Jonathas Castilho

Agile as Rule-Governed Behavior: A Radical Behaviorist Examination of Software Development Practices in AR environment

São Paulo

20 de janeiro de 2026

Jonathas Castilho

Agile as Rule-Governed Behavior: A Radical Behaviorist Examination of Software Development Practices in AR environment

Work submitted to the Computer Science Program at the University of São Paulo, as a partial requirement for obtaining the Bachelor of Computer Science degree.

Universidade de São Paulo

Instituto de Matemática, Estatística e Computação

Bacharelado em Ciência da Computação

Advisor Professor Dr. Eduardo Colli

São Paulo

20 de janeiro de 2026

Abstract

The development of augmented reality applications is characterized by high variability, technical constraints, and frequent environmental change. In the absence of an explicit development process, such variability tends to generate disorganized practices that fail to produce software increments with observable value for the client. This work describes the advancements on the case study MatematecAR and examines agile software development as a set of rules interpreted by Skinner's behavioral analysis psychology.

Keywords: agile; software development; punishment; radical behaviorism; augmented reality.

Sumário

1 Introduction

Matemateca is an interactive exhibition that presents mathematical ideas through direct manipulation of physical objects. As shown in Figure1, visitors come to the Institute of Mathematics, Statistics and Computer Science to explore this environment, but they are often confronted with many possible interactions at once. For newcomers, this abundance of possibilities can be confusing and may hinder effective engagement with the exhibits.

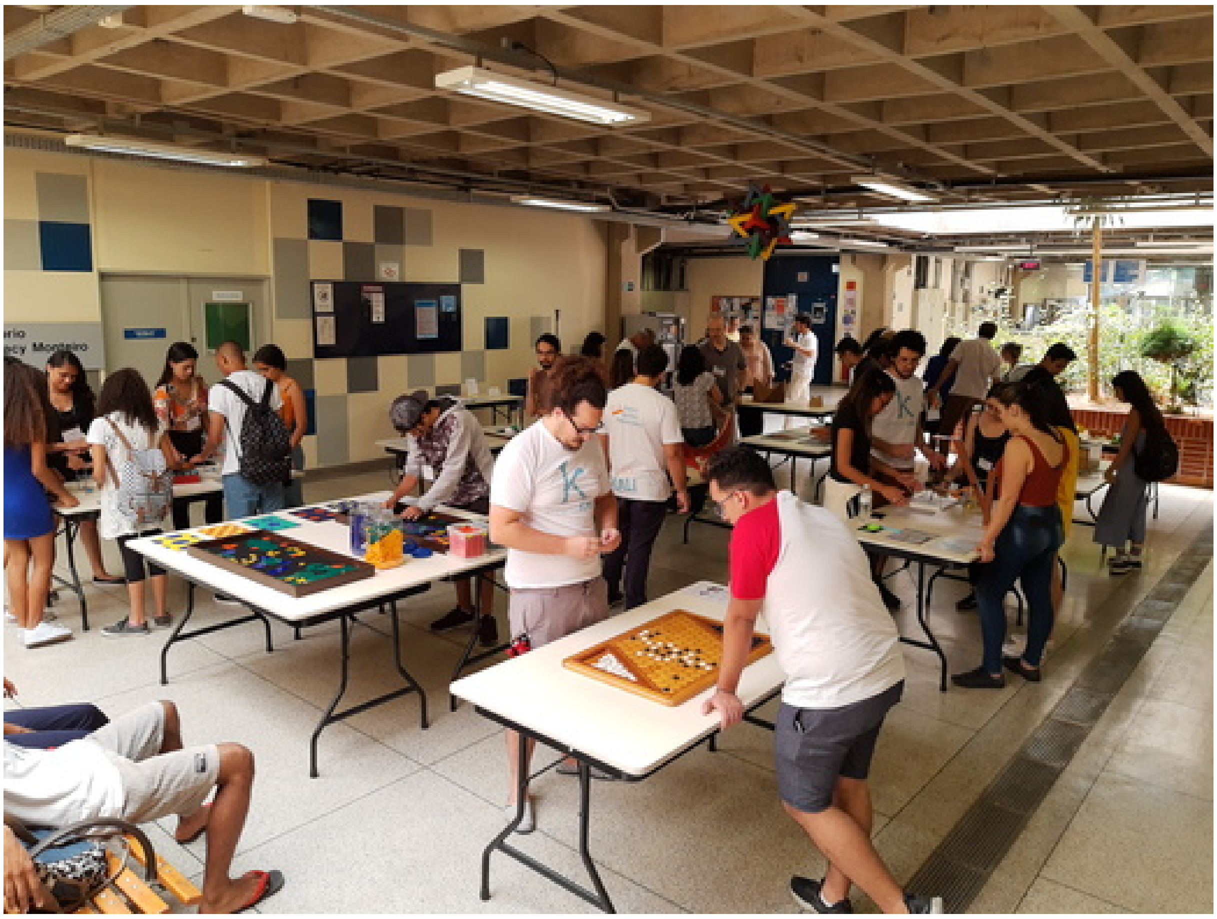


Figura 1 – View of the Matemateca exhibition at the Institute of Mathematics and Statistics (IME–USP), University of São Paulo, October 26, 2019. Source: <https://matemateca.ime.usp.br/expo.html>

To address this issue, Professor Eduardo Colli initiated the development of an augmented reality application called MatematecAR. It guides visitors by visually demonstrating how to interact with the exhibition's real objects in a tutorial-like manner. In Figure 2, the app shows how to set up the Königsberg bridges as the original problem proposed by Euler. Then, users are encourage to try to find a path using that settings.

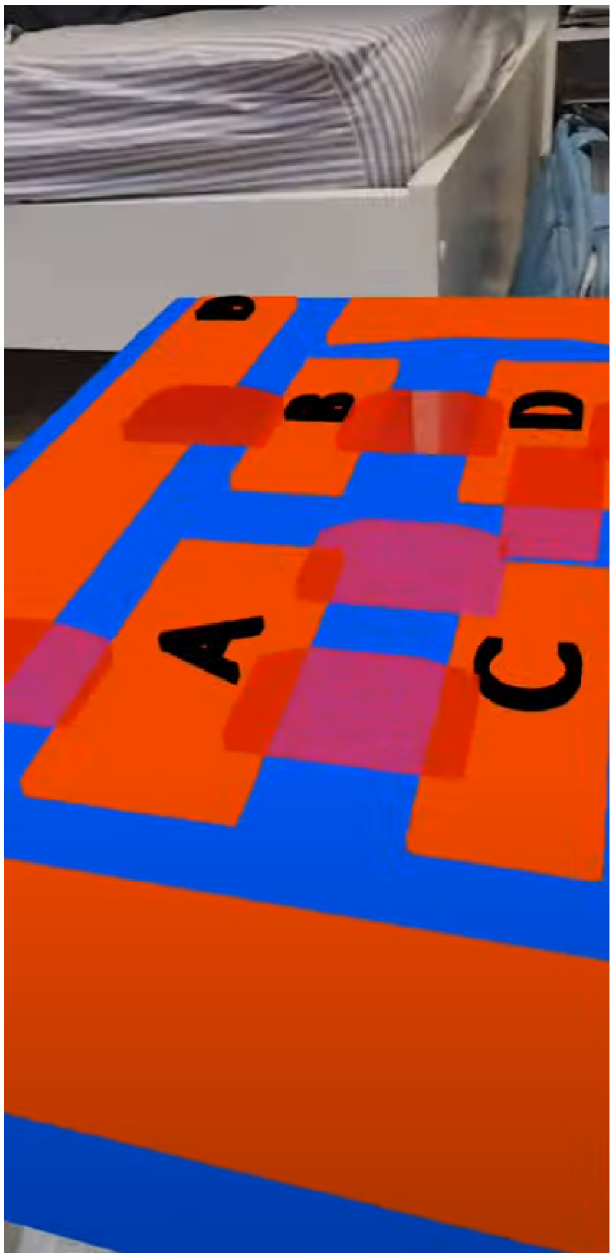


Figura 2 – One of MatematecAR scenes. The experience reconstructs the original Seven Bridges of Königsberg problem in 3D. Visitors are encouraged to navigate the city and attempt to cross each of the seven bridges exactly once. Through this interaction, the app reveals an inherently unsolvable configuration.

A secondary objective was to improve the explanation of the mathematical concepts underlying each piece. However, before these explanations could be effectively delivered, the software needed to accurately align virtual 3D models with the corresponding physical objects. This alignment task—known in augmented reality systems as the registration problem (Azuma 1997)—proved to be more complex than initially anticipated and became the primary technical challenge of the project.

Early development of the application proceeded without stable engineering practices, which led to limited and inconsistent progress. Architectural decisions accumulated without regular validation, and feedback cycles were long, making it difficult to identify which approaches were effective. As a result, the project reached a point where further development under the existing architecture became increasingly inefficient.

This work therefore focuses on the transition from a monolithic development approach to the adoption of agile methodologies. It documents the technical and organizational challenges involved in implementing the application while simultaneously establishing an agile development environment. It finishes by making a discussion on the principles and how to effectively apply the them on any project. Rather than treating agile as rigid rules to be followed, they are interpreted as shared practices that shape how the team works, communicates, and adapts. Through this lens, agility emerges not as a prescribed formula for success, but as a set of conditions that support continuous learning, coordination, and improvement throughout the development process.

2 Background

Before starting any development, the team must understand the domain of the problem they are trying to solve. This chapter is dedicated to present all information gathered to contextualize the literature and limitations of Augmented Reality. All that information was used to conduct the decisions on the project and they serve as arguments for the those choices throughout the development history of the project.

2.1 Mixed Reality (MR)

Augmented Reality (AR) can be understood as a specific instance of what Fred Brooks termed *Intelligence Amplification* (IA): the use of computational systems as tools that reduce the effort required for humans to perform tasks. As Azuma notes, AR systems do not replace human activity but extend it by providing contextual computational support (Azuma 1997). Despite their potential for engagement and entertainment, AR applications present substantial technical challenges.

According to Milgram's taxonomy (Milgram 1994), applications that combine real and virtual elements belong to a *virtuality continuum*. At one extreme lies a completely real environment; at the other, a fully virtual one. The degree to which the system must model the world in order to maintain the experience determines its position along this continuum. Virtual Reality (VR) applications lie closer to the virtual extreme, while Augmented Reality ones lie closer to the real-world end.

In virtual reality systems, the observer is placed within a fully simulated scene. Under these conditions, developers maintain total control over spatial relations, interaction rules, and physical dynamics. The system has complete access to its own state, allowing predictable and consistent responses. To sustain the illusion of realism, the system must emit responses analogous to those expected from real-world physics. For example, a virtual plate placed on a table must be graspable, movable, and subject to gravity when released. If dropped from sufficient height, the plate should fracture according to the simulated force applied. These responses are not derived from the real world but modeled in the system.

Augmented reality applications operate under fundamentally different constraints. Rather than simulating the entire environment, AR systems must continuously sample the real world through sensors and adjust virtual content accordingly. The system must update object position, scale, lighting, and occlusion in real time to maintain coherence between virtual and physical elements. Here, the illusion depends on precise tracking and rapid feedback: virtual objects must respond appropriately to changes in the physical environment, including movement, obstruction, and illumination.

2.2 Real world applications

Although current AR systems fall short of full immersion, they have been applied across multiple domains. The level of immersion achievable depends mainly on the available hardware capabilities. Smartphones, for instance, rely on relatively low-precision sensors, which constrains spatial accuracy and limits the reliability of environmental data, making it difficult to maintain registration. As a result, developers must carefully evaluate whether accessible technology can support the requirements of a project.

Contemporary AR applications have been deployed in areas such as:

- **Repair and maintenance:** architecture realtime visualization of building projects ; internal mapping of building, showing wires and pipes. Industry building lines, internal structure of machines, etc; (Cuperschmid, Ruschel e Freitas 2012)
- **Telepresence:** projecting robots and simulating its path during missions; projecting your body to a meeting in another place far away from the office (Azuma 1997);
- **Medical visualization:** simulation of surgical procedures; using sensors like Magnetic Resonance Imaging (MRI); ComputedTomography scans (CT); or ultrasound imaging. It is possible to have a 3D model of the insides of a patient, combining the data with their real body giving doctors a live “X-ray” (Azuma 1997).

- **Education:** interactive books or projecting 3D models of animals, molecules in the educational space; simulation of high risk or high coast training (Azuma 1997);
- **Entertainment:** games, such as Pokemon GO; playstation's dual sense controller, which gives tactical feedback according to the game's context and increases the force needed to press the buttons. (DualSense Wireless Controller)
- **Sound equipment:** noise cancelling headphones, which block outside sounds by adding (augmenting) the ambient sound with the exact frequency needed to superimpose waves and cancel outside noise sounds.

