Universidade de São Paulo
Instituto de Matemática, Estatística e Ciência da Computação
Bacharelado em Ciência da Computação

Monografia Final
Trabalho de Formatura Supervisionado
MAC0499

PyBlox: Uso de Interfaces Tangíveis e Realidade Aumentada no Aprendizado de Python

Clara Yuki Sano e Júlia Melo Teixeira dos Santos
Orientador: Prof. Dr. Carlos Hitoshi Morimoto

São Paulo
Dezembro de 2025

Agradecimentos

“Dedico este trabalho à minha mãe, Ana Karina, e a agradeço pelo seu apoio e incentivo durante toda minha trajetória pessoal, acadêmica e profissional. Sem ela, as comemorações perderiam seu brilho e os momentos difíceis pareceriam insuperáveis.

Agradeço à minha família por torcer por mim e vibrar com minhas conquistas. Sei que posso contar com todos, independente da distância física. Agradeço ao Matheus por compartilhar sua força e sua motivação no encerramento da graduação que vivenciamos juntos. Agradeço à Idian pelas risadas e pelas histórias que acumulamos desde o primeiro dia de universidade.

Por fim, agradeço à Yuki por dividir os esforços e trazer leveza para esse desafio, ao Prof. Dr. Hitoshi por sua orientação e à Universidade de São Paulo pelas oportunidades e aprendizados que moldaram a minha maneira de ver e contribuir para o mundo.”

Júlia Melo Teixeira dos Santos

“Agradeço ao professor Hitoshi, por sua orientação durante o ano de desenvolvimento do TCC.

À minha amiga Samina, por ter topado a ideia de fazer este trabalho em conjunto. Não sei o que teria sido de mim enfrentando esse desafio sem você, obrigada de coração pelo apoio e pela amizade que construímos juntas nesses últimos anos.

À minha mãe Simone, meu pai Ruben, minha irmã Sophia e meu irmão Enrique. Apesar da distância geográfica que se estabeleceu entre nós nos últimos quatro anos, me sinto a cada dia que passa cada vez mais energizada pelo amor de vocês. Obrigada por sempre acreditarem em mim e me apoiarem incondicionalmente em tudo.

Ao meu companheiro, Pedro. Obrigada por sempre compartilhar comigo suas paixões; posso perceber o quanto elas incendeiam seu coração, e isso irá para sempre me inspirar :) Viver a vida contigo é muito mais divertido!

E à minha vovó Edna, que sempre me dá forças nos momentos mais difíceis. Te amo infinitamente.”

Clara Yuki Sano

Resumo

Clara Yuki Sano e Júlia Melo Teixeira dos Santos. PyBlox: Uso de Interfaces Tangíveis e Realidade Aumentada no Aprendizado de Python. Monografia (Bacharelado). Instituto de Matemática, Estatística e Ciência da Computação, Universidade de São Paulo, São Paulo, 2025.

Este trabalho detalha o desenvolvimento do **PyBlox**, uma aplicação que utiliza Realidade Aumentada (*Augmented Reality*, AR) para o ensino introdutório de lógica de programação em Python. Seu funcionamento consiste na manipulação de blocos físicos pelo usuário que serão rastreados e associados a trechos de código Python pelo *software*, resultando na simulação do código final quando o arranjo dos blocos estiver correto. Apesar de inspirado na linguagem Scratch, este trabalho se destaca ao empregar código em uma linguagem de programação popular e interação física, visando promover uma experiência pedagógica mais imersiva e envolvente. A implementação da aplicação utiliza a *engine* de jogos Unity, sua *framework* AR Foundation e a biblioteca Google ARCore.

Palavras-chave: Realidade Aumentada (AR), Realidade Estendida (XR), Ensino de Programação, Aprendizagem Baseada em Blocos.

Abstract

Clara Yuki Sano and Júlia Melo Teixeira dos Santos. Learning Python with Augmented Reality. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

This thesis details the development of **PyBlox**, an application that utilizes Augmented Reality (AR) for the introductory teaching of programming logic in Python. Its operation is based on the user's manipulation of physical blocks that are tracked and associated with Python code snippets by the software, resulting in the simulation of the final code when the block arrangement is correct. Although inspired by the Scratch language, this work distinguishes itself by employing code in a popular programming language and physical interaction, aiming to promote a more immersive and engaging pedagogical experience. The application's implementation leverages the Unity game engine, its AR Foundation framework, and the Google ARCore library.

Keywords: Augmented Reality (AR), Extended Reality (XR), Teaching Programming, Block-Based Learning.

Lista de Figuras

3.1 Diagrama da interface de renderização da simulação da execução do código 11

3.2 Diagrama da interface do elemento de deslizamento para ajuste de variáveis 12

3.3 Diagrama da interface dos botões incrementais para ajuste de variáveis 13

3.4 Cadeia de mapeamentos do espaço tridimensional do mundo real para espaço bidimensional S para o espaço discreto F. 18

3.5 Aplicação de T_y em comparações feitas no espaço S. 19

3.6 Cadeia de transformações para detecção de bordas na função remota no Google Cloud Run. . 21

3.7 Comparação entre os sistemas de coordenadas utilizados na representação de imagens pelas aplicações Unity (espaço S) e OpenCV (espaço O). 22

3.8 Cadeia de transformações para detecção de bordas na classe auxiliar no projeto da Unity. . . 23

3.9 Conjunto de identificadores **"minimalista"**. 26

3.10 Conjunto de identificadores **"*remake* minimalista"**. 27

3.11 Conjunto de identificadores **"álbuns"**. 27

3.12 Detecção de bordas internas ofuscando bordas externas do conjunto de identificadores **"álbuns"**. 27

3.13 Conjunto de identificadores **"*identicons*"**. 28

3.14 Composição do conjunto de identificadores **"*half-and-half*"** utilizando metades diagonais das variações de cores dos identificadores "*identicons*". 28

3.15 Conjunto de identificadores **"*half-and-half*"**. 28

3.16 Resultado da reformulação de separação da tela. 29

4.1 Tela do dispositivo mostrando os blocos com pedaços de código em camadas sobrepostas a eles, com o resultado de uma execução falha do Python sobrepondo centralmente a visualização. 32

4.2 Tela do dispositivo mostrando os blocos com pedaços de código em camadas sobrepostas a eles, com o resultado de uma execução bem-sucedida do Python sobrepondo centralmente a visualização. 33

4.3 Disposição dos blocos de código que irá gerar uma execução bem-sucedida. 34

4.4 Interface após a execução bem-sucedida. O lado direito é segmentado, exibindo o código Python gerado (superior) e a simulação gráfica do resultado do código (inferior). 35

4.5 Disposição dos blocos de código que irá gerar uma execução falha. 35

4.6 Interface após a execução falha. O lado direito exibe o código Python gerado (superior) e uma mensagem de erro detalhada no *canvas* inferior, auxiliando o usuário na identificação do problema. 36

Sumário

Lista de Figuras **vii**

Sumário **x**

1 Introdução **1**

1.1 Estrutura desta monografia 2

2 Fundamentação teórica **4**

2.1 Realidade Aumentada (AR) vs. Realidade Virtual (VR) 4

2.2 Rastreamento de movimento 4

2.3 *Feature Points* 5

2.4 *Anchor Points* 5

3 Metodologia **7**

3.1 Tecnologias 7

3.1.1 Unity 7

3.1.2 AR Foundation e ARCore 8

3.1.3 Google Cloud 8

3.1.4 OpenCV 8

3.2 Planejamento de estudo 8

3.3 Proposta de solução 8

3.3.1 Objetos reais 9

3.3.1.1 Reconhecimento de blocos 9

3.3.2 Objetos virtuais 10

3.3.2.1 Tradução de arranjo físico para código 10

3.3.2.2 Projeção da simulação 10

3.3.3 Interação com usuário . . . 11
3.3.3.1 Customização de variáveis . . . 11
3.3.3.2 Controle de simulação . . . 13
3.3.4 Tutorial da aplicação . . . 14
3.4 Execução e adaptação de solução . . . 15
3.4.1 Implementação . . . 15
3.4.1.1 Execução de código em Python . . . 15
3.4.1.2 Geração de código a partir do arranjo de blocos . . . 16
3.4.1.3 Detecção de bordas de blocos . . . 18
3.4.1.3.1 Função remota no Google Cloud Run . . . 19
3.4.1.3.2 Classe auxiliar no projeto da Unity . . . 21
3.4.1.4 Simulação de código executado . . . 23
3.4.1.4.1 Algoritmo de mapeamento e visualização . . . 24
3.4.2 Interface . . . 25
3.4.2.1 Imagens usadas como identificadores de blocos . . . 25
3.4.2.2 Separação da tela . . . 29

4 Resultados **32**
4.1 Versão inicial . . . 32
4.2 Versão final . . . 33

5 Conclusão **38**
5.1 Considerações finais . . . 38

6 Bibliografia **41**

Capítulo 1

Introdução

O ensino de programação tem conquistado espaço nos currículos das etapas iniciais da educação, antecipando o desenvolvimento do pensamento computacional por crianças e jovens para que seu papel diante dos recursos digitais não seja apenas de consumo, mas também de produção e reflexão crítica. Apesar das novas gerações apresentarem facilidade ao interagir com o mundo tecnológico, a pedagogia defende que o aprendizado formal da lógica de programação é necessário para que elas possam contribuir ativamente para as comunidades digitais. Especialmente quando aplicada de forma ampla, abrangendo tanto redes públicas quanto particulares, essa iniciativa promove inclusão social, uma vez que o domínio da computação é capaz de afugentar a exclusão digital [1].

