/JavaScript, which draws a tetrahedron with random vertices and face colors, so that the tetrahedron can be rotated by the mouse to be observed under different angles.

Such a formulation of the goal suggests using the sentence as a prompt for an AI agent to get the code without any effort. The prompt was presented to GitHub Copilot [1], Anthropic Claude [3], OpenAI ChatGPT [4], and Google Gemini [3].

Copilot returned a perfect application, because it found a ready-to-use solution in the section Tetrahedron Geometry of the package THREE.js. The remaining three AI agents used direct drawing into the canvas - and all of them *failed*. Under some viewing angles, they painted faces of a tetrahedron that were invisible or their parts, see [1, 3, 4, 2]. This again confirmes that AI-generated programs *may fail and often fail* even when solving problems of moderate complexity.

Let us explicitly remind that the goal of the experiment was not to ask students to make the program decomposition as described generally in the previous sections and exemplified in the present experiment, but to invite them to use the description of the program decomposition presented by the instructor to prepare a prompt for an AI agent to translate it into an unknown target language.

This is why the instructor presented *orally*, with the help of Fig. 1 the algorithms used for tetrahedron painting, as described in Subsection 4.2, followed by the designed structure of the application, as given in theframed box at the end of the article, while projecting Fig. 1 and 2 and function heading, see the frame boxes with the darker background.

During the presentation, students were asked to write down a description of the application essentially as the gray background text in the next subsection and, combined into one prompt, to submit is to their favorite LLM for translation to HTML/JavaScript.

Students were allowed to work individually or in pairs, there was one group of 3 students.

4.2 Tetrahedron drawing algorithms

This subsection describes in full detail the architecture of a tetrahedron drawing program, but many observation presented here apply to programming in general and not only to the special case considered in the experiment.

Tetrahedron painting is an excellent example that a programmer with good knowledge of algorithms can find that writing a code for a problem of moderate to high complexity from scratch, leaving only obvious tasks to AI, is easier and *much more reliable* than asking AI to solve theproblem or studying a non-trivial philosophy of a codebase, which contains a ready-to-use solution.

The main part of our tetrahedron-painting application is the class *observer*, which contains three following obvious functions:
$rotateX(U)$ and $rotateY(U)$, which rotate the 3D space about the axe X or Y by some angle and return the coordinates of the image of U after rotation;$project(U)$, which projects a 3D point to the plane by forgetting the z-coordinate of U.

Our program will work with composition $transf(U)$ of the functions, defined as $project(rotateY(rotateX(U)))$.

Rotating properly a triangulated surface is not a problem, but the failure of Claude, ChatGPT and Gemini shows that it is not easy to show only visible faces of the body and hide the invisible ones.

There are two main approaches how to draw a triangulated surface to hide invisible faces of a tetrahedron in the screen.

One possibility is to use Painter's algorithm [40], (sometimes applied in the form of Z-buffer - especially in the case when painting is performed pixel-by-pixel). The algorithm sorts faces by their distance from the observer or camera and paint them in this order, the most distant first, closer later, hoping that the visible faces are closer to the observer and will completely cover the invisible ones.

However, the distance of a face can be defined in many ways (the maximum, minimum, or average of distances of its vertices) and obviously an incorrect distance definition was reason for AI agents' failure to paint a tetrahedron (ChatGPT and Gemini explicitly acknowledged using Painter's algorithm).

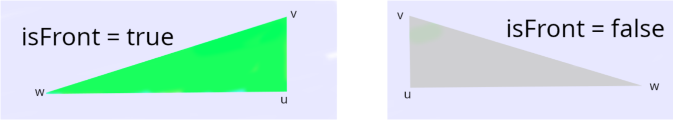

Figure 1: The green front and gray back of uvw

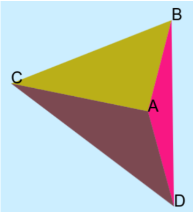

Figure 2: Orientation of faces

Another approach identifies invisible faces and does not paint them. We use the notion of an oriented triangle, which is a triangle in the plane with a fixed ordering of its 3 vertices. We say that we see the *front* of a triangle uvw, if visiting vertices u, v w in this order is running around the perimeter of the triangle counterclockwise, otherwise we see its *back*, see Fig. 1,

We checked that all above mentioned AI agents are able to decide between front and back view by checking the sign of the z-coordinate of the cross product of vectors $\overrightarrow{uv}$ and $\overrightarrow{uw}$ and hence we only use the description of the previous paragraph to define a function $isFront(U, V, W)$, which returns true if and only if the oriented triangle uvw, where u, v w are images of U, V W under the function $transform$, is visible from the front.