Despite these applications, the literature suggests that AR technology has not yet reached a level of maturity at which it consistently provides clear advantages over traditional interfaces. In many cases, the technical complexity and development cost outweigh the practical benefits delivered to users. As a result, AR is often unsuitable for low-budget projects and may offer limited return on investment outside specialized contexts.

2.3 Presence

Presence is a fundamental concept in mixed–reality applications. Milgram's continuum (Milgram 1994) highlights *Real Time Imaging* as the maximal point of presence, a condition in which users experience no observable distinction between computer–generated imagery and the real environment. According to his formulation, such an effect would be approached through the use of non–mediated Head Mounted Displays (HMDs). In practice, this is nearly impossible, but reducing reliance on resynthesized or sampled imagery tends to bring the virtual and natural environments closer together.

Technically, according to Azuma (Azuma 1997), there are some events which can reduce the sense of presence:

- Optical distortion: cellphone cameras often present radial and tangential distortions due to compact lens assemblies. These distortions reduce the accuracy of feature detection and require calibration procedures and correction models based on mappings (Rolland, Hua e Gao 1994).

- Errors in the tracking system: tracking based on feature extraction and sensor fusion is susceptible to drift, noise, and unstable feature sets. In mobile devices, these errors increase because illumination changes, motion blur, and rapid device movement degrade the stability of the point sets used for pose estimation (Schubert, Qin e Cremers 2021, Billinghurst, Clark e Lee 2015).

- Mechanical misalignments: small displacements among the camera, inertial sensors, and display create fixed offsets that require compensation. These offsets behave as fixed transformations under compositions in the form $T = R \circ S$, and inaccurate estimation leads to cumulative positioning deviations (Rolland, Hua e Gao 1994).

- Incorrect viewing parameters: mobile displays differ in pixel density, field of view, and aspect ratio, generating mismatches between the projection model and the physical display. Without proper calibration, the projection mapping $P : R^3 \to R^2$ deviates from the intended geometric relation (Azuma 1997).

- Dynamic errors caused by system delays: delays arise from sensor sampling intervals, frame processing, and rendering operations. These delays generate systematic lags in the temporal sequence, altering the correspondence between the organism's movement and the device's produced visual consequences (Steed 2016).

Thereafter, he suggests improvements on:

- Reducing system lag: decreasing the end-to-end latency requires optimized pipelines, hardware acceleration, and reduced computational overhead during feature processing and rendering (Steed 2016).

- Reducing apparent lag: apparent lag can be mitigated using techniques such as late-stage reprojection or pose prediction, which adjust the rendered frame immediately before display refresh (Steed 2016).

- Matching temporal streams (with video-based systems): sensor streams must be synchronized so that inertial readings, camera frames, and rendering operations correspond to the same temporal interval. Without synchrony, the composed output represents mismatched events (Wen, Schwerdtfeger e Klinker 2013).
- Predicting future locations: prediction techniques estimate future device poses using motion models defined on sets of past observations. These predictions reduce perceived delay by aligning rendered output with the organism's upcoming movement pattern (Azuma 1997, Steed 2016).
- Using depth: Finally, occlusion is the strongest depth clue (Azuma 1997), but most mobile devices do not have support for algorithms, api or sensors which would allow applications to create this effect.

All those points would be easier to solve using proper technology focused on mitigating the registration problem in AR. However, phones are not optimal for that and could not be solved easily in the case study.

The software developed and analyzed in this work applies augmented reality to an educational context using smartphones as the primary platform. The author concentrates its efforts on maximizing the sense of presence within the constraints imposed by mobile devices.

As a result, design and implementation decisions prioritize performance, tracking stability, and perceptual coherence over visual complexity. The goal is not to showcase advanced graphics, but to ensure that virtual elements remain consistently aligned with physical objects and respond smoothly to user interaction.

3 Challenges

As mentioned previously, augmented reality has advanced in parallel with modern computing technologies; nevertheless, the registration problem remains one of its most complex and persistent challenges.

Registration refers to the system's ability to correctly align and maintain virtual elements in relation to the physical world. This difficulty arises primarily because accurate registration depends on factors that are only partially under the developer's control. These include the quality and calibration of client-side equipment, the precision and stability of available sensors, and the system's capacity to transfer and process large volumes of data with minimal latency. Small deviations in sensor readings or delays in data processing can result in perceptible misalignment, which degrades user experience and compromises the intended interaction.

From the development perspective, additional constraints are introduced by the need for specialized environments capable of supporting computationally demanding workflows. High-performance hardware, advanced tracking libraries, and extensive testing across heterogeneous devices are often required to achieve acceptable registration accuracy. These requirements increase development cost and complexity, further limiting the feasibility of AR solutions in contexts where resources are constrained.

MatematecAR were not an exception. It faced several difficulties during its development. The application needed to operate on Android smartphones, whose hardware specifications are highly heterogeneous. This variability made it difficult to predict which features could be safely implemented without compromising compatibility and accessibility for visitors of Matemateca. Decisions regarding performance, tracking accuracy, graphical fidelity, and sensor usage had to account for a wide range of devices, from low-end models to more capable smartphones.

These challenges are describe in details below.

3.1 Registration

Initially, the system depended exclusively on image matching and tracking to establish the model's registration in the environment. By locating and maintaining an anchor in physical space, the application could render the 3D objects in the scene. However, occlusion events or abrupt movements weakened the sense of presence by disrupting registration. A single strategy proved insufficient, and the solution required combining two techniques: image tracking and plane detection. The image established the anchor position, while the plane provided stability for the virtual object once placed.

Presently accessible technologies remain imperfect in guaranteeing a consistent superposition between real and virtual elements. Even high–end systems fail to generate a fully continuous illusion. When designing an augmented reality solution, minimizing registration and sensing errors becomes essential (Azuma 1997). In this project, the application targeted Android devices, which differ widely in sensor configurations, precision, processing power, and data–handling capacity. These variations generate unreliable knowledge of the world (KOW) (Milgram 1994), weakening the degree of presence attainable in the XR environment.

3.2 Hardware

Hardware diversity also influences performance: differences in cameras, GPUs, and CPUs among mobile devices limits the rendering pipeline. It would be good to have sensors which alloweded depth effects and real-time filters to make the experience more realistic. However, most graphical systems rely on the pinhole model, which renders every element in uniform focus. Any advanced effect must be applied as a post-render detail, which is computational consuming. Then, the field of view for real-time rendering is fixed. This disrupts the immersion.

3.3 Client support

Compatibility for final users also acted as a restrictive variable. Expecting that visitors will have the best phones would exclude many ones from the experience developed. Sponsoring a complete set of Head Mounted Displays (HMDs) for all visitors was also infeasible, and widespread public access to such devices remains limited. These factors selected a different course of action for the project, which culminated in the adoption of smartphones as the primary interaction tool. Their broad availability enables consistent delivery of software to a diverse audience.

Still, they introduce constraints related to processing capabilities, sensor precision, and data capture, as largely discussed above.

3.4 Missing assets

The project was developed mainly by non-artistical students. Then, asset creation slowed the development process. Then, there was the need to manually model them. This was a bottleneck in the production process, because a lot of time was spent measuring and modeling each scene.

Lack of precision during modeling was also problematic. Superposition won't be possible if it is not precise. It would be much easier to use scanners to recreate the physical objects, but the excessive number of polygons would lead to optimization problems, which is discussed below.

3.5 Optimization

Optimization in augmented reality is crucial for maintaining immersion. Rendering latency can prevent 3D content from being reliably superimposed onto the physical scene, and degraded performance disrupts the sense of presence. For this reason, optimization was treated as a central concern from the outset of development rather than as a later refinement.

Several design decisions followed from this constraint. Scene assets were limited to low-polygon models, and only a small number of 3D objects were rendered simultaneously. These restrictions reduced computational load, stabilized frame rates, and increased the reliability of spatial alignment across devices with heterogeneous hardware capabilities.

4 Methodology

This investigation combined the construction of a software artifact with a review of agile literature interpreted by behavioral analysis (Skinner 1969). It describes the evolutionary history of MatematecAR through user stories, emphasizing how agile principles function as shared guidelines that influence developer decision-making.

4.1 Procedures

User stories was written as registration of improvement. Each of them followed the traditional model of answering "who, what and why". When viewed through a behavior-analytic approach, this template resembles a contingency, that is, it specifies relations between an organism's responses and the environmental conditions that select them (Skinner 1969). It is important to mention the differences as well as to clarify that a user story intertwines two distinct contingencies: it organizes the client's sought-after reinforcement conditions while simultaneously arranging the developer's required responses, which in most cases are the implementation of the solution.

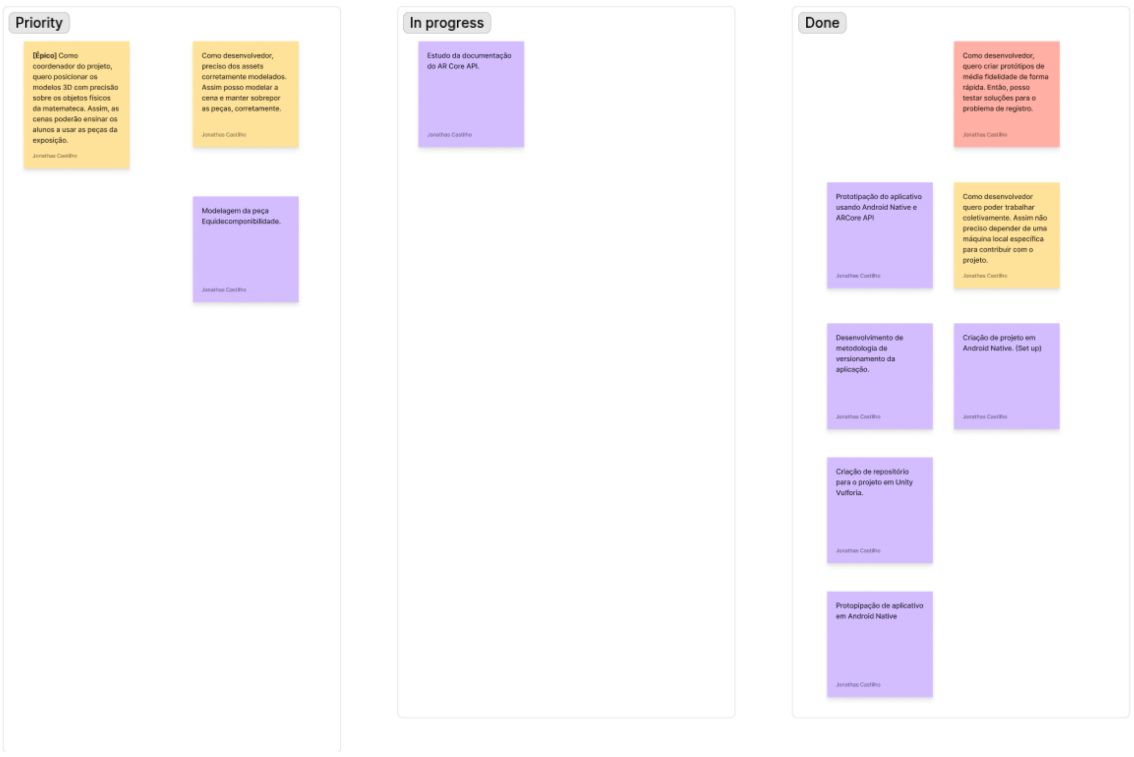
Evaluation of increments occurred with the participation of the real client, Professor Eduardo Colli. The client's validation acted as an external selective agent by providing immediate consequences for each software iteration (Skinner 1981).

4.2 Workflow

The team held weekly meetings, which functioned as regular occasions for evaluating the outcomes of the previous cycle and arranging the tasks for the next one.

The project adopted weekly sprints as the shortest interval for delivering increments. Each sprint began with the specification of observable objectives and ended with the delivery of verifiable results.

A simple Scrum board structured this workflow, as shown in Figure 3. It included three columns—Sprint Backlog, In Progress, and Done—allowing the continuous tracking of task movement.