Impulsionados por esse mesmo cenário, pesquisadores do MIT Media Lab desenvolveram uma linguagem de programação visual baseada em blocos, chamada de **Scratch**. Lançada em 2007 e livremente distribuída para as plataformas Windows, MacOS e Linux, essa ferramenta se consolidou como revolucionária por partir da percepção de que jovens são pensadores que constroem conhecimento ao engajar ativamente em experiências de aprendizagem [10] e materializar essa noção em um produto intuitivo e acessível, capaz de ser compreendido por desenvolvedores de qualquer idade e sem experiência prévia em programação.

A interface do Scratch permite que os usuários interajam à moda *drag and drop* (arrastar e soltar) com trechos de código encapsulados em blocos encaixáveis para que sua combinação resulte em um código completo e funcional. É inegável, entretanto, que a experiência pedagógica proporcionada poderia ter suas qualidades imersivas e práticas extrapoladas pela incorporação de **interfaces tangíveis** e de uma **linguagem de programação difundida nos meios profissional e acadêmico**.

Diante da urgência em tornar o ensino de computação mais acessível e das amplas oportunidades de avanço no cenário de ferramentas disponíveis para tal propósito, esse trabalho propõe e executa a criação do **PyBlox**, uma aplicação voltada para introdução da lógica de programação em Python através de Realidade Aumentada, que pode ser facilmente replicado e difundido.

A adição de aspectos de Realidade Aumentada viabiliza o emprego de uma **TUI**, ou *Tangible User Interface* (Interface de Usuário Tangível), e enraíza a interação entre o

usuário e o software no mundo físico ao mesmo tempo que a complementa através de uma interface virtual auxiliar. Dessa forma, a manipulação de blocos físicos e a projeção dos trechos de código sobre seus respectivos blocos na tela de um dispositivo móvel pelo PyBlox seria equivalente ao *drag and drop* de trechos de código encapsulados por blocos encaixáveis pelo Scratch.

Já a substituição da linguagem simplificada nativa do Scratch por **Python**, elencada a linguagem de programação mais popular em 5 dos 10 últimos anos pelo índice TIOBE [8], permite que o aprendizado obtido através do PyBlox seja diretamente aplicável nos contextos de mercado e da academia. Uma vez que ambas a lógica e a linguagem de programação são preservadas, é necessário apenas que o usuário se adapte à transição do encaixe de blocos para a escrita de código, facilitando o processo de superação da barreira de entrada no mundo de desenvolvimento de *software*.

1.1 Estrutura desta monografia

No Capítulo 2, serão explorados os conceitos fundamentais de Realidade Aumentada e Estendida que são cruciais para o entendimento do restante desta monografia. Esta seção inclui definições de palavras-chaves e exploração sobre as atuais limitações da área, sem que detalhes matemáticos ou de implementação sejam abordados.

No Capítulo 3, será descrito o processo completo de desenvolvimento do projeto, desde a escolha de tecnologias utilizadas e as propostas iniciais de solução até a efetiva execução da solução, que justifica as adaptações feitas e destrincha as decisões de implementação.

No Capítulo 4, serão apresentadas imagens da versão final do PyBlox e fluxogramas que ilustram as possibilidades de ação do usuário e os resultados exibidos pela aplicação mediante as interações.

Por fim, o Capítulo 5 fará o arremate do projeto ao refletir sobre os resultados obtidos e apontar melhorias que podem ser aplicadas ao PyBlox.

Capítulo 2

Fundamentação teórica

A implementação de uma experiência de Realidade Aumentada como o PyBlox exige a compreensão de alguns conceitos importantes que governam a interação entre o mundo real e o conteúdo digital. Estes conceitos, em sua maioria habilitados por *frameworks* como **Google ARCore**, são essenciais para manter a ilusão de realidade e a coerência na experiência do usuário.

2.1 Realidade Aumentada (AR) vs. Realidade Virtual (VR)

Um dos pontos mais relevantes para entender Realidade Aumentada é saber distinguila da Realidade Virtual. A VR consiste no uso de tecnologia de computador para criar um ambiente simulado, colocando o usuário totalmente dentro de uma experiência e, frequentemente, exigindo hardware específico (como *headsets*) com o objetivo de isolar o mundo real. Em contraste, a AR oferece uma visualização direta ou indireta e em tempo real de um ambiente do mundo real cujos elementos são aprimorados por informações perceptuais geradas por computador, ou seja, pelo conteúdo virtual. O mundo real é, portanto, **mantido e complementado** no caso de AR [11].

2.2 Rastreamento de movimento

Para que os objetos virtuais permaneçam fixos no mundo real e interajam de forma convincente, os dispositivos de Realidade Aumentada dependem de tecnologias avançadas de percepção. O rastreamento em plataformas móveis, como *smartphones*, utiliza o método interno-externo (*inside-out tracking*), no qual a própria unidade de dispositivo usa suas câmeras e sensores inerciais (acelerômetros e giroscópios) para detectar seu movimento e posicionamento [2].

Esse processo é baseado em **SLAM** (*Simultaneous Localization And Mapping*), que consiste no algoritmo central que permite ao dispositivo simultaneamente localizar sua posição e orientação no espaço (localização) e construir um mapa do ambiente ao redor em tempo

real (mapeamento) [19].

2.3 *Feature Points*

Para que o **SLAM** possa funcionar, o software subjacente (como o **Google ARCore**) necessita identificar elementos visuais estáveis no ambiente. Estes são os *Feature points* (pontos de característica), que são capturados e rastreados pelo sistema usando a câmera do dispositivo. Os *Feature Points* são características visualmente distintas, como bordas, cantos e texturas ricas. Agrupamentos desses pontos formam a base para que o algoritmo determine a localização precisa do dispositivo em relação ao mundo circundante.

O sucesso do rastreamento é, portanto, diretamente afetado pela qualidade da detecção desses pontos. Ambientes com baixa luminosidade ou que apresentam superfícies sem textura e contraste (como uma parede branca lisa ou um piso uniforme) dificultam a identificação de *Feature Points*, introduzindo imprecisão no mapa e no posicionamento virtual. Por outro lado, a detecção de um número suficiente e estável desses pontos é o que permite ao sistema avançar para o *Plane Finding* (Entendimento de Planos), que é a capacidade de detectar e gerar superfícies planas, como mesas e pisos, a partir do mapa de pontos [19].

2.4 *Anchor Points*

Anchor points (pontos de ancoragem) são fundamentais no processo de estabilização e colocação de objetos digitais em um ambiente de Realidade Aumentada. Eles consistem em pontos de interesse fixos, definidos pelo usuário ou pelo sistema, sobre os quais os objetos de AR são permanentemente colocados após o motor de visão computacional ter analisado e mapeado o ambiente, identificando planos e referências estáveis.

Capítulo 3

Metodologia

Neste capítulo serão apresentados detalhes da metodologia utilizada para o desenvolvimento do projeto.

3.1 Tecnologias

A conceptualização e implementação do projeto de Realidade Aumentada voltado para a educação exigiu a seleção estratégica de tecnologias que oferecessem flexibilidade, facilidade em realizar ciclos ágeis de prototipagem e compatibilidade com bibliotecas de Realidade Estendida.