Now, the solution of of hidden invisible faces is simple: orient all faces so that they are visible when observed from outside (see Fig. 2, where the oriented triangles ABC, ADB, ACD are visible from front and oriented counterclockwise, and the hidden triangle, which would be visible from back, is oriented clockwise),
and then fill exactly those triangles, the transformation of which under selected view angles are visble from the front.

Write a program in JavaScript that has the following 4 objects:
`listener, painter, observer, and tester.`
The program also defines classes P2D with variables x,y, and
P3D with variables x,y,z, describing points in 2D and 3D space.

The object listener listens to mouse events; when a mouse event occurs, it calls the corresponding mouse function of tester, provided it exists

The object painter has a void returning function that fills the triangle with vertices v1,v2,v3 in the application window by the given color c:
`painter.fillTriangle(P2D v1, P2D v2, P2D v3, Color c)`

The object observer has two variables alpha, phi, representing angles in degrees, and functions

`P3D observer.rotateX(P3D U),`
which rotates the 3D space about the axe X by the angle alpha, and returns the image of U under the rotation.

`P3D observer.rotateY(P3D U),`
is similar, but rotates about the axe Y by the angle phi.

`P2D observer.project(P3D U),`
which projects U into 2D space by forgetting U.z,

`P2D observer.transf(P3D U),`
defined by transf(U) = project(rotateY(rotateX(U))),

`Boolean isFront(P2D a, P2D b, P2D c),`
which returns true if and only if the path abc goes around the perimeter of the triangle abc in the counterclockwise direction.

The object tester generates
4 randomly generated objects A,B,C,D of the class P3D, and
4 randomly generated colors
colorABC, colorADB, colorACD, colorBDC,
it has also a P3D variable cursorLocation0 and numeric variables alpha0, phi0.

The definition of its mouse handling functions is:

`tester.mouseDown(cursorLocation),`
which stores the parameter value in cursorLocation0 and values of variables alpha and phi of observer in alpha0,phi0,
and calls tester.paint(),

`tester.mouseDragged(cursorLocation)` sets
Number dx = cursorLocation.x-cursorLocation0.x;
Number dy = cursorLocation.y-cursorLocation0.y;
observer.alpha = tester.alpha0 + dx,
observer.phi = tester.phi0 + dy,
and calls tester.paint(),

`tester.mouseUp` is the same as tester.mouseDragged.

The object tester has also a void returning function
`tester.paintTriangle(P3D U, P3D V, P3D W, Color c):`
P2D u = observer.transf(U);
P2D v = observer.transf(V);
P2D w = observer.transf(W);
if observer.isFront(u,v,w)
then call painter.fillTriangle(u,v,w,c),

and a function
`Void paint():`
fill the application window by light blue color;
tester.paintTriangle(A,B,C,colorABC);
tester.paintTriangle(A,D,B,colorADB);
tester.paintTriangle(A,C,D,colorACD);
tester.paintTriangle(B,D,C,colorBDC);

5 Curricular implications

Let us mention that the main goal of the assay presented in this article was not to check students' ability to finish correctly one particular task, but to observe how they accommodate with a completely new style of creating a program. The results indicate that they have no problems to follow the method. Interviews after finishing the assay reveal two observations:

- Some students expressed rather surprised feeling that they did not need much of their course of (Python) programming (directed to low level feature of the language)
- essentially all students said that they now understand better that the main competency of a programmer will be and is the ability of the construction of the architecture of a program, while implementation features are of the kind of a technical skill that is less important for the final success of the program

Even though it does not seem that changes in teaching programming at our university will be slow and conservative, the following curriculum seems to be a viable alternative to the present arrangement:

- the introductory course of programming, bringing the fundamental general topics like the structure of the control flow as it is reflected in the natural language (if-then-else, for-all, do while/until, etc.) and the object-oriented approach (with classes, constructors, variables, functions and objects).
- Decomposition of nontrivial tasks into a structure based on modules, classes and objects with standard composition and cooperation of well-established simple functions formulated in natural language or by a reference to mathematics, computer science and software engineering, the function being later translated into a target language for compilation into the final program
- Using basic and more advanced algorithms in designing project structure according the previous item.

The author served as a reviewer of CS the Curricula 2023 [33] and invites the new initiative of Living Curricula [11] and continues his research while hoping that the new approach will be able to influence the near future CS curriculum in a way which will fully use advantages of using GenAI in programming.

References

[1] anonymous. Packalgo visualization (anonymized), a github copilot tetrahedron visualization application, 2026.

[2] anonymous. Packalgo visualization (anonymized), a google gemini tetrahedron visualization application, 2026.

[3] anonymous. Packalgo visualization (anonymized), an anthropic claude tetrahedron visualization application, 2026.

[4] anonymous. Packalgo visualization (anonymized), an openai chatgpt tetrahedron visualization application, 2026.