Priority

[Épico] Como coordenador do projeto, quero posicionar os modelos 3D com precisão sobre os objetos físicos da matemateca. Assim, as cenas poderão ensinar os alunos a usar as peças da exposição.
Jonathas Castilho

Como desenvolvedor, preciso dos assets corretamente modelados. Assim posso modelar a cena e manter sobrepor as peças, corretamente.
Jonathas Castilho

Modelagem da peça Equidecomponibilidade.
Jonathas Castilho

In progress

Estudo da documentação do AR Core API.
Jonathas Castilho

Done

Como desenvolvedor, quero criar protótipos de média fidelidade de forma rápida. Então, posso testar soluções para o problema de registro.
Jonathas Castilho

Prototipação do aplicativo usando Android Native e ARCore API
Jonathas Castilho

Como desenvolvedor quero poder trabalhar coletivamente. Assim não preciso depender de uma máquina local específica para contribuir com o projeto.
Jonathas Castilho

Desenvolvimento de metodologia de versionamento da aplicação.
Jonathas Castilho

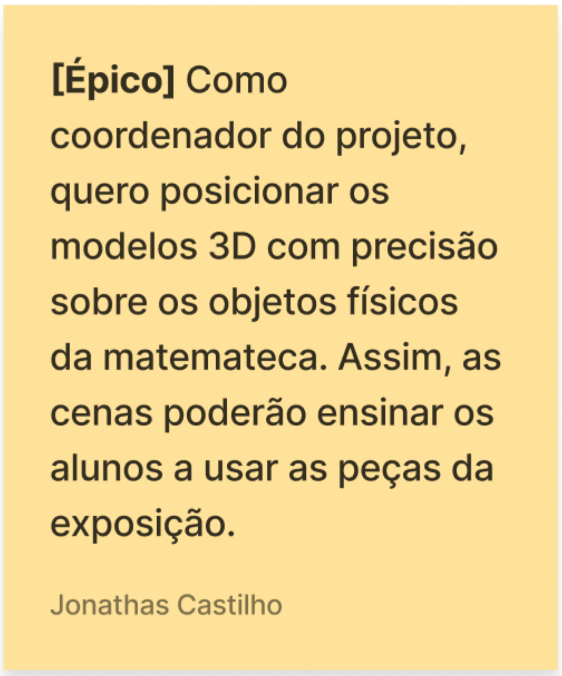
Criação de projeto em Android Native. (Set up)
Jonathas Castilho

Criação de repositório para o projeto em Unity Vulforia.
Jonathas Castilho

Protopipação de aplicativo em Android Native
Jonathas Castilho

Figura 3 – Scrum board used in the development process of MatematecAR. It containers three columns. Namely, sprint backlog (priority), in progress and done. Available at: <https://www.figma.com/board/TFvkX8LncqEaK2aDdQRW0f/Matemateca?node-id=0-1&t=lGfHv8kfytAjT72Z-1>

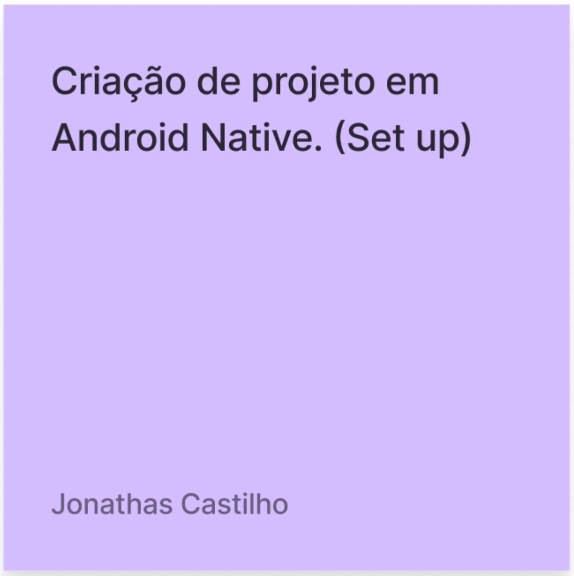
Each post-it represented a differnt aspect of development. Yellow cards are user stories (Figure 4); purple ones are atomic tasks related to the completion of a particular story (Figure 5); and red represents overdone ones (Figure 6).



[Épico] Como coordenador do projeto, quero posicionar os modelos 3D com precisão sobre os objetos físicos da matemateca. Assim, as cenas poderão ensinar os alunos a usar as peças da exposição.

Jonathas Castilho

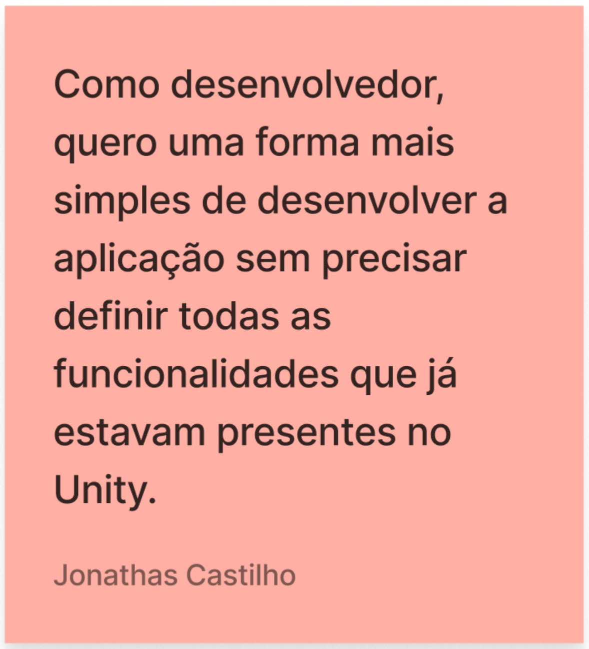
Figura 4 – Description of the main request from the client. It is marked as Epic, for it can derive other user stories.



Criação de projeto em Android Native. (Set up)

Jonathas Castilho

Figura 5 – Task of development of a particular user story.



Como desenvolvedor, quero uma forma mais simples de desenvolver a aplicação sem precisar definir todas as funcionalidades que já estavam presentes no Unity.

Jonathas Castilho

Figura 6 – Overdue user story.

The extensive use of assets, scenes, and other non-textual resources introduces significant obstacles when applying traditional version control methodologies. Unlike software projects in which most changes occur in text files, game-related assets are typically stored as binary data, making conflict detection and resolution impractical or impossible. As a consequence, the project adopted a centralized version control system (CVS), which better accommodates binary versioning at the cost of flexibility.

Classical software for smartphones can be tested by running automated tests. Aspects of a software that are costly to implements in automation can be built locally and manually tested. In the case study, developers needed to have physical objects from the exhibition to make sure the application was running as intended. To solve this, it was necessary to create a simple virtualy simulated scene and run the AR application on it. Unity provides this feature, allowing developers to model the environment, such that it is possible to test if everything is working properly without having to be in a specific place. Figure 7 shows the use of Unity's XR Environment to create a testable environment as mentioned.

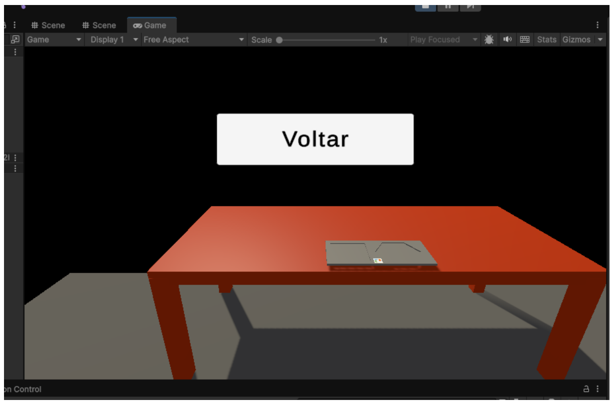


Figura 7 – Simulation of a Matemateca setting modeled in 3D. It uses Unity's XR Environments to run the application over the simulated environment.

4.3 Implementation Environment

Project wise, MatematecAR was developed using Unity's 6000.1.10f1. It follows a component-based architecture, using Scriptable Objects to manage game data and the MVC pattern to separate logic from UI. Figure 8 illustrate this architecture.

4.4 Client-side environment

The main constraint for the implementation was the use of mobiles phones, particularly Android OS. This is why operation system must be at least in version Android 7.0 Nougat (or iOS 11, for Apple's device) and meet the sensors requirements as follows: one camera sensor (two for more precision), accelerometer and gyroscope.

Although not a requirement, if the user desires to get the best experience, the phone should have ToF (time of flight sensor-common on top-line Android Phones), LiDAR (Light detection and ranging sensor-only available on Pro version of Apple's phones) and support Depth API.

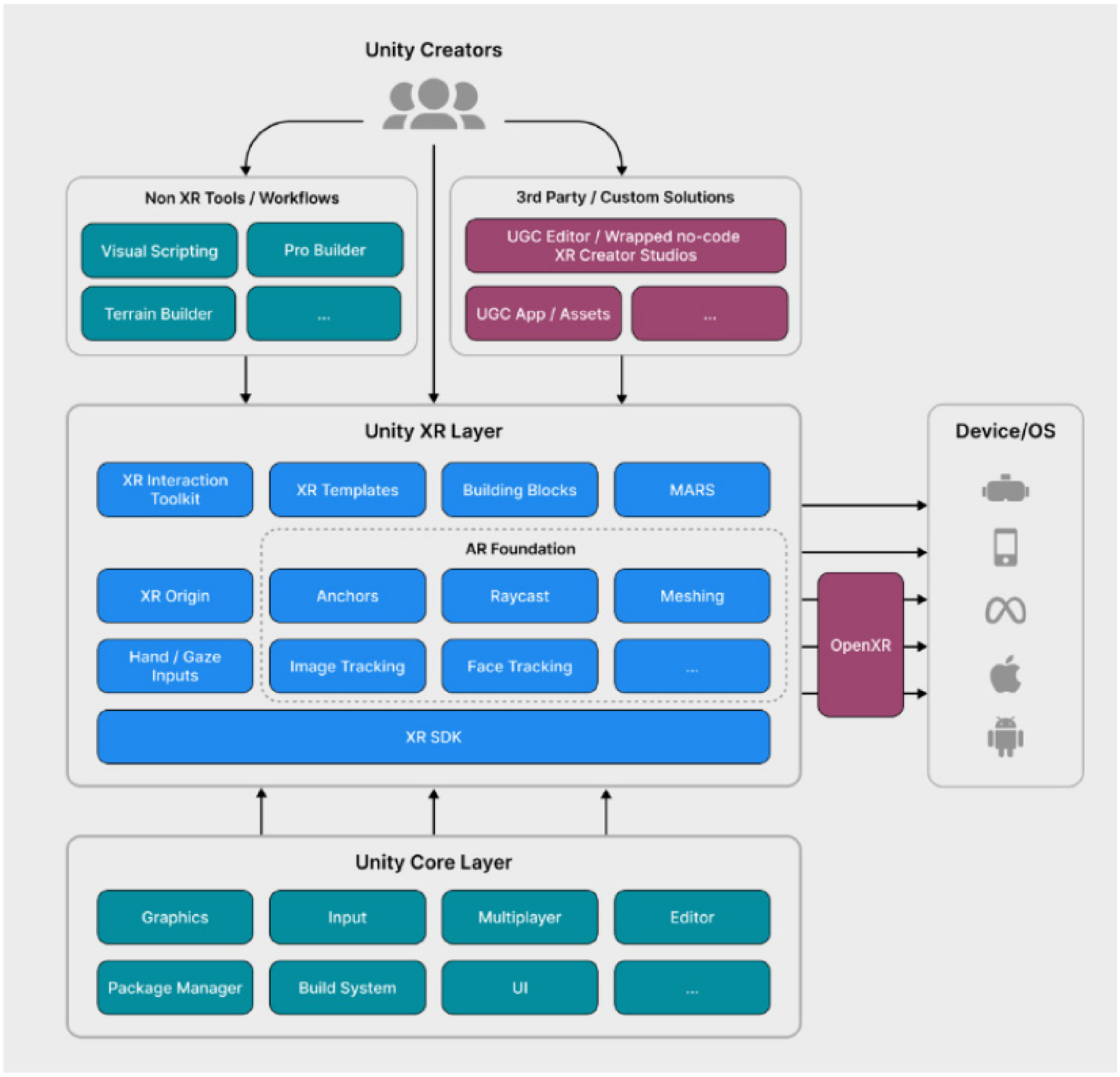


Figura 8 – Project archtecture based on Unity guidelines for XR Apps

5 Results

Initially, the project was approached as a game application. Within this domain, many software engineering practices rely on weak or inadequate development methodologies. According to Aleem (Aleem, Capretz e Ahmed 2016), the game industry commonly adopts approaches derived from the Waterfall model, in which developers prototype, implement, and test with little or no client involvement. This practice is often justified as a means of preventing spoilers and preserving the intended player experience by avoiding prior exposure to the game's narrative. However, MatematecAR should not have been treated under these assumptions.

After two years of development, the outcome was limited to a poorly implemented prototype. It consisted of a single scene for the Equidecomponibilidade board, featuring buttons labeled by color and corresponding modeled puzzle pieces. A physical board was marked with a QR code, and the application relied on camera-based image tracking to infer the spatial relation between virtual objects and the real artifact.

This implementation failed to meet the core requirement: accurately aligning virtual objects with physical artifacts in a way that effectively supported visitor interaction and learning. The objective was straightforward: to overlay 3D models onto Matemateca's physical objects so that visitors could readily discriminate how to interact with the exhibition. Achieving this required highly precise registration capable of maintaining stability across multiple simultaneous interactions and under the environmental variability typical of a museum setting.