3.1.1 Unity

Unity é um motor de jogos multiplataforma amplamente utilizada na indústria de jogos digitais, de visualização arquitetônica e de simulação, as quais comumente apoderam-se de funcionalidades de AR para valorizar seus produtos. Seu público-alvo abrange desde desenvolvedores independentes até grandes estúdios, sendo reconhecida pela sua interface visual de desenvolvimento, vasto ecossistema de bibliotecas e documentação e tutoriais acessíveis.

O ambiente de desenvolvimento integrado (IDE) próprio de aplicações para plataforma Android, chamado de **Android Studio**, foi considerado como alternativa por oferecer controle flexível e direto sobre o hardware dos dispositivos utilizados. Porém, a escolha por Unity em detrimento desta outra plataforma foi motivada principalmente pelo seu caráter amigável para iniciantes. Isso possibilitou que o desenvolvimento acelerado do PyBlox focasse em implementar lógicas avançadas e abstratas de interação entre o mundo virtual e real, enquanto funcionalidades básicas e de baixo nível ficassem a cargo das ferramentas oferecidas pela Unity.

3.1.2 AR Foundation e ARCore

AR Foundation é um arcabouço da Unity que atua como camada de abstração entre a lógica do aplicativo e as bibliotecas de desenvolvimento de *software* (*software development kits*, SDKs) voltados para o desenvolvimento de funcionalidades de AR nativo de sistemas operacionais como **Google ARCore** para Android ou **Apple ARKit** para iOS. Seu principal diferencial é sua portabilidade inerente, que permite que um único projeto seja simultaneamente compatível com dispositivos Android e iOS sem que seja necessário reescrever ou adaptar o código-fonte.

Apesar da apreciação pela versatilidade, o ciclo de criação do PyBlox foi inteiramente imaginado, implementado e validado com a experiência Android em mente. Por isso, considera-se que apenas a biblioteca ARCore contribuiu para a capacidade de percepção e análise do ambiente real pelas lentes virtuais assumidas pelo usuário ao entregar-se à experiência educativa do PyBlox.

3.1.3 Google Cloud

Google Cloud é uma plataforma de serviços de computação em nuvem e sua participação no projeto ocorreu principalmente pelo uso do **Google Cloud Run**. Esse serviço de arquitetura sem servidor foi empregado como núcleo computacional avulso e adicional, sendo responsável por lidar com intensa carga de processamento demandada por algoritmos de visão computacional e pela execução remota de código Python gerado a partir do arranjo dos blocos físicos.

3.1.4 OpenCV

OpenCV é uma biblioteca de código aberto que fornece uma vasta coleção de algoritmos de visão computacional e aprendizagem de máquina. Criada em 2000 pela empresa Intel, essa ferramenta versátil possibilitou a execução das tarefas de processamento de imagem em tempo real pela função de computação remota criada através do **Google Cloud Run**.

3.2 Planejamento de estudo

3.3 Proposta de solução

O desenvolvimento do PyBlox foi precedido por um processo estruturado de design de jogo, materializado em um *Game Design Document* (GDD) que estabeleceu as diretrizes arquiteturais e de experiência do usuário para a aplicação. O documento serviu como instrumento de planejamento estratégico, permitindo a avaliação de diferentes abordagens técnicas antes da implementação efetiva do sistema.

O GDD organizou as decisões de design em três categorias principais: objetos reais, objetos virtuais e interação com o usuário. Para cada categoria, foram analisados diversos aspectos, sendo dado a cada um deles uma solução preferida. Na escolha da solução foram considerados critérios de viabilidade técnica, experiência do usuário e alinhamento com os objetivos pedagógicos do projeto.

3.3.1 Objetos reais

3.3.1.1 Reconhecimento de blocos

A primeira decisão arquitetural fundamental referiu-se ao mecanismo de identificação e diferenciação dos blocos físicos pelo sistema. Duas abordagens principais foram consideradas: reconhecimento baseado em características físicas intrínsecas dos objetos e reconhecimento baseado em identificadores visuais externos.

A primeira abordagem propunha que usuário e sistema reconhecessem a função de cada bloco através de suas características físicas naturais, como cor, tamanho ou formato geométrico. Essa estratégia ofereceria uma interface física minimalista, onde a própria morfologia do objeto seria suficiente para estabelecer sua associação com um trecho específico de código Python. Contudo, essa solução foi descartada devido a limitações tecnológicas impostas pelo escopo do projeto.

A funcionalidade de reconhecimento e rastreamento de objetos tridimensionais reais, necessária para validar as características físicas naturais dos blocos, está disponível exclusivamente na framework **ARKit** para plataformas iOS, enquanto o projeto PyBlox foi direcionado especificamente para a plataforma **Android** utilizando **ARCore**. Além dessa incompatibilidade, a abordagem baseada em características físicas impossibilitaria estratégias de agrupamento visual, como a utilização de cores idênticas para blocos que representam diferentes trechos de código. Essa limitação decorre do fato de que o reconhecimento, acompanhamento e distinção através de visão computacional dependem necessariamente de traços únicos e diferenciadores entre os objetos.

A segunda abordagem, por sua vez, baseia-se no reconhecimento através de identificadores visuais bidimensionais posicionados na superfície superior dos blocos. Nessa estratégia, tanto o usuário quanto o sistema reconhecem que um bloco representa determinado trecho de código ou marcador de indentação através da imagem identificadora presente em sua face superior.

Essa solução resolve simultaneamente as limitações tecnológicas da abordagem anterior, aproveitando as capacidades nativas do **ARCore** para detecção e rastreamento de imagens bidimensionais. A compatibilidade com a plataforma alvo é garantida, enquanto a flexibilidade para agrupamento visual de blocos é preservada, uma vez que o reconhecimento, acompanhamento e distinção dependem de identificadores externos não relacionados às características físicas intrínsecas dos objetos.

Dentro dessa abordagem, foi necessário também definir a natureza dos identificadores visuais. A opção de utilizar imagens legíveis pelo usuário (como números e formas

geométricas) foi rejeitada devido ao risco de criar dependência excessiva da memorização, potencialmente levando ao abandono do aspecto de realidade aumentada conforme o usuário internalizasse as associações entre identificadores e trechos de código. A alternativa foi utilizar identificadores ilegíveis (códigos visuais complexos, como *QR Codes*), garantindo que o usuário mantenha o engajamento com a interface de AR para estabelecer a associação entre blocos físicos e trechos de código.

3.3.2 Objetos virtuais

3.3.2.1 Tradução de arranjo físico para código

O processo de conversão do arranjo tridimensional dos blocos em código **Python** bidimensional constituiu um desafio técnico central do projeto. A solução proposta no documento baseou-se na projeção das coordenadas tridimensionais dos blocos para um sistema de coordenadas bidimensional, seguida de normalização para uma *grid* 2D discreta.

Essa abordagem permitiria a preservação da intuitividade da manipulação física enquanto garantiria a geração de código sintaticamente correto e logicamente ordenado. A ideia era considerar tanto a ordenação vertical (linhas de código) quanto horizontal (sequência dentro de uma linha), implementando tolerâncias dinâmicas para determinar o alinhamento lógico dos blocos.

3.3.2.2 Projeção da simulação

A definição do local onde os resultados da execução do código seriam visualizados constituiu um dos desafios arquiteturais do projeto, com implicações diretas na experiência do usuário e na complexidade de implementação. Três estratégias distintas foram avaliadas, cada uma oferecendo vantagens e limitações específicas.

A primeira abordagem propunha que o usuário definisse ativamente o local de projeção através de interação direta com a tela durante a fase inicial da aplicação. Nesse sistema, o usuário tocaria na tela para indicar uma superfície específica do ambiente físico onde desejaria visualizar os resultados da simulação. Esse toque criaria um ponto de ancoragem persistente, armazenado pela aplicação e utilizado como referência para todas as projeções subsequentes. Embora essa estratégia oferecesse controle direto ao usuário e constituísse uma oportunidade pedagógica valiosa para introduzir conceitos fundamentais de realidade aumentada, apresentava uma limitação crítica: o risco de desconexão visual durante a transição entre as fases de montagem de código e visualização de resultados. O usuário poderia inadvertidamente posicionar o ponto de ancoragem fora de seu campo de visão, perdendo o início da simulação enquanto procurava reorientar a câmera para o local previamente selecionado.