[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma andHenryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[6] Nikolaos Avouris, Kyriakos Sgarbas, George Caridakis, and Christos Sintoris. Teaching introduction to programming in the times of ai: A case study of a course re-design, 2025.

[7] Shraddha Barke, Michael B. James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7:85 – 111, 2023.

[8] Enna Basic and Alberto Giaretta. From vulnerabilities to remediation: A systematic literature review of llms in code security, 2026.

[9] Santiago Berrezueta Guzman and Stephan Krusche. Recommendations to create programming exercises to overcome chatgpt. In *2023 IEEE 35th International Conference on Software Engineering Education and Training*, pages 147–151, 08 2023.

[10] Hong Chen. Synergybug: A deep learning approach to autonomous debugging and code remediation. *Scientific Reports*, 15(1):2045–2322, 2025.

[11] Alison Clear and Amruth Kumar. The development of a 'living curriculum' for all computing. In *Proceedings of the ACM Global on Computing Education Conference 2025 Vol 2*, page 388–389, New York, NY, USA, 2025. Association for Computing Machinery.

[12] Jonathan Cordeiro, Shayan Noei, and Ying Zou. An empirical study on the code refactoring capability of large language models. *ACM Trans. Softw. Eng. Methodol.*, 2026.

[13] Debby Cotton, Peter Cotton, and Reuben Shipway. Chatting and cheating. ensuring academic integrity in the era of chatgpt, 2023.

[14] Kaléu Delphino. Assessing the prevalence of ai-assisted cheating in programming courses: A pilot study, 2025.

[15] Ruiqi Deng, Maoli Jiang, Xinlu Yu, Yuyan Lu, and Shasha Liu. Does chatgpt enhance student learning? a systematic review and meta-analysis of experimental studies. *Computers & Education*, 227:105224, 2025.

[16] Mark Chen et al. Evaluating large language models trained on code, 2021.

[17] Terry Yue Zhuo et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions, 2025.

[18] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A. Becker. My ai wants to know if this will be on the exam: Testing openai's codex on cs2 programming exercises. In *Proceedings of the 25th Australasian Computing Education Conference*, page 97–104, New York, NY, USA, 2023. Association for Computing Machinery.

[19] Mohsen Ghorbian, Mostafa Ghobaei-Arani, and Ali Shakarami. Large language models for code generation: A survey. *Computer Standards & Interfaces*, 98:104–165, 2026.

[20] Burak Gülmez. Code generation with large language models: a survey from neural program synthesis to autonomous software development. *Applied Intelligence*, 56(200):1573–7497, 2026.

[21] Hacer Güner and Erkan Er. Ai in the classroom: Exploring students' interaction with chatgpt in programming learning. *Education and Information Technologies*, 30:12681–12707, 2025.

[22] Baskhad Idrisov and Tim Schlippe. Program code generation with generative ais. *Algorithms*, 17(2):62, 2024.

[23] Maggie Johnson. Generative ai and cs education. *Communications of the ACM*, 67(4):23–24, 2024.

[24] Martin Jonsson, Jakob Tholander, Mattias Rost, Advait Sarkar, Ylva Fernaeus, Kharrufa Ahmed, and Thomas Ludwig. The end of programming as we know it. In *AAR Adjunct '25: Adjunct Proceedings of the Sixth Decennial Aarhus Conference: Computing X Crisis*, volume 30, 2025.

[25] Gregor Jošt, Viktor Taneski, and Sašo Karakatič. The impact of large language models on programming education and student learning outcomes. *Applied Sciences*, 14:4115, 2024.

[26] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. Explainable automated debugging via large language model-driven scientific debugging. *Empirical Software Engineering*, 30:1573–7616, 2024.

[27] Oscar Karnalim, Hapnes Toba, Meliana Christianti Johan, Erico Darmawan Handoyo andPlagiarism and ai assistance misuse in web programming: Unfair benefits and characteristics, 2023.

[28] Ruhma Khan, Sumit Gulwani, Vu Le, Arjun Radhakrishna, Ashish Tiwari, and Gust Verbruggen. Llm-guided compositional program synthesis, 2025.

[29] Raphaël Khoury, Anderson R. Avila, Jacob Brunelle, and Baba Mamadou Camara. How secure is code generated by chatgpt?, 2026.

[30] David Y. J. Kim. Redefining computer science education: Code-centric to natural language programming with ai-based no-code platforms, 2023.

[31] Dan Kohen-Vacs, Maya Usher, and Marc Jansen. Integrating generative ai into programming education: Student perceptions and the challenge of correcting ai errors. *International Journal of Artificial Intelligence in Education*, 35:3166–3184, 2025.

[32] Ludek Kucera, Ildar Akhmetov, Cruz Izu, and Olufisayo Omojokun. Adapting computing curricula in the era of automated software development. In *Proceedings of the ACM Global Computing Education Conference 2025 - Volume 2*, page 353–355, New York, NY, USA, 2025. Association for Computing Machinery.