The initial solution relied exclusively on QR-code-based tracking techniques. Its precision depended heavily on sensor quality and camera resolution, which varied widely across Android devices. Smartphones with lower-resolution cameras produced inconsistent results, and maintaining continuous marker visibility was not always feasible. Moreover, because exhibition artifacts were frequently manipulated by visitors, the markers could not be guaranteed to remain intact or retain sufficient visual fidelity over time.

The design of the initial scenes followed a classical imitation-based approach. Teaching through imitation represents one of the simplest procedures for shaping behavior (Skinner 1986).

From a development perspective, it is also one of the least costly strategies, as it avoids the need for complex scene orchestration or elaborate implementation. By presenting a modeled action that resembles the target interaction, the system establishes a discriminable relation between stimulus and response (Sidman 1971).

Demonstrating the use of a Matemateca artifact through a similar, though not identical, action can effectively evoke the appropriate response class in the visitor. As noted by Skinner (Skinner 1986), when an organism reproduces the behavior of another, the probability of contacting similar reinforcing consequences increases. Indeed, "organisms must have profited from the behavior of each other at a very early stage through imitation" (Skinner 1986). In this sense, imitation functions as an efficient mechanism for rapidly establishing functional interaction with the exhibited artifacts.

5.1 Project history

After the adoption of agile practices, several alternative solutions to the initial problem were proposed. One of the first attempts involved replacing the proprietary Vuforia–Unity integration with an Android native application developed using the ARCore API. Figure 9 shows the first version of that implementation.

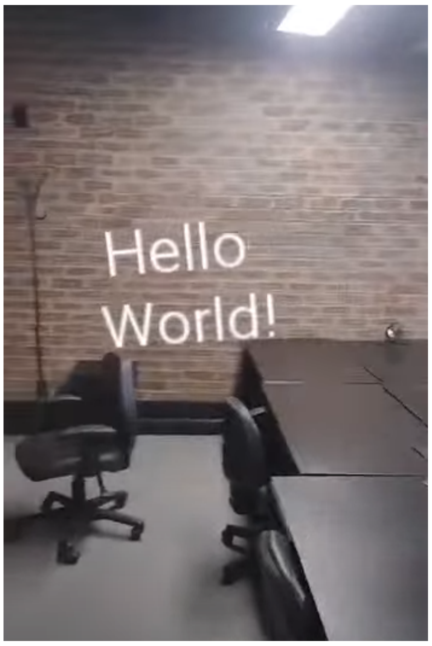


Figura 9 – Android native implementation of MatematecAR using Kotlin and ARCore API.

The outcome, however, was not satisfactory. Many functionalities previously handled by Unity required full re-implementation, including API communication, 3D projection handling, and pose-correction mechanisms. The development effort increased, and progress slowed.

To address this, a simpler framework was used. Using react and AR libraries from the community there were a simplification of the development. It offered a more accessible environment, faster iteration cycles, and access to a robust ecosystem of open-source tools. Figure 10 is the home page for the app developed using HTML and CSS technologies.

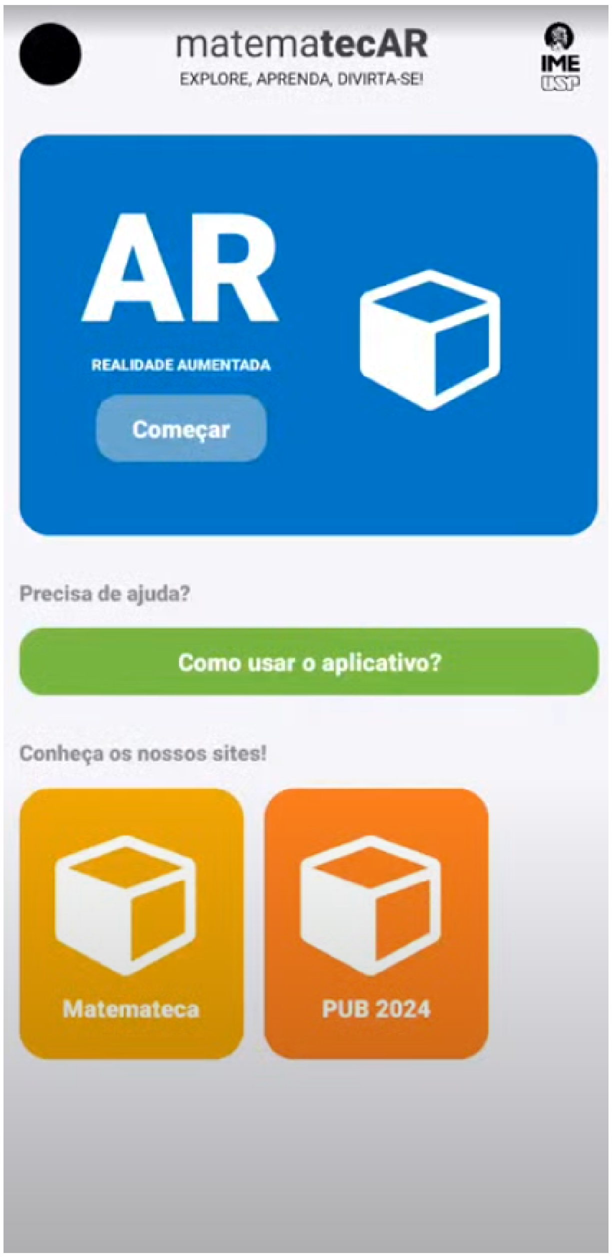


Figura 10 – User interface developed with the React Native framework. HTML-like tags and Tailwind-style utilities improved responsiveness and layouts.

Using this framework, early prototypes of the enhanced augmented-reality features were produced. The library Viro React enabled rapid integration of 3D content using plane detection and direct manipulation gestures. First scene developed on this technology is shown on Figure 11.

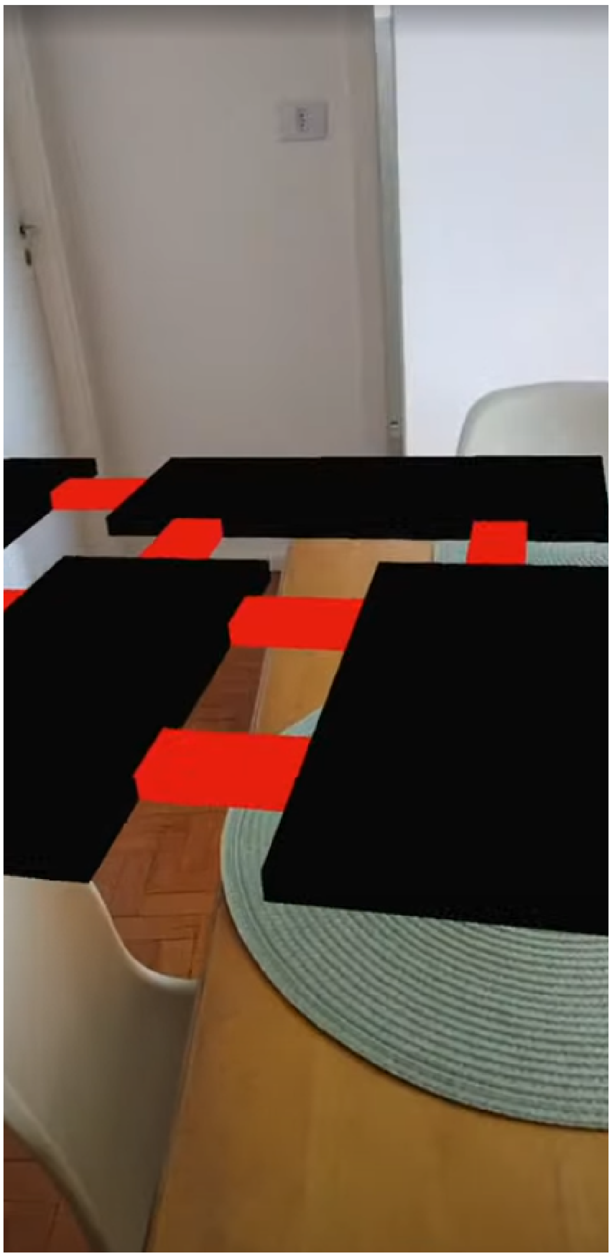


Figura 11 – Initial overlay prototype for the Königsberg Bridge model using Viro React. Plane detection positioned the 3D object, allowing users to drag the model across the detected surface.

Further iterations led to more refined models. Blender was used to create accurate 3D assets based on existing Matemateca phsyical objects, and animation libraries for React facilitated interactive implementations.

Beyond technical prototyping, agile processes encouraged the exploration of learning strategies before committing to software implementation. Low-fidelity experiments helped validate interactions and conceptual structures.

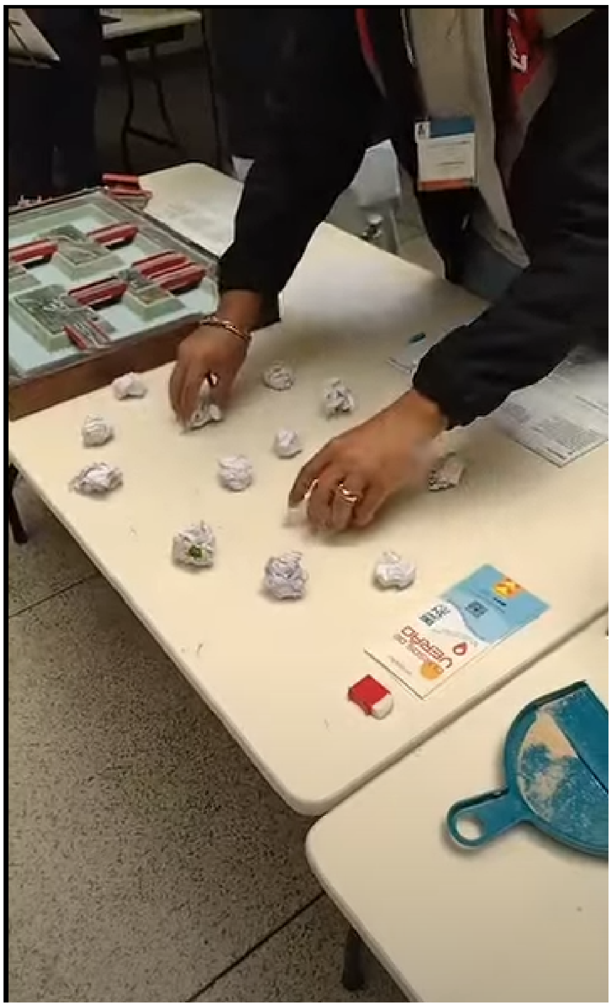


Figura 12 – Paper prototype illustrating graph reasoning being introduced to Matemateca's visitors. They connected nodes to determine shortest paths, later incorporating weighted edges to also determine the shortest path. It introduced them to ideas related to the Eulerian bridge problem.

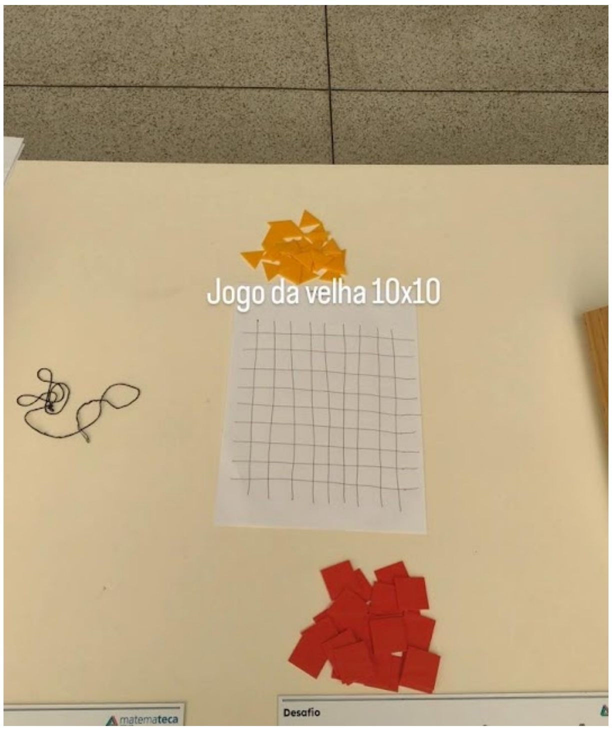


Figura 13 – Prototype illustrating the Gomoku game. The objective was to identified arrangements of five aligned positions, expanding the traditional 3x3 tic-tac-toe. Visitors contact this prototype to acknowledge variation of the game.

Finally, after many iterations, it was possible to achieve a great solution. Professor Colli emphasized the need for strong registration, which was achieved through a combined strategy of image tracking and plane detection. Together, these techniques ensured precise alignment of 3D elements in the scene, even under occlusion. Rapid camera movement and shifting visual context were also addressed by this combination.