A segunda alternativa, por sua vez, explorava a utilização de um bloco físico específico como indicador dinâmico do local de projeção. Nessa abordagem, o usuário manipularia um bloco dedicado exclusivamente à função de exibição da simulação do código, posicionando-o no local desejado para a visualização dos resultados. O sistema utilizaria a posição desse bloco

no momento da iniciação da simulação como referência para a projeção dos resultados. Essa solução manteria consistência conceitual com o modo de interação principal da aplicação, onde tanto a representação de trechos de código quanto a definição de local de simulação utilizariam a mesma linguagem de blocos. Contudo, essa abordagem mantinha o mesmo risco fundamental de perda visual identificado na primeira opção, uma vez que o usuário poderia inadvertidamente mover o bloco marcador para fora do enquadramento da câmera.

A terceira estratégia, efetivamente implementada na versão final do PyBlox, elimina completamente o aspecto de realidade aumentada durante a fase de visualização, direcionando a simulação diretamente à tela do dispositivo. Quando o usuário inicia a simulação, os resultados são apresentados em uma área dedicada da interface, garantindo visibilidade constante independentemente da orientação da câmera ou da posição dos blocos físicos. Embora essa decisão reduza a imersão da experiência ao abandonar a projeção em realidade aumentada, oferece vantagens práticas que justificaram sua adoção para o escopo do projeto: implementação substancialmente simplificada, garantia absoluta de visibilidade dos resultados e independência total das condições de rastreamento de AR. Essa estratégia permitiu concentrar os esforços de desenvolvimento no algoritmo de renderização da simulação propriamente dita, estabelecendo uma base sólida para futuras expansões que possam reintegrar o aspecto de realidade aumentada na visualização dos resultados. A figura 3.1 demonstra um diagrama que modela essa solução.

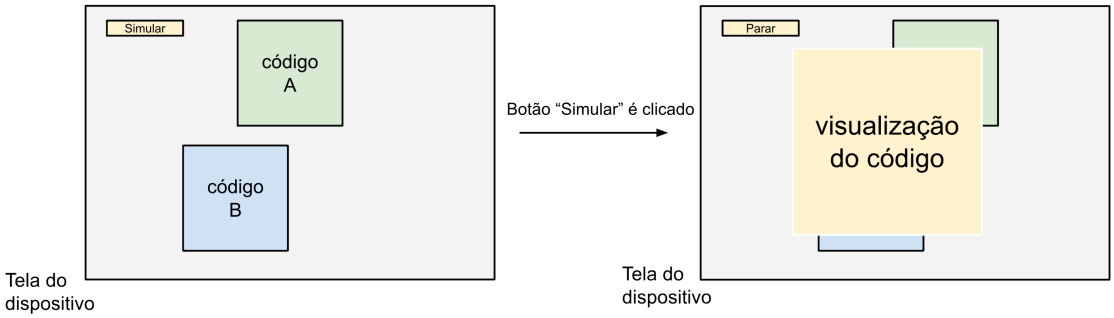

Figura 3.1: Diagrama da interface de renderização da simulação da execução do código

3.3.3 Interação com usuário

3.3.3.1 Customização de variáveis

Um dos aspectos pensados no quesito de interação com o usuário foi como permitir que usuários personalizem valores numéricos em blocos que representam variáveis, isto é, blocos cujos valores numéricos são customizáveis e podem mudar. Essa fase do design exigiu uma solução que equilibrasse precisão, usabilidade e integração com o aspecto de realidade aumentada da aplicação. Para isso, duas abordagens principais foram consideradas:

A primeira abordagem baseia-se em um sistema de interação por toque prolongado seguido de interface por deslizamento. O mecanismo requer que o usuário aponte a câmera do

dispositivo para o bloco físico desejado, posicione o dedo sobre a imagem do bloco apresentada na tela e mantenha o toque por um período predeterminado. Após esse intervalo de toque contínuo, um elemento de deslizamento virtual surge na interface, permitindo ajuste fino dos valores com atualizações em tempo real. O elemento de deslizamento permanece ativo durante a interação e desaparece automaticamente após um período de inatividade, retornando a interface ao seu estado normal.

Dentro dessa abordagem, duas variações de posicionamento da componente de deslizamento foram avaliadas. A primeira propunha posicionar o elemento de interface diretamente sobre a imagem do bloco na tela, acompanhando suas movimentações físicas através de atualizações constantes de pose. Essa solução ofereceria máxima integração visual entre o elemento físico e sua interface de controle, mas foi descartada devido à sobrecarga computacional significativa. A necessidade de recálculo contínuo da posição do elemento de interface em espaço de mundo geraria sobrecarga de processamento, potencialmente comprometendo a fluidez da experiência.

A variação escolhida posiciona o elemento de deslizamento em local fixo na tela, independente de movimentações do bloco físico. Essa estratégia elimina completamente a complexidade de atualização de pose e garante estabilidade da interface durante a interação, permitindo que o usuário ajuste valores sem preocupações com o posicionamento físico dos blocos ou com a orientação da câmera. Um diagrama dessa solução pode ser encontrado abaixo:

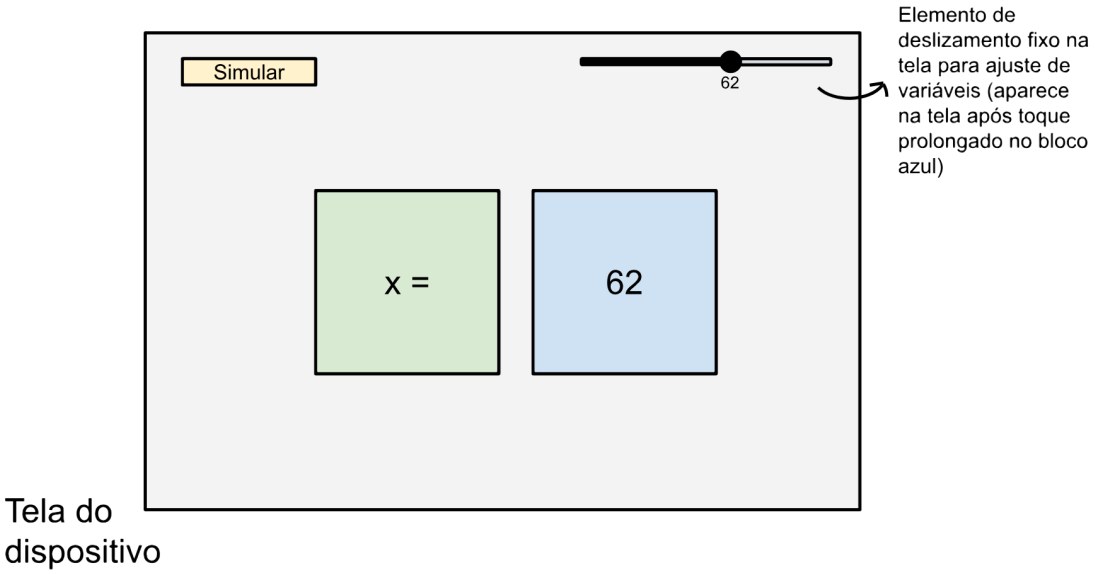

Figura 3.2: Diagrama da interface do elemento de deslizamento para ajuste de variáveis

A segunda abordagem considerada propunha a utilização de botões incrementais (+/-) que surgiriam em local fixo da tela após toque simples no bloco. O usuário poderia pressionar esses botões para incrementar ou decrementar valores gradualmente, com atualizações em tempo real. Embora essa solução oferecesse interface mais simples e implementação mais direta, foi descartada devido à limitação de velocidade de ajuste, especialmente problemática

para valores numericamente grandes que exigiriam múltiplos toques consecutivos. Um diagrama dos botões pode ser observado abaixo:

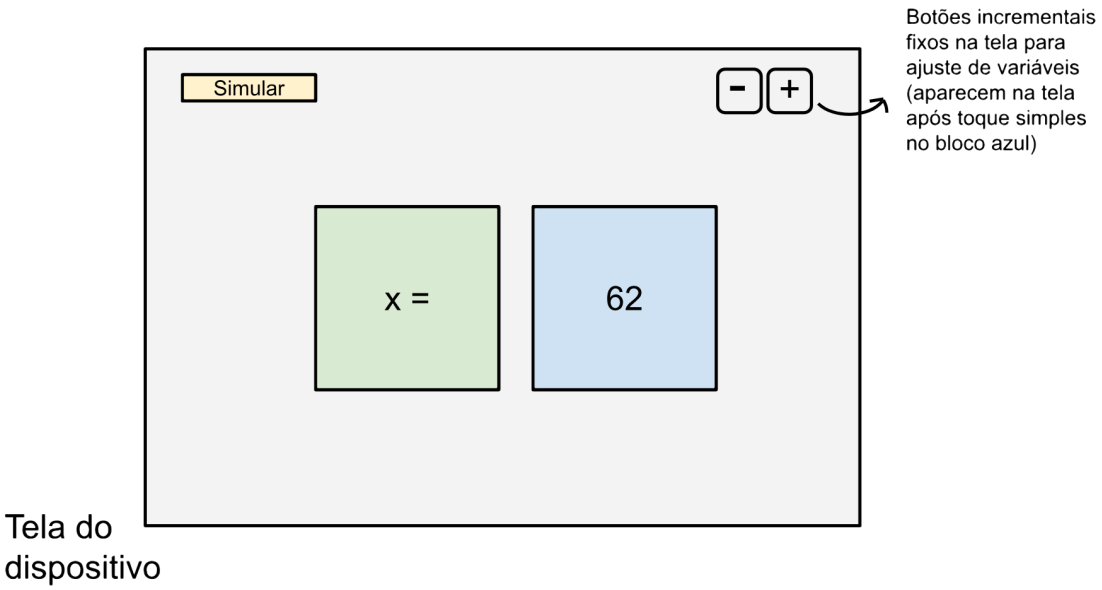

Figura 3.3: Diagrama da interface dos botões incrementais para ajuste de variáveis

3.3.3.2 Controle de simulação

Outro aspecto relacionado à interação com o usuário consistiu na definição do mecanismo de controle da execução da simulação.

A primeira estratégia, escolhida para a implementação, utiliza um botão fixo posicionado permanentemente na interface em espaço da tela. Esse elemento alterna entre os estados *Simular* e *Parar*, permitindo ao usuário iniciar e interromper a execução do código gerado a partir do arranjo de blocos. O botão permanece constantemente visível e acessível, independentemente da orientação da câmera, posição dos blocos ou condições de rastreamento de AR. Embora essa solução possa ocasionalmente obstruir parcialmente a visualização da simulação, oferece garantia absoluta de acessibilidade e simplicidade de implementação.

A segunda abordagem explorava a utilização de um bloco físico específico como controlador dedicado da simulação. Nesse sistema, o usuário manipularia um bloco identificado exclusivamente com a função de controle, tocando na tela sobre sua imagem para iniciar ou pausar a execução. Essa solução manteria consistência conceitual com a linguagem de interação baseada em blocos da aplicação e proporcionaria integração natural com o aspecto de realidade aumentada, eliminando a necessidade de elementos fixos na interface. Contudo, apresentava limitações práticas significativas: a possibilidade de toques acidentais durante a manipulação de outros blocos próximos e a necessidade de manter o bloco controlador constantemente visível no campo de visão da câmera para garantir acessibilidade da funcionalidade.

A terceira e última alternativa propunha uma solução ainda mais integrada fisicamente, utilizando um bloco com marcadores diferentes em faces opostas. Uma face apresentaria identificador visual representando *Play* enquanto a face oposta exibiria marcador correspondente a *Stop*. A orientação física do bloco controlaria diretamente o estado da simulação: posicionar a face *Play* voltada para cima iniciaria a execução, enquanto virar o bloco para expor a face *Stop* pausaria o processo. Essa abordagem ofereceria interação física através da manipulação direta do objeto, introduzindo elemento lúdico adicional através da mecânica de rotação e reduzindo a dependência de interação através da tela do dispositivo. Além disso, reforçaria conceitos pedagógicos importantes, como a noção de estados binários comum em programação. Entretanto, apresentava complexidade de implementação maior e risco de transições abruptas caso o bloco fosse girado acidentalmente durante outras manipulações.

3.3.4 Tutorial da aplicação

Por último, foram analisadas no GDD algumas formas de como estruturar o tutorial do PyBlox ao usuário, principalmente levando em conta que grande parte do público-alvo nunca teve nenhum contato prévio com programação. Duas abordagens principais foram avaliadas, cada uma representando filosofias distintas de design instrucional. Embora detalhadas no planejamento, ambas as estratégias não foram integradas à versão final da aplicação devido às limitações de tempo inerentes ao cronograma de desenvolvimento desta monografia. Contudo, seu detalhamento técnico permanece como uma boa diretriz para futuras iterações do projeto.

A primeira, que foi a escolhida como a solução preferida, adota uma estratégia de apresentação concentrada através de tela única com instruções introdutórias. Essa tela surge automaticamente durante a primeira execução da aplicação, apresentando em formato tela cheia uma explicação abrangente do funcionamento básico, objetivos pedagógicos e mecânicas de interação. Após a leitura, o usuário pode iniciar a utilização da aplicação imediatamente, com o conteúdo instrucional permanecendo acessível através de um botão de ajuda integrado à interface principal. Essa abordagem oferece simplicidade de implementação e manutenção, eliminando complexidades relacionadas a estados de tutorial ou mecanismos de ativação condicionais. Contudo, apresenta a limitação de fornecer toda a informação simultaneamente, sem progressão gradual que se integre à experiência de uso.

A segunda abordagem explorava a implementação de tutorial contextual e progressivo, utilizando dicas que surgiriam dinamicamente de acordo com as ações e progressão do usuário. Esse sistema empregaria elementos visuais discretos, denominados "bolhas informativas", posicionados próximos aos elementos relevantes de cada etapa do processo de aprendizagem. A progressão seguiria uma lógica natural baseada na detecção de ações específicas: uma instrução inicial como "Aponte a câmera para os blocos"seria automaticamente substituída por "Toque em um dos blocos"após a detecção bem-sucedida dos objetos pelo sistema de rastreamento. Subsequentemente, outras instruções surgiriam conforme o usuário demonstrasse compreensão e execução das ações anteriores, criando uma experiência de aprendizagem orgânica e prática.

Essa abordagem alternativa ofereceria vantagens pedagógicas significativas: manu-

tenção do contexto de realidade aumentada sem interrupções, aprendizado mais prático e interativo e progressão natural baseada na demonstração efetiva de competências. Entretanto, foi descartada devido a algumas considerações técnicas. A implementação exigiria complexidade substancial para gerenciar estados de tutorial, detectar ações do usuário de forma confiável e coordenar a apresentação de dicas contextuais. Mais criticamente, o sistema dependeria da qualidade consistente do rastreamento de AR para funcionar adequadamente. Atrasos entre a execução de ações pelo usuário e a detecção pelo sistema, ou falhas temporárias no rastreamento, poderiam gerar confusão e frustração, comprometendo a experiência pedagógica pretendida.

3.4 Execução e adaptação de solução

Nesta seção, serão descritos os processos de implementação do projeto, englobando desafios arquiteturais, algoritmos utilizados e cenários em que alguns aspectos planejados tiveram que sofrer modificações.

3.4.1 Implementação

3.4.1.1 Execução de código em Python

O desenvolvimento de aplicações interativas e jogos em Unity muitas vezes pode impor um desafio arquitetural relacionado à necessidade de integrar e executar funcionalidades específicas de outras linguagens de programação, que não C#. Em particular, no caso do PyBlox, o requisito de projeto de processar e executar o código Python gerado pelo usuário tornou essencial o estabelecimento de um mecanismo para a execução do Python a partir do ambiente Unity.

A abordagem inicial para solucionar essa integração debruçou-se na **execução local de um binário Python** por meio de processos externos. Essa estratégia fazia uso da classe `System.Diagnostics.Process` do namespace C#, que facilita a invocação e o gerenciamento de programas executáveis. Para testar essa abordagem, um binário do Python (`python.exe`) foi incorporado à pasta `StreamingAssets` do projeto Unity. O código C# foi configurado para invocar esse interpretador de forma assíncrona, permitindo a execução dos *scripts* Python quando o arranjo de código era gerado pela junção dos blocos feita pelo usuário.

Em uma versão de testes do PyBlox para *desktop*, essa solução demonstrou eficácia, validando a execução do código e permitindo a captura simultânea da saída padrão (`stdout`) e dos erros (`stderr`) do processo externo. Contudo, essa estratégia acabou sendo confrontada pela incompatibilidade arquitetural do sistema operacional Android, elemento central do projeto PyBlox. A arquitetura do Android, baseada em Linux, é incompatível com executáveis binários compilados para o Windows (formato `.exe`).