[33] Amruth N. Kumar. Acm/ieee/aaai computer science curricula 2023, 2024.

[34] Oka Kurniawan, Erick Chandra, Christopher M. Poskitt, Yannic Noller, Kenny Tsu Wei Choo, and Cyrille Jegourel. Designing for novice debuggers: A pilot study on an ai-assisted debugging tool. In *Proceedings of the 25th Koli Calling International Conference on Computing Education Research*, page 1–7. ACM, 2025.

[35] Hao-Ping (Hank) Lee, Advait Sarkar, Lev Tankelevitch, Ian Drosos, Sean Rintel, Richard Banks, and Nicholas Wilson. The impact of generative ai on critical thinking: Self-reported reductions in cognitive effort and confidence effects from a survey of knowledge workers. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2025. Association for Computing Machinery.

[36] Victor R. Lee, Denise Pope, Sarah Miles, and Rosalía C. Zárate. Cheating in the age of generative ai: A high school survey study of cheating behaviors before and after the release of chatgpt. *Computers and Education: Artificial Intelligence*, 7:100253, 2024.

[37] Marina Lepp and Joosep Kaimre. Does generative ai help in learning programming: Students' perceptions, reported use and relation to performance. *Computers in Human Behavior Reports*, 18:100642, 2025.

[38] Kyla H. Levin, Nicolas van Kempen, Emery D. Berger, and Stephen N. Freund. Chatdbg: Augmenting debugging with large language models. *Proceedings of the ACM on Software Engineering*, 2:1892–1913, 2025.

[39] Jingjing Liu, Zeming Liu, Zihao Cheng, Mengliang He, Xiaoming Shi, Yuhang Guo, Xiangrong Zhu, Yuanfang Guo, Yunhong Wang, and Haifeng Wang. Repodebug: Repository-level multi-task and multi-language debugging evaluation of large language models, 2025.

[40] T. L. Sancha M. E. Newell, R. G. Newell. A solution to the hidden surface problem. In *ACM '72: Proceedings of the ACM annual conference*, volume 1, pages 443 – 450, 1972.

[41] Ivica Pesovski, Daniela Vorkel, and Vladimir Trajkovik. Ai-powered education: Rethinking the way programming is taught using ai tools and reversed bloom's taxonomy. In *21st International Conference on Information Technology Based Higher Education and Training*, pages 1–6, 11 2024.

[42] James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Randrianasolo, Brett A. Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. The widening gap: The benefits and harms of generative ai for novice programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1*, page 469–486, New York, NY, USA, 2024. Association for Computing Machinery.

[43] Christian Rahe and Walid Maalej. How do programming students use generative ai? *Proceedings of the ACM on Software Engineering*, 2(FSE):978–1000, 2025.

[44] Antoine Saint Éxupéry. *Terre des hommes*, chapter III - L'avion. Gallimard, Paris, 1939.

[45] Antoine Saint Éxupéry. *Wind, Sand and Stars (English translation of Terre des hommes by Lewis Galantière)*. Reynal and Hitchcock, New York, 1939.

[46] Jaromir Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. Thrilled by your progress! large language models (gpt-4) no longer struggle to pass assessments in higher education programming courses. In *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1*, page 78–92. ACM, 2023.

[47] JXin Sun, Daniel Ståhl, Kristian Sandahl, and Christoph Kessler. Quality assurance of llm-generated code: Addressing non-functional quality characteristics, 2026.

[48] Wannita Takerngsaksiri, Cleshan Warusavitarne, Christian Yaacoub, Matthew Hee Keng Hou, and Chakkrit Tantithamthavorn. Students' perspective on ai code completion: Benefits and challenges, 2024.

[49] Vinayak Venugopal, Rebecca Chen, Pranav Srikanth, Rutu Raj, and Mihai Boicu. Personalized debugging feedback: A comparative study of ai assistants for novice and expert programming tasks on codeforces. *Journal of Student Scientists' Research*, 7, 2025.

[50] Yuankai Xue, Hanlin Chen, Gina R. Bai, Robert Tairas, and Yu Huang. Does chatgpt help with introductory programming?an experiment of students using chatgpt in cs1. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, page 331–341, New York, NY, USA, 2024. Association for Computing Machinery.

[51] Meriem Zerkouk, Miloud Mihoubi, and Belkacem Chikhaoui. A comprehensive review of ai-based intelligent tutoring systems: Applications and challenges, 2025.

[52] Wenqing Zheng, S P Sharan, Ajay Kumar Jaiswal, Kevin Wang, Yihan Xi, Dejia Xu, and Zhangyang Wang. Outline, then details: Syntactically guided coarse-to-fine code generation, 2023.