By using fiducial marker and plane detection, constraints that initially appear insurmountable was addressed through a combination of design decisions and technical strategies. First, the application detects all the plains in the environment, as shown in Figure 14. It gives stability to continous maintainance of registration. After that, users only need to find the fiducial marker, which consists of a printed image glued onto the real object. For Equidecomponibilidade scene, it was used the image in Figure 15. Figure 16 shows how the marker is used to determine the origin of the scene.

In short terms, software alone could not ensure the level of precision required for release, especially under conditions of limited compatibility and inconsistent sensor performance. The

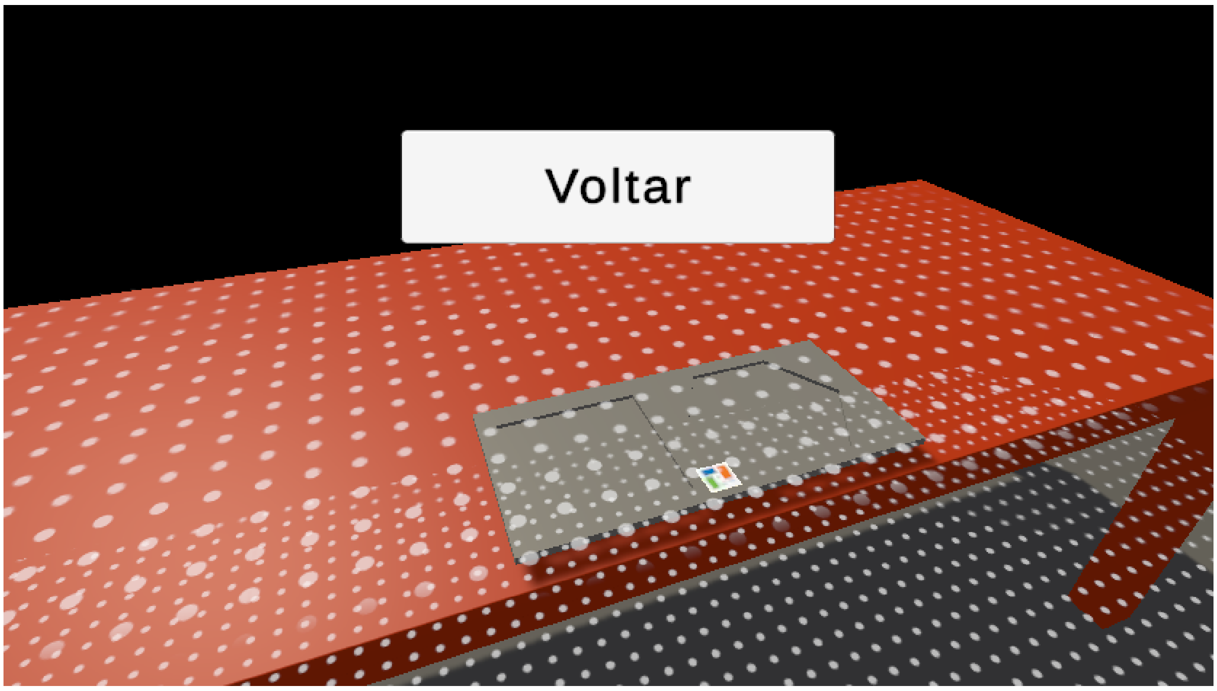


Figura 14 – Simluated AR environment. The white dots over the 3D modeled table and floor represent the mesh plane detected in real-time.

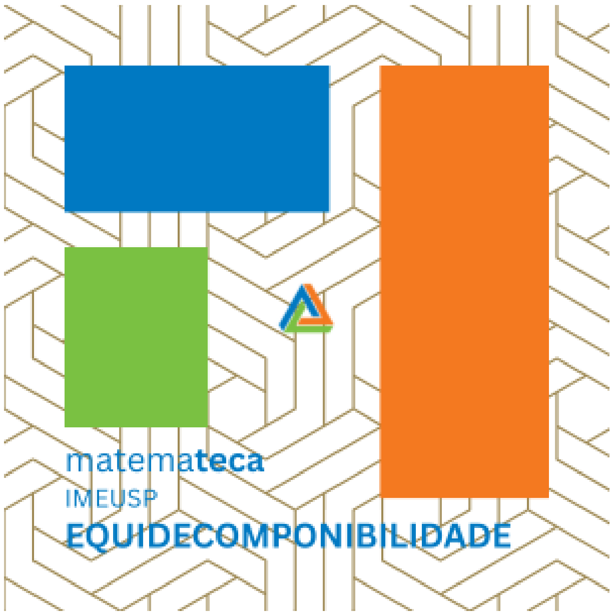


Figura 15 – Asymmetrical image with non-repetitive patterns. Those characteristics define an accurate fiducial marker to be used in AR.

solution emerged from balancing user-guided actions—such as detecting planes—with automated processes like image detection of fiducial markers. Although the goal was originally to track a physical object and place a 3D model directly on it, this hybrid sequence produced the precision necessary to meet the project's requirements. These adjustments illustrate how technical constraints shape the final workflow, often requiring the addition or reordering of steps to achieve

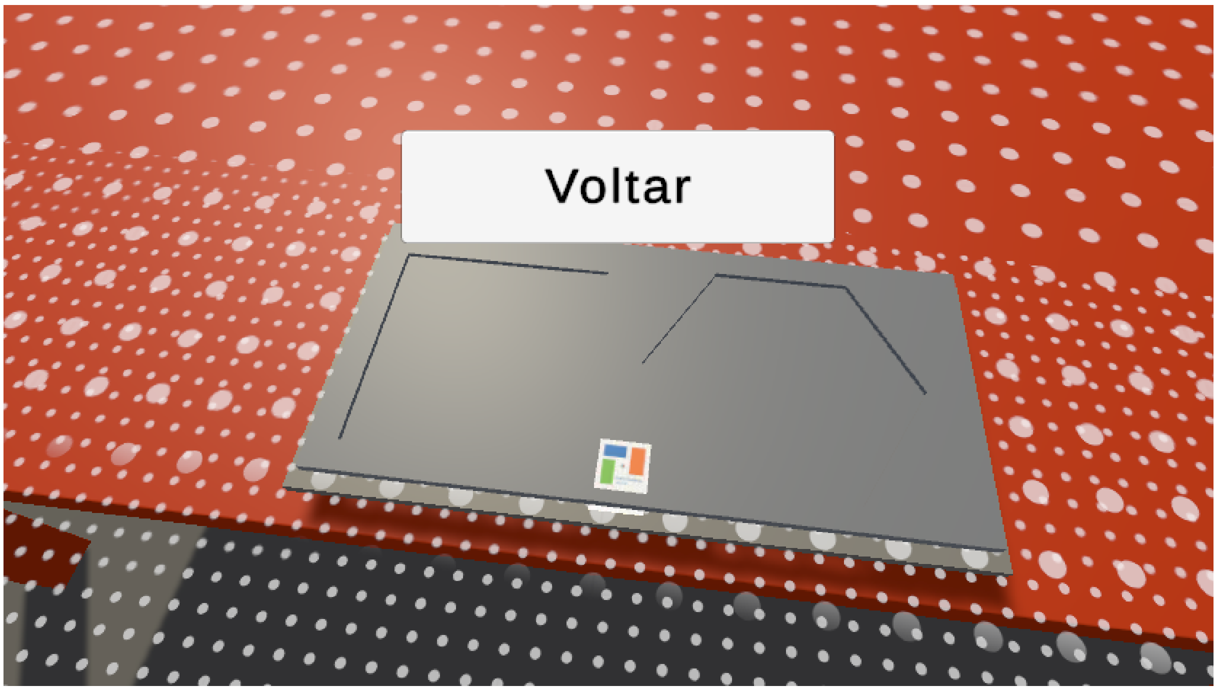


Figura 16 – Once the image is detected, using fiducial tracking tecniques, the virtual version of the Equidecomponibilidade board is place onto the simulated one.

the intended design outcome.

To demonstrate the increase of delivered value, agile metrics were collected. The burndown chart below (Figure 17) shows how work progressed throughout the sprints.

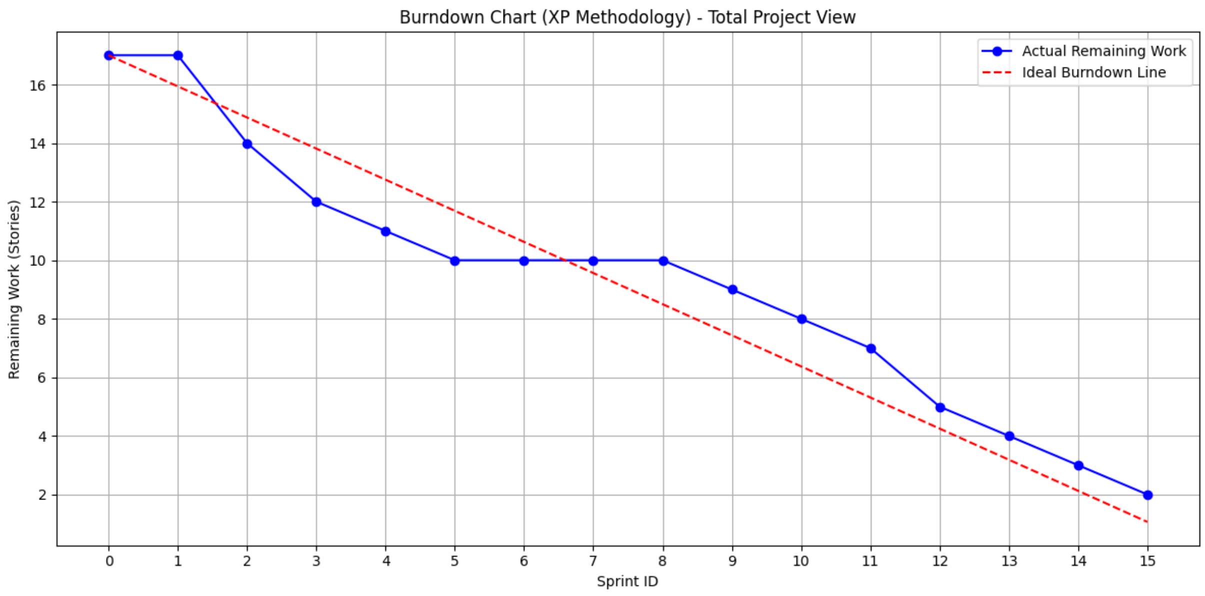


Figura 17 – Burndown chart displaying user-story completion over time. The red line shows the ideal speed of delivery for new features.

The cumulative flow diagram provides an additional view, illustrating the relationship between new stories and completed ones.

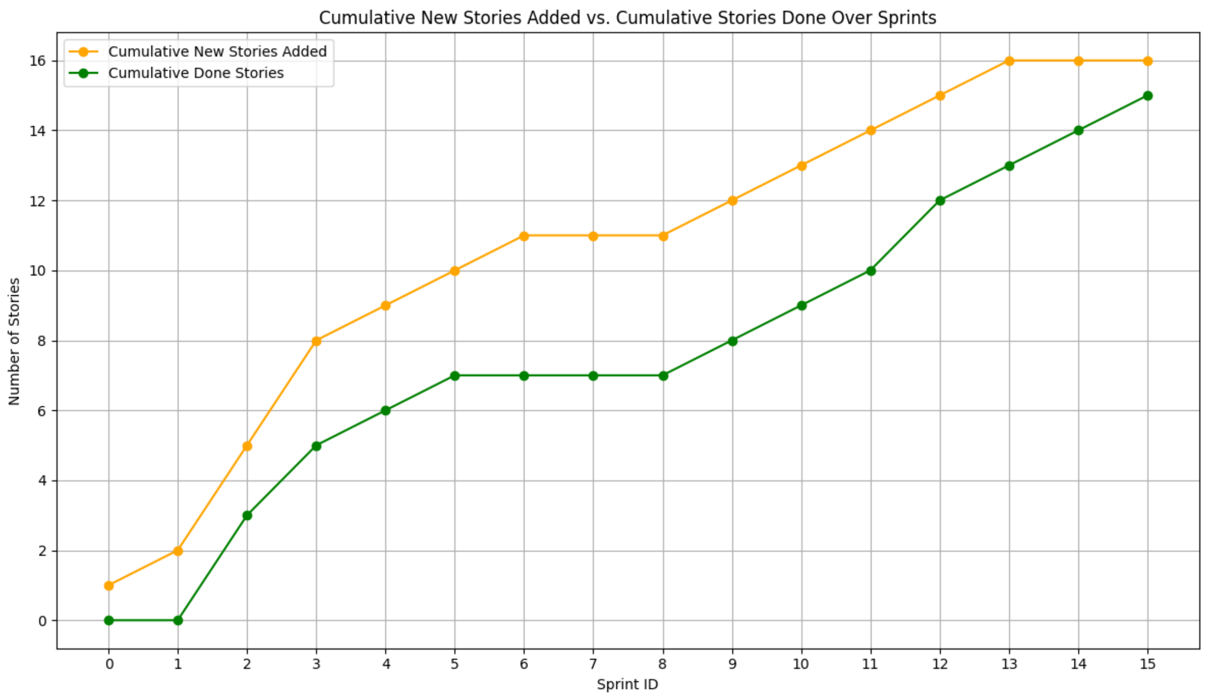


Figura 18 – Cumulative flow diagram showing the rate of story creation and completion over the project timeline. The small and stable distance between the yellow and green lines indicates that proposed increments were implemented shortly after being defined, suggesting that the agile process supported balanced planning and delivery.