Além disso, houve uma tentativa de procurar por algum *plugin* da Unity que suportasse a execução de Python em dispositivos móveis Android de forma estável e distribuível. Nessa linha, foi avaliado o pacote *Python for Unity*, que oferece uma integração do Python

no ambiente de desenvolvimento da Unity. Contudo, essa alternativa demonstrou-se inviável, uma vez que o uso desse plugin é restrito ao Editor do ambiente Unity e não suporta a inclusão do interpretador Python nas versões finais das aplicações. Como o PyBlox tem o Android como principal plataforma-alvo, a impossibilidade de executar o código Python diretamente no dispositivo eliminou essa opção [5].

Diante dessas limitações, a estratégia foi reorientada para a implementação de uma arquitetura descentralizada, delegando o **processamento do código Python a um servidor remoto**. Essa decisão buscou isolar a execução do lado do cliente da Unity, garantindo confiabilidade e escalabilidade independentemente do ambiente de execução do usuário final.

Para tal, foi configurada uma plataforma *serverless* no Google Cloud, atuando como um *endpoint* HTTP dedicado. O componente principal dessa arquitetura consiste em uma função *serverless* [4], desenvolvida em Python, projetada para receber requisições no formato JSON contendo o código a ser processado. A execução do código ocorre em um ambiente isolado e seguro no servidor, e a saída resultante (incluindo qualquer registro de erro) é capturada, serializada em uma resposta JSON e enviada de volta à aplicação da Unity.

A comunicação eficiente entre o PyBlox e a função *serverless* foi estabelecida em C# através de uma arquitetura que utiliza `UniTask` [20] e a classe `UnityWebRequest` [7]. A classe `UnityWebRequest` foi escolhida por sua capacidade de gerenciar comunicação assíncrona e por garantir compatibilidade nativa com todas as plataformas de *build* suportadas pela Unity, um fator essencial para a portabilidade do projeto.

O fluxo de envio de código e recepção de resultados é estritamente baseado no protocolo HTTP e na troca de dados JSON. No método assíncrono `SendWebRequestAsync`, o código Python a ser executado é serializado no formato JSON, utilizando a classe `JsonUtility` e o modelo `PythonCodeRequest` (Requisição de Código em Python). Este JSON é então convertido em bytes e anexado como o corpo (`UploadHandlerRaw`) da requisição POST para o *endpoint* do servidor. Essa abordagem garante a integridade estrutural e o formato adequado dos dados a serem processados.

Para otimizar o fluxo de controle, a biblioteca `UniTask` foi adotada. Essa biblioteca permite o uso do padrão `async`/`await` em C#, substituindo as corrotinas nativas da Unity e resultando em um código mais limpo e eficiente para gerenciar a espera pela resposta do servidor. Após a requisição (`await request.SendWebRequest()`), o código Unity verifica o resultado da operação. Se a requisição for bem-sucedida, o corpo da resposta, contido em `request.downloadHandler.text`, é desserializado novamente via `JsonUtility` para o modelo `PythonCodeResponse` (Resposta de Código em Python). Este modelo permite o tratamento discriminado de sucesso ou falha na execução do Python (verificando o campo `success`), garantindo que tanto a saída padrão (`output`) quanto eventuais mensagens de erro (`error`) sejam corretamente extraídas e apresentadas ao usuário final.

3.4.1.2 Geração de código a partir do arranjo de blocos

O processo de geração do código em Python é fundamentado no rastreamento contínuo dos blocos. Sua detecção é realizada pelo componente `ARTrackedImageManager` (Geren-

ciador de Imagens Rastreadas em AR), que utiliza as imagens identificadoras de blocos previamente registradas no *asset* de `ReferenceImageLibrary` (Biblioteca de Imagens de Referência) para localizá-las caso estejam presentes na imagem capturada pela câmera do dispositivo do usuário.

A partir dos dados fornecidos pelo rastreamento, como a posição e a rotação de cada bloco, é realizado o primeiro mapeamento do arranjo dos **blocos no espaço tridimensional (3D) do mundo real** para o arranjo dos seus respectivos **trechos de código no espaço bidimensional (2D) da tela** do dispositivo do usuário, utilizando o método `WorldToScreenPoint` (Ponto do Mundo para Ponto da Tela) da câmera do componente `XROrigin` (Origem de Realidade Estendida) [6]. Essa transformação parte das três dimensões e finda nas duas dimensões, alinhando o arranjo de blocos com a natureza bidimensional de um arquivo de código e aproximando a disposição física da sua interpretação digital.

Entretanto, a noção lógica de ordenação entre blocos ainda não foi estabelecida e deve ser imposta de forma a simular a convenção de leitura estabelecida pela língua portuguesa: as linhas de texto progridem conforme são lidas de cima para baixo e as colunas de caracteres de uma linha progridem conforme são lidas da esquerda para direita. Portanto, o segundo e final mapeamento parte das posições dos trechos de código no espaço contínuo da tela e finda nas linhas e colunas onde estão localizados os trechos de código no **espaço discreto do arquivo de código**:

- **Espaço da tela (S)**: estabelece um sistema de coordenadas baseado em números reais $\mathbb{R}$ com origem $(0, 0)$ no canto inferior esquerdo e expansão em ambas as dimensões até o canto superior direito localizado em $(W - 1, H - 1)$, em que W é a largura e H é a altura da tela do dispositivo do usuário.

$$S = \{(x, y) \in \mathbb{R}^2 \mid 0 \leq x \leq W - 1,\ 0 \leq y \leq H - 1\}$$

- **Espaço do arquivo de código (F)**: estabelece um sistema de coordenadas baseado em números naturais $\mathbb{N}_0$ com origem $(0, 0)$ no canto superior esquerdo e expansão em ambas as dimensões até o canto inferior direito localizado em $(L - 1, C - 1)$, em que L é a quantidade de linhas e C é a quantidade de colunas no arquivo de código gerado a partir do arranjo de blocos.

$$F = \{(x, y) \in \mathbb{N}^2 \mid 0 \leq x \leq C - 1,\ 0 \leq y \leq L - 1\}$$

A lógica dessa transformação baseia-se em ordenar os trechos de código do sistema S utilizando o valor da **ordenada y como fator de comparação primário**. Por esse critério inicial, a ordenação é aplicada em ordem **decrescente**, dado que a expansão vertical do espaço S, caracterizado pelo alinhamento de valores maiores do eixo y com posições na região superior da tela, ocorre no sentido contrário do espaço F, caracterizado pelo alinhamento de valores menores da ordenada y com as primeiras linhas do arquivo.

Caso os trechos de código estejam suficientemente distantes um do outro na orientação vertical, ou seja, a diferença entre os valores da ordenada y de dois trechos seja maior do que a tolerância T_y, a ordenação primária é considerada final. Caso contrário, a ordenação primária é refinada ao utilizar o valor da **ordenada x como fator de comparação**

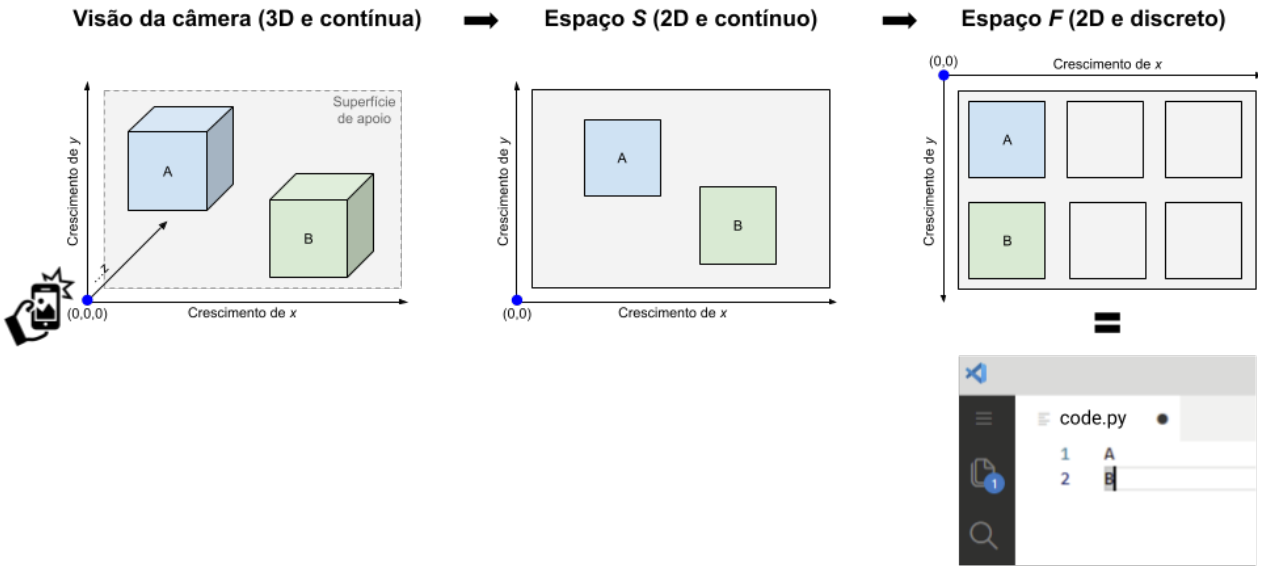

Figura 3.4: Cadeia de mapeamentos do espaço tridimensional do mundo real para espaço bidimensional S para o espaço discreto F.