6 Discussion

"Evolution favors those that operate with maximum exposure to environmental change and have optimised for flexible adaptation to change”, says Ken Schwaber (Schwaber). From this perspective, software development can be understood as a system continuously shaped by external constraints such as users, markets, and organizational demands. Developers are the primary agents in this system, and the continued usefulness of the software they produce depends on how well their decisions adapt to these changing conditions.

Agile methodologies such as Scrum, Extreme Programming, and Lean emerged as structured responses to this variability. While their practices and artifacts differ, they share a common set of principles. These principles are derived from observed regularities in successful software projects and prescribe procedures that tend to increase desirable outcomes, such as faster feedback, higher quality, and improved adaptability. In practice, they function as explicit guidelines that describe how certain actions are expected to lead to particular results.

From this perspective, agile practices resemble specifications of cause-and-effect relationships within a development system. For example, short iterations and continuous integration create tighter feedback loops, allowing teams to detect mismatches between expectations and outcomes earlier. This mirrors how adaptive systems improve performance by responding quickly to signals from their environment rather than relying on long-term predictions.

Despite the empirical evidence of success, adopting agile practices as written procedures does not automatically change how developers work on a daily basis. These practices describe how the system should behave, but they do not by themselves guarantee that the system will actually behave that way. This is the very definition of rule-governed behavior (Skinner 1969). Without concrete feedback mechanisms that clearly connect actions to outcomes, the practices remain abstract rules rather than operational drivers of change.

As a result, omitting critical elements—such as meaningful feedback, visible consequences of decisions, or opportunities to adjust behavior—prevents teams from achieving genuine adaptability. In such cases, agile processes may become rigid checklists that enforce compliance rather than support learning and improvement.

The following discussion examines agile principles in terms of how they may unintentionally discourage effective behavior and how such effects can be avoided. The analysis assumes a simple system principle: only feedback that improves future decisions leads to sustained improvement. Measures that merely suppress errors without teaching better responses do not increase long-term adaptability (Sidman 1989)-also known as punishments.

6.1 Agile manifesto

Technology companies typically organize their work around the effectiveness of the software development process. These organizations operate as complex systems driven primarily by economic outcomes. Financial return functions as the dominant success signal, shaping decisions at managerial and organizational levels. Managers and project owners are therefore responsible for designing working conditions so that teams can clearly perceive how their actions contribute to positive business results and to the commercial viability of the product.

Within this environment, each participant acts according to their own access to benefits such as compensation, recognition, autonomy, or career stability. As a result, day-to-day actions are not sustained by abstract ethical ideals but by how the system links individual decisions to tangible outcomes. In system terms, behavior is stabilized by the structure of incentives, constraints, and feedback loops embedded in the organization.

To increase the probability of favorable outcomes under these conditions, a group of developers articulated the Agile Manifesto. Agile approaches organize work through recurring coordination mechanisms—such as daily stand-ups, iteration planning, and retrospectives—and through technical procedures like code review and pair programming.

In the following analysis, each practice is examined in terms of how it shapes developer decision-making. Rather than invoking internal motivations or intentions, the discussion focuses on how different configurations of the development process encourage or discourage specific patterns of action within the system.

6.1.1 Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

When developers work on requirements without validating what has been implemented, they operate with limited feedback about whether the solution actually solves the intended problem. In this situation, incorrect assumptions are likely to accumulate, potentially rendering the product unusable (or useless). Short development cycles mitigate this risk by allowing clients to request changes as early as possible. Early validation reduces wasted effort on complex implementations that may later be discarded.

These rapid adjustment cycles function as clear signals that guide future decisions. When developers introduce a variation—such as a new design approach or a code change—the system must respond quickly so that the outcome of that variation is unambiguous. If feedback is delayed, the connection between a decision and its result becomes unclear (Skinner 1981), making it harder to learn which choices were effective. In software development, when a solution is presented, the client effectively selects the alternative that produces better results. Over time, this selection process clarifies which practices improve outcomes and which require modification, in the same way that environmental constraints shape successful adaptations in evolutionary systems (Darwin 1859).

To prevent the system from drifting toward developer-centered value, each increment must provide observable utility. Decisions based solely on the developers' local perspective may favor familiar solutions or personal convenience rather than actual user needs. In such cases, features are shaped by ease of implementation instead of by customer demand.

6.1.2 Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Unlike rigid or prematurely fixed development approaches, agile explicitly embraces change. This is one of the most significant advances introduced by agile software development methodologies. By encouraging frequent adjustments, agile exposes developers to a wider range of feedbacks. Over time, teams can identify which approaches best satisfy client requirements, leading to solutions that are better aligned with real needs.

When a system is exposed to diverse feedback during its evolution, it tends to develop a broader set of effective responses (Skinner 1981). In human terms, this corresponds to accumulated experience: encountering multiple outcomes allows better judgment in future situations and improves the ability to adapt solutions to new contexts. The same dynamic applies to software. Teams that experiment with alternatives and evaluate their outcomes are better equipped to respond appropriately when requirements or constraints change.

This effect can be observed in the case study. There was no resistance to exploring different technologies or architectural alternatives while searching for a viable solution. Initially, the team evaluated implementing the entire project using native Android development with direct access to the ARCore API. Although this approach offered fine-grained control, it introduced significant complexity. That exploration, however, was not wasted effort. It enabled the team to make an informed comparison and ultimately select a more suitable alternative: developing with a game engine while using AR Foundation, which builds on the ARCore concepts already studied. As a result, prior work contributed to better decision-making rather than being discarded entirely.

6.1.3 Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

In many immature project making, product development follows a rigid pipeline with a clearly defined beginning, middle, and end. This structure assumes that requirements can be fully specified upfront and validated only at the final stage. Software development, however, does not fit this model well. Software functionality must be continuously tested, evaluated, and adjusted as new information emerges. Software must be soft, malible, not stiff and unchangeble (Martin 2008). The shorter the time between implementing a change and receiving client evaluation, the greater the chance that useful changes will be identified, kept, extended and reinforced.

For effective selection among alternatives to occur, outcomes must follow decisions with minimal delay. Systems do not improve by relying on long-delayed evaluation; improvement happens when the effects of an action are observed close in time to the action itself (Skinner 1969). As this temporal distance grows, it becomes increasingly difficult to determine which decisions led to which outcomes (Skinner 1981). Immediate feedback strengthens the connection between an implementation choice and its result, increasing the likelihood that successful patterns will be repeated (Skinner 1984). In software engineering terms, this corresponds to a stable relationship in which clients continue to fund development because delivered features consistently produce value.

This principle is directly supported by continuous integration (CI) and continuous delivery (CD). From a software engineering standpoint, CI works by frequently merging code changes into a shared repository, running automated tests, and detecting integration errors early. Each change produces an immediate, observable system state: either the software still works as expected, or it does not. This rapid feedback loop allows ineffective changes to be identified and reverted quickly, while effective ones are preserved and built upon. In this sense, CI functions as a practical selection mechanism: only changes that survive testing and real execution remain in the codebase.

Moreover, continuous delivery extends this process by ensuring that the software is always in a deployable state. Instead of accumulating large batches of unvalidated changes, CD favors small, incremental modifications that can be tested, deployed, and evaluated almost immediately by real users in real case scenarios. From a systems perspective, this reduces uncertainty. Each increment has a clear effect on the running system, making it easier to attribute observed outcomes to specific changes.

This approach also supports long-term code quality. Codebases developed under rigid, long-term plans often evolve without sufficient feedback from execution and use. As a result, architectural decisions harden prematurely, dependencies increase, and coupling becomes tighter. As argued by Martin (Martin 2008), the best software architectures are those that delay decisions that would restrict future alternatives.

Another counterpoint of not delivering working software as soon as possible is the resources waste from invaliable software. When a feature requires substantial time and effort to complete before it can be evaluated, teams become reluctant to discard it—even if it proves ineffective. This is a well-known effect in software engineering, often discussed as sunk-cost bias, but here it can be understood structurally: delayed feedback increases the cost of correction.

By contrast, producing code only when it demonstrably supports client goals limits unnecessary structural growth. Features are kept small enough that rejecting or refactoring them remains inexpensive. This mirrors how adaptive systems evolve: changes persist only if they improve performance under current constraints, not because they were planned far in advance.

6.1.4 Business people and developers must work together daily throughout the project.

Developers bring technical expertise shaped by repeated engagement with programming tasks, tools, architectures, and design practices. This accumulated experience allows them to implement what is requested, but it does not, by itself, guarantee that the resulting software will deliver value once deployed. Technical correctness and business usefulness are distinct dimensions, and success in production depends on both.

Business representatives bring experience related to product value, user needs, and market constraints. When they participate as active members of the team, their input helps align technical decisions with client expectations. Under these conditions, the team gains shared understanding of the problem domain and the economic forces shaping the product. This shared understanding guides decisions about what should be built first and what can be deferred. Through repeated cycles of collaboration, feedback, and adjustment, developers gradually acquire domain-specific knowledge that complements their technical skills. This is what is refered in behaviorism as acquire repertoire (Skinner 1969).

This interaction is mutually beneficial. Increasing product value is a cooperative effort rather than a task owned by a single role. Developers understand the operational implications of prioritizing a feature in the backlog—such as dependencies, technical risks, and implementation costs (Beck e Andres 2004). By communicating these constraints clearly, they enable business representatives to form more realistic expectations about timelines and delivery scope.

Clearly communicating such limitations to business stakeholders reduces unnecessary workload and developer overwork. It prevents quality degradation and avoids defects that often arise when rushed implementations are pursued solely to meet fixed deadlines without regard for technical readiness.

Mastery of this domain emerges through close collaboration and observation of how experienced business stakeholders reason about value and constraints. By absorbing these patterns, developers adjust their technical decisions to better match real-world demands. This alignment increases the likelihood that each delivered increment functions correctly in production and contributes meaningfully to the overall product.

6.1.5 Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

Motivation should not be understood as an internal force that appears spontaneously or exists prior to action. Rather, it emerges from the interaction between people and the conditions in which they work. What is commonly labeled as “motivation” becomes visible only after actions have repeatedly led to positive outcomes. As Skinner emphasized, people act because similar actions have worked before, not because of an assumed internal energy that initiates behavior (Skinner 1969). When certain actions reliably lead to beneficial results in a given context, those actions tend to be repeated. Observers then describe this recurring pattern as someone “being motivated.”

In software development, this is straightforward to observe. Developers experience positive outcomes when implementing a feature that works, is appreciated, or leads to progress. Social feedback plays a particularly strong role in many teams. Simple actions such as recognizing effective solutions, acknowledging effort, or encouraging the continuation of good practices reinforce productive work patterns. These responses do not need to be elaborate; timely and clear feedback is often sufficient to sustain effective behavior.

In contrast, approaches based on pressure or punishment tend to produce the opposite effect. As Sidman warns, forcing compliance through threats or negative consequences may produce short-term results, but it also generates disengagement, isolation, rigidity, hostility, and resistance over time (Sidman 1989). In development teams, this often appears as reduced initiative, minimal compliance, or reluctance to propose alternatives—all of which undermine long-term performance.

Another factor that influences what is commonly called motivation is the availability of valued outcomes. When access to something desirable—such as autonomy, recognition, or meaningful work—is limited, it tends to become more valuable. When it is overused or guaranteed regardless of contribution, its effect weakens (Michael 1982). For this reason, positive outcomes should not be distributed mechanically or excessively. Instead, their availability should vary in a way that highlights the connection between effective work and meaningful results (Skinner 1981).

However, limiting access to valued outcomes must not take the form of coercion. Sidman describes situations in which people must comply simply to avoid losing something as inherently coercive (Sidman 1989). Although not every constraint is harmful, systems built around "do this or lose access" function much like punishment. They rarely support stable, long-term engagement. While scarcity can increase the perceived value of an outcome, using deprivation as a control mechanism often creates competition, tension, and social pressure, which undermine cooperation within teams (Sidman 1989).

When managed carefully and without coercion, variation in access to valued outcomes can increase engagement (Skinner 1984). Under such conditions, what is described as motivation improves a person's ability to acquire the specific skills and work patterns needed for high-quality output. Consistent involvement in a task increases the chance of encountering successful results, reinforcing effective approaches and sharpening the ability to distinguish productive actions from unproductive ones.

Finally, what may appear as "innate motivation" is better understood as the result of prior experience. Recognizing this prevents teams from treating motivation as an inherent personal trait. Instead, it shifts attention toward designing work environments that reliably support engagement, learning, and cooperation—conditions under which motivated behavior can emerge and be sustained across the entire team.

6.1.6 The most efficient and effective method of conveying information to and within a development team is face-to-face conversation

Emails and text messages are prone to ambiguous interpretation. Human communication relies not only on words but also on posture, tone of voice, facial expression, and timing. These nonverbal elements provide contextual information that helps recipients interpret intent, urgency, and emotional state. Research in communication shows that such cues significantly influence how messages are understood and how people respond to them (Mehrabian 1972). Written communication removes most of this context, making it harder for team members to accurately infer client needs and adjust their actions accordingly.

Asynchronous communication also competes with many other demands on the client's attention. Messages are read alongside unrelated tasks, notifications, and interruptions, which can distort interpretation or delay response. In contrast, face-to-face interaction creates a shared context in which participants focus on the same signals at the same time. This shared setting reduces competing distractions and increases the likelihood that relevant details—such as hesitation, emphasis, or concern—are noticed.

For example, after a sequence of stressful events—often summarized informally as "a bad day"—a client may react more strongly to a minor defect than they would under calmer circumstances. In a synchronous conversation, such contextual factors become immediately visible, allowing the team to interpret the reaction appropriately rather than misclassifying it as a purely technical complaint. Research on communication grounding shows that real-time interaction enables faster clarification, correction of misunderstandings, and alignment between participants (Clark e Brennan 1991).

That said, face-to-face communication is not always feasible. In such cases, the communication environment should be adapted to the client's established habits and preferences. Attempting to force synchronous interaction through pressure or rigid rules is likely to be counterproductive. Effective collaboration emerges when communication methods are shaped by what consistently leads to clarity and progress, not by coercion.

When possible, direct conversation—whether in person or via real-time video online meetings as an efficient medium for aligning the team with the real constraints and priorities of the project.

It strengthens coordination, increases the likelihood that delivered increments are genuinely useful, and reduces errors caused by incomplete or distorted information exchange.

6.1.7 Working software is the primary measure of progress.

In agile development, progress is not inferred from plans, documentation, or isolated components, but from software that demonstrably works. Working software is understood as an integrated system that can be executed, observed, and evaluated in its intended environment. Partially implemented features, mockups, or standalone modules do not represent progress, because they do not yet participate in the operational system. While such artifacts may be useful to developers during construction, they do not provide value to the client until they function as part of the whole.

Only software that actually runs can be evaluated against real conditions. Code that exists only in repositories or features that are not integrated cannot be tested in practice, cannot be meaningfully assessed, and cannot inform future decisions. As a result, these artifacts offer no concrete evidence that the system is improving. In contrast, working software produces immediate, observable outcomes, allowing stakeholders to see what the system does, identify mismatches with expectations, and guide the next development steps.

Moreover using working software as the primary measure of progress also limits overproduction. Plans, specifications, and partially completed features can accumulate without ever being exposed to real usage conditions. This accumulation may create the illusion of advancement while failing to increase the system's actual usefulness. Working software, even in early iterations, must meet minimal standards of correctness, usability, and reliability, making its value concrete rather than hypothetical.

6.1.8 Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

When positive outcomes are delivered excessively, their effect diminishes due to oversupply, as mentioned before (Skinner 1969). Outcomes that occur regardless of performance lose their ability to guide future decisions (Skinner 1981). In agile development, a similar dynamic appears: when signals intended to encourage productive work—such as praise, rewards, or relaxed acceptance criteria—are applied without calibration, they stop clearly indicating which actions are effective. As a result, consistency in productive behavior decreases, and teams may struggle to identify what actually leads to success.

Sudden removal of expected outcomes can be equally disruptive. When a previously reliable outcome stops occurring all at once, people tend to experiment more widely in search of an alternative path forward. In behaviorism, this short period of increased variability is known as an extinction burst. Skinner demonstrated this effect in laboratory settings: when an action that previously produced a result no longer did so, subjects briefly increased the range and frequency of their actions before settling into a new pattern (Skinner 1969).

In software projects, similar patterns emerge when expectations are suddenly disrupted. When teams no longer receive the outcomes they were accustomed to—such as acknowledgment of deliveries, clear acceptance criteria, or predictable feedback cycles—their work patterns often become unstable. This instability can be observed as irregular productivity, frequent context switching, stalled tasks, or abrupt changes in how effort is allocated across features.

Stakeholders may react in parallel. When expected results stop materializing, they often attempt to recover value by redefining project goals, narrowing scope, or redirecting the product toward alternative uses. In more severe cases, they may replace collaborative incentives with financial pressure, stricter deadlines, or contractual threats in an effort to force progress.

From a systems perspective, these reactions are not random. They are responses to the sudden removal of previously reliable signals that guided behavior—such as sprint completion rules, delivery approval, or continued investment. When these signals disappear without a gradual transition, both teams and stakeholders experiment with alternative strategies in an attempt to restore predictability and control.

In practical terms, this highlights the importance of maintaining stable feedback mechanisms in development processes. When evaluation criteria, incentives, or validation loops change abruptly, the system as a whole enters a volatile state. Preserving continuity in how progress is recognized and validated reduces erratic behavior and helps teams adapt incrementally rather than through disruptive shifts.

6.1.9 Continuous attention to technical excellence and good design enhances agility.

Design is a broad concept applied across many domains, but its core purpose remains consistent: to produce structures that work effectively and communicate clearly. Good design prioritizes human understanding, ensuring that artifacts can be read, reasoned about, modified, and extended without unnecessary effort. As commonly emphasized in software engineering literature, "any fool can write code that a computer can understand. Good programmers write code that humans can understand" (Fowler 2018).

Within agile development, technical excellence and sound design are not optional enhancements but foundational requirements for agility itself. Agility should not be confused with improvisation or the absence of planning. Although agile processes promote frequent delivery, continuous integration and deployment must not be used to justify weakened coding standards or careless architectural decisions. Poorly designed systems accumulate structural complexity over time, which restricts future change and makes adaptation progressively slower and more expensive.

Consistent attention to design quality reduces friction throughout the development process. Code that is readable, modular, and well-structured allows new functionality to be introduced with minimal impact on existing behavior. This structural clarity expands the range of feasible future changes, enabling teams to respond quickly when requirements evolve. Agility, therefore, does not arise from neglecting design, but from maintaining technical excellence continuously across the entire development lifecycle.

6.1.10 Simplicity—the art of maximizing the amount of work not done—is essential.

Simplicity plays a central role in preventing unnecessary development and in preserving a system's ability to change. Rather than encouraging early expansion or speculative features, agile practice emphasizes limiting scope to what is strictly required at each stage. As Fowler notes, "Simplicity—the art of maximizing the amount of work not done—is essential" (Fowler 2001). This idea, widely known in software engineering as the KISS guideline, constrains structural growth and helps control long-term maintenance costs.

When complexity is minimized, individual components are more likely to remain robust as requirements evolve. Simple implementations are easier to understand, evaluate, modify, or remove when they fail to meet expectations. If a component is small and loosely coupled, discarding or replacing it imposes minimal cost on the rest of the system, enabling rapid correction of ineffective solutions.

Reaching this level of simplicity depends on clearly defined user stories with explicit completion criteria. These constraints focus development on observable functionality that satisfies concrete requirements, preventing features from being introduced based on speculation rather than need. In this way, simplicity supports continuous adjustment while preserving flexibility, ensuring that development effort is directed toward functionality that produces measurable value.

6.1.11 The best architectures, requirements, and designs emerge from self-organizing teams.

Self-organizing teams operate without rigid hierarchical control, relying instead on shared responsibility and a common understanding of goals. When team members have autonomy to make decisions and a clear view of what needs to be done, coordination overhead is reduced and development tends to move faster. In contrast, when authority attempts to enforce behavior through pressure, threat, or excessive control, teams lose flexibility. Such environments discourage experimentation and suppress alternative solutions, limiting the system's ability to adapt to change (Sidman 1989).

In a self-organizing team, individuals align their actions with shared objectives rather than focusing on compliance with directives. When roles are understood and expectations are explicit, coordination emerges through continuous adjustment between team members instead of through command-and-control structures. This setup supports rapid exploration of architectural and design alternatives, allowing effective solutions to emerge based on how well they perform in practice.

Under these conditions, collaboration and collective oversight develop naturally. Team members observe the outcomes of each other's decisions and adjust their own work accordingly. Practices that lead to successful results tend to be repeated and shared, while ineffective approaches are abandoned. Over time, this leads to stronger architectures and better-aligned requirements—not because they were imposed from above, but because they consistently proved effective within the constraints of the project.

6.1.12 At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

The Agile Manifesto states that "the best architectures, requirements, and designs emerge from self-organizing teams" (Beck et al. 2001). In such teams, ways of working are shaped primarily by ongoing feedback from the development process itself rather than by rigid, externally imposed control. Regular moments of reflection—such as retrospectives—allow teams to observe the consequences of their own practices. Ineffective approaches can then be reduced, while practices that improve reliability, maintainability, and delivered value are reinforced through continued use. This recurring evaluation closely resembles an evolutionary process: teams try different approaches during their work, and those that prove effective under real constraints are retained.

When problems arise in software development, the manner in which they are addressed is crucial. Harsh criticism and personal blame do not lead to improvement. Such responses tend to reduce participation, discourage initiative, and limit exploration of alternative solutions. Rather than enabling change, blame suppresses the very variation needed to discover better ways of working.

Improvement requires shifting attention from individuals to the conditions under which work is performed. By examining what leads to inefficiencies—such as unclear requirements, delayed feedback, or conflicting priorities—teams can adjust the environment in which decisions are made. Changing these conditions makes effective practices more likely to occur naturally, without relying on enforcement or pressure.

Instead of assigning fault, teams should focus on proposing immediate and actionable adjustments. By redesigning workflows to support collaboration, experimentation, and rapid feedback, teams actively guide their own development process toward more effective patterns. Continuous improvement, in this sense, is not driven by authority, but by the systematic selection of practices that consistently produce better outcomes.

6.2 Punishment in Agile

Punitive dynamics often emerge in agile environments even when teams explicitly promote collaboration. In practice, pressure-based control appears in many socially accepted forms, partly because both organizational habits and cultural norms have historically favored suppressing behavior that is perceived as disruptive to group activity (Sidman 1989). These dynamics persist even in teams that formally adopt agile values.

In software organizations, such pressure frequently regulates developer activity through harsh or intimidating interactions with clients; sarcastic or dismissive remarks during meetings; preferential treatment of a small subset of “exceptional” contributors while marginalizing consistently competent ones; public reprimands during daily stand-ups; and unstable or ambiguous task definitions that increase cognitive and emotional load—particularly for neurodivergent developers. Other common mechanisms include repeated formal warnings and the systematic assignment of undesirable or low-status tasks.

Pressure can also be introduced through practices that publicly expose mistakes. Examples include meetings where communication failures are highlighted in front of peers, or requiring developers to lead client demonstrations that reveal unfinished or flawed implementations. These situations often occur during daily ceremonies or retrospectives and operate by associating participation with embarrassment or stress. Additional cases include documenting resistance to proposed changes as a negative marker, assigning extra documentation or review duties as a response to dissent, or framing disagreement as misalignment with organizational values. In some teams, developers who question decisions are required to take responsibility for refining the very requirements they challenged, increasing the personal cost of raising concerns.

Loss of access to valued activities is another common mechanism. Developers may be excluded from client discussions, removed from collaborative decision-making, denied assistance when requesting help, or stripped of autonomy in task selection. In these cases, participation in reinforcing aspects of the work—such as influence, learning opportunities, or meaningful collaboration—is restricted. Organizations may further narrow recognition by valuing only contributions that align with imposed changes, discounting broader or dissenting contributions.

These dynamics often extend into broader organizational practices. Developers may be dismissed, and metrics generated by agile routines—such as burndown charts, version control activity, or delivery statistics—can become tools that contribute to termination decisions. Other pressure-based practices include removing bonuses, restricting task choice, excluding individuals from client-facing work, or introducing competitive gamification systems that regulate access to privileges. Such conditions are widely reported in development communities and are frequently associated with descriptions of high-strain or abusive work environments.

While these approaches can suppress unwanted behavior in the short term, they do not build the stable patterns of action required for long-term project success. Instead, sustained exposure to pressure often produces resistance or disengagement. Developers subjected to repeated criticism may reduce collaboration, limit communication, or minimize visible effort. These outcomes are predictable responses to coercive environments rather than indicators of individual inadequacy (Sidman 1989).

Fear-based environments produce a different dynamic. Anticipation of punishment encourages avoidance rather than engagement. One common manifestation is procrastination. In most cases, procrastination does not reflect laziness or lack of commitment but rather a rational shift toward activities that offer more immediate or reliable positive experiences. When the work environment becomes aversive, alternative activities—such as distractions or low-effort tasks—become more attractive, weakening sustained focus and progress.