secundário. Por esse critério alternativo, a ordenação é aplicada em ordem **crescente**, dado que a expansão horizontal do espaço S, caracterizado pelo alinhamento de valores maiores do eixo x com posições na região direito da tela, ocorre no mesmo sentido do espaço F, caracterizado pelo alinhamento de valores maiores do eixo x com as últimas colunas de caracteres de uma linha do arquivo.

A **tolerância da proximidade vertical** T_y determina se dois trechos de código são considerados logicamente **alinhados na mesma linha** do arquivo de código gerado. Como consequência, o critério alternativo (ordenada x como fator de comparação) é utilizado como desempate posicional e passa a definir a posição ocupada pelos trechos de código dentro da mesma linha. É primordial que o valor T_y seja dinamicamente definido e proporcional ao tamanho ocupado na tela pela imagem identificadora do bloco associado ao trecho de código, lógica explorada na seção 3.4.1.3.

3.4.1.3 Detecção de bordas de blocos

Para que o PyBlox funcione como ferramenta intuitiva e eficaz no ensino de programação em Python, o algoritmo de geração de código deve estabelecer uma interpretação computacional do arranjo de blocos capaz de simular as convenções rotineiramente seguidas pelos usuários. Por exemplo, a seção 3.4.1.2 explorou como a ordenação dos blocos foi implementada para espelhar o padrão de leitura da língua portuguesa, o qual depende tanto da direção de leitura quanto das interrupções no fluxo de leitura por quebras de linha. O primeiro fator foi replicado pela comparação entre coordenadas dos trechos de código no espaço da tela (S) e o segundo fator pela aplicação da tolerância da proximidade vertical (T_y).

Da mesma maneira que as posições das imagens identificadoras dos blocos associados aos trechos de código são dinamicamente extraídas pelo mecanismo de rastreamento, seria

ideal que suas dimensões também o fossem, uma vez que o limiar de T_y é calculado a partir da **metade da altura da imagem identificadora** reconhecida na captura da câmera. Essa lógica de definição do valor da tolerância T_y e sua utilização durante a comparação posicional entre blocos é equivalente a considerar que dois blocos estão verticalmente próximos se o centro de um dos blocos estiver simultaneamente abaixo do topo e acima da base do outro bloco, como ilustra a figura 3.5.

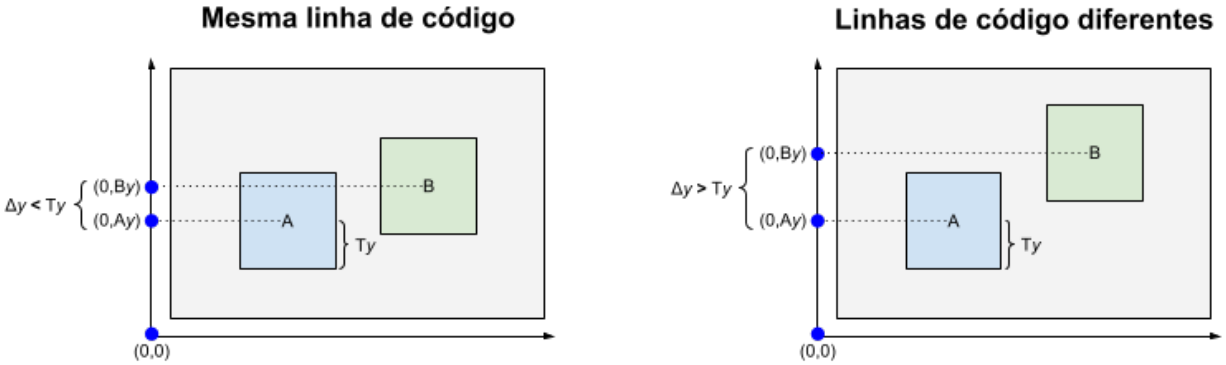

Figura 3.5: Aplicação de T_y em comparações feitas no espaço S.

Entretanto, o reconhecimento das dimensões exatas das imagens rastreadas não é uma funcionalidade nativa suportada pela *framework* AR Foundation nem pela biblioteca ARCore quando integradas à Unity. Em decorrência dessa limitação, surgiu a necessidade de complementar os mecanismos oferecidos pelas tecnologias em uso com algoritmos de Visão Computacional implementados em **Python**, baseados na biblioteca **OpenCV** e executados por uma nova função em um servidor remoto. Tal abordagem foi estrategicamente selecionada por permitir a reutilização do projeto criado na plataforma **Google Cloud** e das configurações de comunicação entre o PyBlox e o serviço de computação em nuvem, previamente estabelecidos para a execução de código em Python, conforme explorado na seção 3.4.1.1. Como resultado, o projeto demandou o desenvolvimento de dois novos componentes.

3.4.1.3.1 Função remota no Google Cloud Run é responsável por detectar bordas de quadriláteros na imagem da câmera. Após ser invocada pelo recebimento da requisição, a função decodifica a imagem da captura de câmera para o formato suportado pela biblioteca `cv2` da OpenCV, e realiza seu pré-processamento em duas etapas antes de efetivamente iniciar a detecção de bordas.

Inicialmente, a imagem é convertida do espaço de cores BGR (iniciais das cores primárias azul, verde e vermelho em inglês) para a **escala cinza**. Esta primeira etapa maximiza a **eficiência** de algoritmos de detecção de bordas, pois permite que a computação atue sobre a variação de intensidade de um único canal de *pixels*, em contraste com um canal de *pixels* para cada uma das cores primárias. A seguir, um **filtro Gaussiano** [18] com um *kernel* de dimensão de 5×5 *pixels* é aplicado à imagem em escala cinza. Esta segunda etapa maximiza a **eficácia** do algoritmo de detecção de bordas ao suavizar texturas e minimizar ruído, prevenindo o reconhecimento de bordas falsas por meio do efeito de desfoque.

A detecção de bordas em si é realizada pelo **algoritmo de Canny** [14], que utiliza

dois limiares ajustados empiricamente ($L_inf = 50$ como inferior e $L_sup = 110$ como superior) para identificar bordas caracterizadas da seguinte forma:

- **Bordas fortes**: compostas por *pixels* cujo gradiente de intensidade é superior a L_sup, que são imediatamente classificados como **válidos**.
- **Bordas fracas**: compostas por *pixels* cujo gradiente de intensidade é inferior a L_inf, que são considerados ruídos **inválidos**.
- **Bordas conectadas**: compostas por *pixels* cujo gradiente de intensidade é inferior a L_sup, mas superior a L_inf, que são classificados como **válidos somente se estiverem conectados** a um *pixel* pertencente à borda forte.

O resultado do algoritmo de Canny é uma imagem binária, cujos *pixels* brancos (1) são os componentes das bordas válidas (fortes e conectadas) e os pretos (0) são componentes das bordas inválidas (fracas). Sobre esta imagem é aplicada a **transformação morfológica de *closing*** (fechamento) [17] com *kernel* de dimensão 5×5 *pixels*, que consiste em uma dilatação seguida de erosão [13]. Esta operação é crucial para conectar pequenas rupturas nas bordas detectadas e garantir a continuidade dos contornos dos quadriláteros.