As a result, productivity declines because both management and developers begin optimizing primarily to avoid negative consequences rather than to improve the product. Effort shifts toward risk avoidance, minimal compliance, and defensive behavior. As Sidman demonstrated, pressure-based control does not create effective work patterns; it suppresses activity, encourages avoidance, and gradually erodes the system's capacity to adapt (Sidman 1989). Skinner similarly emphasized that such approaches lead only to temporary suppression and, when sustained, drive systems toward minimal output rather than durable improvement (Skinner 1953). Under these conditions, teams converge on the smallest amount of work necessary to avoid negative outcomes, and the variation required for learning and improvement steadily disappears.

6.3 Reinforcement Instead

Punishment does not lead to durable improvements in how teams work. Stable progress emerges when effective actions are consistently followed by positive outcomes. In high-performing development teams, improvement is achieved by encouraging and amplifying practices that increase project value, rather than by suppressing deviations through pressure or blame. When mistakes or shortcomings are treated as failures deserving reprimand, participation decreases and experimentation narrows. By contrast, allowing multiple solution paths to be explored increases the chance that effective approaches will emerge and be retained.

Supportive feedback can take both social and material forms. Public recognition during reviews, acknowledgment of effective problem-solving in meetings, or increased autonomy after successful delivery all function as signals that certain practices are valued. For example, when a developer introduces a refactoring that reduces complexity and improves maintainability, immediate recognition and continued opportunity to work on similar improvements increase the likelihood that this approach will be used again. Likewise, timely confirmation that a delivered feature meets user needs reinforces attention to quality and alignment with requirements. When the software itself produces visible benefits—such as reduced defects, improved usability, or positive client feedback—these outcomes further strengthen the connection between agile practices and successful results.

For learning to be sustained and transferable, improvement must occur incrementally. Large changes in how teams work rarely appear all at once. Instead, effective practices are adopted gradually through small adjustments that are validated and encouraged along the way. Organizational maturity models provide a structured way to support this process. As teams progress through higher levels of process maturity, useful practices are reinforced through concrete outcomes such as expanded decision-making authority, increased trust, financial incentives, or formal recognition. Frameworks like CMMI, for example, define specific process areas whose successful adoption can be followed by such consequences. For these mechanisms to work, feedback must be timely and clearly connected to the practice being encouraged; otherwise, the improvement will not take hold.

For this reason, partial progress should be supported rather than discouraged. If a developer begins involving the client more actively—something that did not previously occur—but does not yet fully meet expectations around simplicity or technical excellence, the appropriate response is encouragement rather than reprimand. Strengthening this initial change increases the likelihood that further refinements will follow in subsequent iterations. Over time, closer alignment with agile principles emerges through repeated cycles of feedback and adjustment, rather than through enforcement or criticism.

7 Conclusion

People change, markets shift, user preferences vary, projects are interrupted, and business decisions redirect priorities. These continuous environmental alterations select some implementations over others, but instability and pressure alone do not generate functional variability. Selection operates only on variation that is already present, and variability does not arise from coercion or urgency.

Indeed, variation emerges as a product of prior shaping, and its outcomes are selected by subsequent consequences (Skinner 1981). No effective response appears without a history of reinforcement, and the absence of a given repertoire cannot be attributed to those whose behavior has never been shaped or strengthened. When teams fail to produce robust solutions, the controlling variables lie in the training conditions, not in individual intention or effort.

Empowering developers therefore requires repeated contact with consequences through experimentation, testing, and incremental delivery, rather than punishment for reduced throughput or failed attempts. Restricting exposure to diverse conditions and relying on aversive control accelerates behavioral extinction (Sidman 1989). Innovation supplies behavioral variation, but only consequences that strengthen functional responses allow those variations to persist and contribute to long-term viability (Darwin 1859).

Agile environments counteract coercion when arranging frequent feedback, visible consequences, and reinforcement for functional progress. In the MatematecAR project, productivity and learning increased precisely because agile practices were adopted as a way to ensure continuous variation and selection. Developer efforts were socially recognized and reinforced by Professor Eduardo Colli, even when prototypes failed to meet expectations. This arrangement sustained exploration, prevented extinction of initiative, and transformed unsuccessful prototypes into sources of functional adaptation. Without such reinforcing contingencies, MatematecAR would not have functioned as a productive experimental environment, nor would its development process have remained genuinely agile.

References

Aleem, Capretz e Ahmed 2016 ALEEM, S.; CAPRETZ, L. F.; AHMED, F. Game development software engineering process life cycle: A systematic review. *Journal of Software Engineering Research and Development*, 2016.

Aleem, Capretz e Ahmed 2018 ALEEM, S.; CAPRETZ, L. F.; AHMED, F. Critical success factors to improve the game development process from a developers perspective. *arXiv*, 2018.

Azuma 1997 AZUMA, R. T. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, v. 6, n. 4, p. 355–385, 1997.

Beck e Andres 2004 BECK, K.; ANDRES, C. *Extreme Programming Explained: Embrace Change*. 2. ed. [S.l.]: Addison-Wesley, 2004. ISBN 978-0321278654.

Beck et al. 2001 BECK, K. et al. *Manifesto for Agile Software Development*. 2001. Disponível em: <https://agilemanifesto.org/>.

Billinghurst, Clark e Lee 2015 BILLINGHURST, M.; CLARK, A.; LEE, G. A. A survey of augmented reality. In: *Foundations and Trends in Human–Computer Interaction*. [S.l.: s.n.], 2015. v. 8, n. 2, p. 73–272.

Bloom et al. 2007 BLOOM, J. D. et al. Evolution favors protein mutational robustness in sufficiently large populations. *arXiv*, 2007.

Child in Time: Children as Liminal Agents in Upper Paleolithic Decorated Caves CHILD in Time: Children as Liminal Agents in Upper Paleolithic Decorated Caves.

Clark e Brennan 1991 CLARK, H. H.; BRENNAN, S. E. *Grounding in Communication*. Washington, DC: American Psychological Association, 1991.

Clune, Mouret e Lipson 2013 CLUNE, J.; MOURET, J.-B.; LIPSON, H. The evolutionary origins of modularity. *Proceedings of the Royal Society B*, v. 280, n. 1755, 2013.

Craig CRAIG, A. B. *Understanding Augmented Reality*. [S.l.: s.n.].

Cuperschmid, Ruschel e Freitas 2012 CUPERSCHMID, A. R. M.; RUSCHEL, R. C.; FREITAS, M. R. D. Technologies that support augmented reality applied in architecture and construction. *Cadernos*, n. 19, 2012.

Darwin 1859 DARWIN, C. *On the Origin of Species*. London: John Murray, 1859.

Developers 2025 DEVELOPERS, G. *ARCore API Reference*. 2025. <https://developers.google.com/ar/reference>.

Draghi et al. 2022 DRAGHI, J. A. et al. Exploring the expanse between theoretical questions and evolvability. *Philosophical Transactions of the Royal Society B*, v. 377, n. 1856, 2022.

DualSense Wireless Controller DUALSENSE Wireless Controller. Disponível em: <https://www.playstation.com/pt-br/accessories/dualsense-wireless-controller/>.

Eco ECO, U. *Tratado Geral de Semiótica*. [S.l.: s.n.].

Fowler 2001 FOWLER, M. *Agile Software Development: Principles, Patterns, and Practices*. [S.l.]: Addison-Wesley, 2001.

Fowler 2018 FOWLER, M. *Refactoring: Improving the Design of Existing Code*. 2. ed. Boston: Addison-Wesley, 2018.

Friedlander et al. 2013 FRIEDLANDER, T. et al. Mutation rules and the evolution of sparseness and modularity in biological systems. *arXiv*, 2013.

Gamma et al. 1994 GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley, 1994.

Glenn 1988 GLENN, S. S. Contingencies and metacontingencies: Toward a synthesis of behavior analysis and cultural materialism. *The Behavior Analyst*, v. 11, n. 2, p. 161–179, 1988.

Hansen 2003 HANSEN, T. F. Is modularity necessary for evolvability? *Evolution*, v. 57, n. 8, p. 1523–1530, 2003.

Houle e Rossoni 2022 HOULE, D.; ROSSONI, D. M. Complexity, evolvability, and the process of adaptation. *Annual Review of Ecology, Evolution, and Systematics*, v. 53, p. 137–159, 2022.

Kock 2005 KOCK, N. Media richness or media naturalness? *IEEE Transactions on Professional Communication*, v. 48, n. 2, p. 117–130, 2005.

Malthus 1798 MALTHUS, T. R. *An Essay on the Principle of Population*. London: J. Johnson, 1798.

Martin 2008 MARTIN, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. [S.l.]: Prentice Hall, 2008. ISBN 978-0132350884.

Masel 2010 MASEL, J. Robustness and evolvability. *Trends in Ecology Evolution*, v. 25, n. 9, p. 406–414, 2010.

McColgan et al. 2024 MCCOLGAN, et al. Understanding developmental system drift. *Philosophical Transactions of the Royal Society B*, v. 379, n. 1903, 2024.

Mehrabian 1972 MEHRABIAN, A. *Silent Messages*. Belmont, CA: Wadsworth, 1972.

Melo e Marroig 2016 MELO, D.; MARROIG, G. Modularity: Genes, development and evolution. *Philosophical Transactions of the Royal Society B*, v. 371, n. 1688, 2016.

Michael 1982 MICHAEL, J. Distinguishing between discriminative and motivating functions of stimuli. *Journal of the Experimental Analysis of Behavior*, v. 37, n. 1, p. 149–155, 1982.

Milgram 1994 MILGRAM, P. A taxonomy of mixed reality visual displays. In: . [S.l.: s.n.], 1994.

Nietzsche 1873 NIETZSCHE, F. *On Truth and Lie in an Extra-Moral Sense*. 1873.

Paulk 2001 PAULK, M. C. *A History of the Capability Maturity Model for Software*. [S.l.]: SEI, Carnegie Mellon University, 2001.

Pélabon et al. 2025 PéLABON, C. et al. Evolvability: Progress and key questions. *BioScience*, v. 75, n. 1, p. 1–14, 2025.

Rolland, Hua e Gao 1994 ROLLAND, J. P.; HUA, H.; GAO, G. A calibration method for head-mounted displays. In: IEEE. *Proceedings of Virtual Reality Annual International Symposium*. [S.l.], 1994. p. 246–255.

Schmandt-Besserat SCHMANDT-BESSERAT, D. *How Writing Came About*. [S.l.: s.n.].

Schubert, Qin e Cremers 2021 SCHUBERT, D.; QIN, T.; CREMERS, D. Visual–inertial tracking for mobile augmented reality. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

Schwaber SCHWABER, K. *SCRUM Development Process*.

Sidman 1971 SIDMAN, M. Equivalence relations and behavior: A research story. 1971.

Sidman 1971 SIDMAN, M. Reading and auditory-visual equivalences. 1971.

Sidman 1989 SIDMAN, M. *Coercion and Its Fallout*. [S.l.]: Authors Cooperative, 1989.

Skinner 1932 SKINNER, B. F. On the rate of formation of a conditioned reflex. *Journal of General Psychology*, v. 7, n. 1, p. 22–33, 1932.

Skinner 1938 SKINNER, B. F. *The Behavior of Organisms: An Experimental Analysis*. New York: Appleton-Century-Crofts, 1938.

Skinner 1953 SKINNER, B. F. *Science and Human Behavior*. [S.l.]: Macmillan, 1953.

Skinner 1969 SKINNER, B. F. *Contingencies of Reinforcement: A Theoretical Analysis*. New York: Appleton-Century-Crofts, 1969.

Skinner 1974 SKINNER, B. F. *About Behaviorism*. New York: Alfred A. Knopf, 1974.

Skinner 1981 SKINNER, B. F. Selection by consequences. *Science*, v. 213, n. 4507, p. 501–504, 1981.

Skinner 1984 SKINNER, B. F. The evolution of behavior. *Journal of the Experimental Analysis of Behavior*, v. 41, n. 2, p. 217–221, 1984.

Skinner 1986 SKINNER, B. F. The evolution of verbal behavior. *Journal of the Experimental Analysis of Behavior*, v. 45, n. 1, p. 115–122, 1986.

Steed 2016 STEED, A. Understanding and measuring latency in virtual reality systems. *Presence: Teleoperators and Virtual Environments*, v. 25, n. 2, p. 123–136, 2016.

Wen, Schwerdtfeger e Klinker 2013 WEN, M.; SCHWERDTFEGER, B.; KLINKER, G. Mobile augmented reality: A systematic review of techniques, challenges, and applications. In: IEEE. *Proceedings of the IEEE International Symposium on Mixed and Augmented Reality*. [S.l.], 2013. p. 1–10.