- **Dilatação**: um *pixel* da imagem resultante será branco se pelo menos um *pixel* dentro da área do *kernel* também for.
- **Erosão**: um *pixel* da imagem resultante será branco se todos os *pixels* dentro da área do *kernel* também forem.

Posteriormente, a função `findContours` da OpenCV [15] extrai a **representação vetorial das bordas detectadas** e rasterizadas na imagem binária pelo algoritmo de Canny, fornecendo as coordenadas dos *pixels* pertencentes às bordas válidas. Para minimizar o consumo de memória, duas estratégias de simplificação são aplicadas. A primeira consiste em armazenar apenas dados referentes aos **contornos mais externos** [16], por exemplo, bordas de elementos intrínsecos às imagens identificadoras dos blocos são ofuscadas pela borda criada pelo contraste entre o bloco e a superfície de apoio. Já a segunda consiste em **excluir coordenadas consideradas redundantes** para definição de um contorno, por exemplo, pontos intermediários de uma linha reta são eliminados e representados apenas pelos pontos de suas extremidades.

Por fim, cada borda vetorial extraída passa pelo processo de **aproximação poligonal** através do **algoritmo de Douglas-Peucker** [12], que reduz a quantidade de vértices de uma borda utilizando o limiar de precisão $T_{DP} = 0,02 \times P$. Nesta fórmula, P é o perímetro da borda em análise, garantindo que a simplificação seja mais robusta e adaptativa. Após o processamento, apenas as bordas cuja aproximação poligonal resulta em um **quadrilátero** são mantidas, corroborando a premissa de que os identificadores dos blocos possuem formato quadrangular.

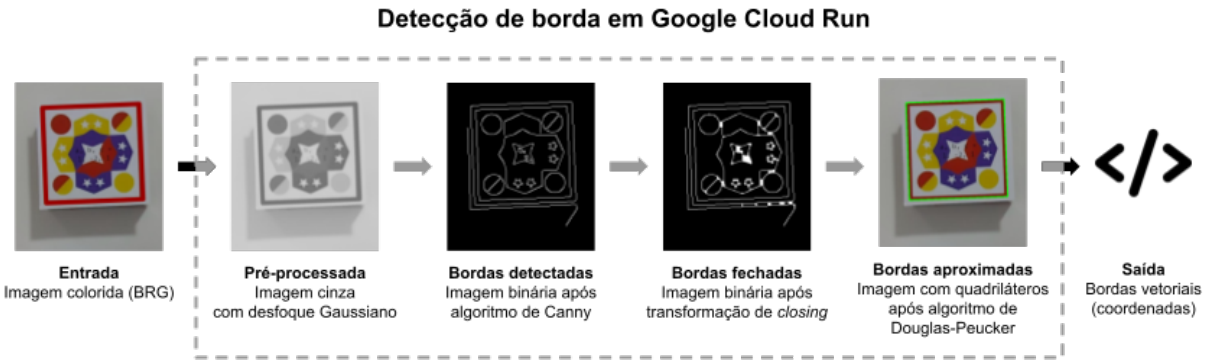

Figura 3.6: Cadeia de transformações para detecção de bordas na função remota no Google Cloud Run.

A sequência de transformações realizadas pelos algoritmos de Visão Computacional foi inspirada por um tutorial de identificação de capas de livros em imagens utilizando a linguagem de programação Python e a biblioteca OpenCV [9]. Esse guia pode ser aplicado ao contexto do PyBlox pois, assim como livros, blocos assumem o formato de quadriláteros quando vistos pelo ponto de vista *top-down* (de cima para baixo).

3.4.1.3.2 Classe auxiliar no projeto da Unity é responsável por orquestrar o fluxo de dados entre a aplicação na Unity e o Google Cloud. O preparo dos dados de entrada consiste em capturar a visão atual da câmera sem os elementos da interface de usuário, como botões e divisões de tela. Esta imagem é codificada no formato JPG, armazenada como *array* de *bytes* e enviada de forma assíncrona ao *endpoint* do servidor remoto através de uma requisição HTTP de método POST, utilizando a função `SendWebRequestAsync`.

O resultado da computação realizada em nuvem, se bem-sucedida, é uma coleção de bordas aproximadas, cada uma definida pela lista de coordenadas dos quatro vértices que a compõem. O arremate do fluxo de dados ocorre quando a resposta é recebida codificada no formato JSON e desserializada via `JsonUtility` para o modelo `BorderDetectorResponse`.

A classe também se encarrega do pós-processamento das bordas detectadas, realizando o **pareamento** destas com os blocos rastreados pelo sistema. O objetivo é descartar detecções acidentais e validar detecções úteis utilizando o contexto do sistema. Para tal, é fundamental o mapeamento inicial das coordenadas dos vértices delimitadores de bordas, pertencentes ao espaço de imagens representadas pela OpenCV (O), para o espaço de imagens representadas pela Unity, equivalente ao espaço da tela (S), previamente definido na seção 3.4.1.2:

- **Representação de imagens por OpenCV (O)**: estabelece um sistema de coordenadas baseado em números naturais $\mathbb{N}_0$ com origem $(0, 0)$ no canto superior esquerdo e expansão em ambas as dimensões até o canto inferior direito localizado em $(W-1, H-1)$, em que W é a largura e H é a altura da tela do dispositivo do usuário. Sua transformação para o espaço S, denominada T, é necessária devido aos sentidos contrários de expansão da ordenada y.

$$T : O \to S, \quad (x, y) \mapsto (\, x,\ \mathrm{H} - 1 - y \,)$$

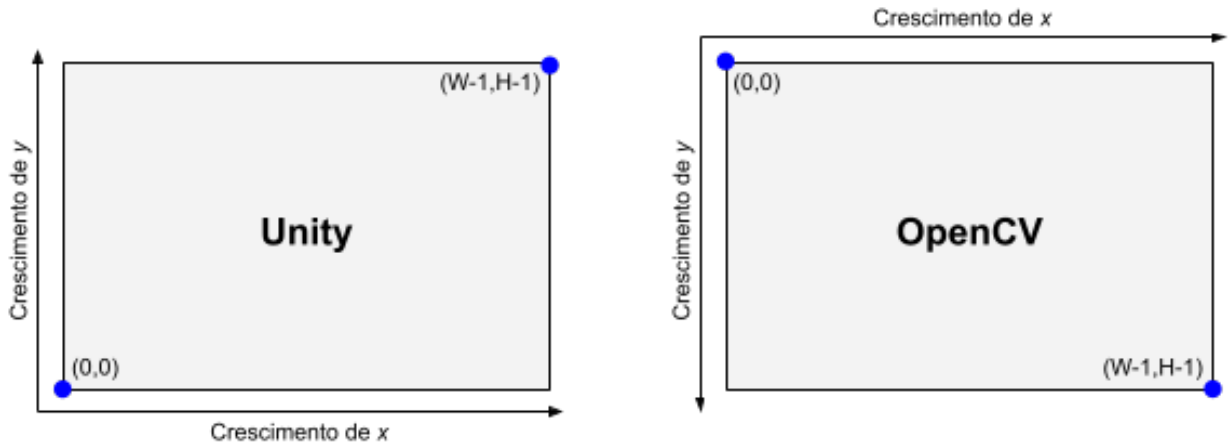

Figura 3.7: Comparação entre os sistemas de coordenadas utilizados na representação de imagens pelas aplicações Unity (espaço S) e OpenCV (espaço O).

A seguir, cada borda é comparada a cada bloco rastreado até que os critérios de encaixe sejam cumpridos:

1. $TopLeft_x \leq Center_x$ `AND` $TopLeft_y \geq Center_y$, em que $TopLeft$ é a coordenada do vértice superior esquerdo da borda e $Center$ é a coordenada do centro do bloco (posição rastreada).
2. $BottomRight_x \geq Center_x$ `AND` $BottomRight_y \leq Center_y$, em que $BottomRight$ é a coordenada do vértice inferior direito da borda e $Center$ é a coordenada do centro do bloco (posição rastreada).

Devido à sensibilidade do algoritmo, é esperado que nem toda borda detectada pertença a um bloco rastreado, podendo pertencer a um objeto quadrangular alheio ou a uma detecção incompleta de uma imagem identificadora. Por esta razão, a tolerância T_y é calculada de forma única para cada bloco, com dois possíveis cenários:

- **Bloco pareado com sucesso a uma borda**: seu T_y é definido como a metade da média do comprimento dos lados da borda, calculado a partir da distância entre dois vértices adjacentes (delimitam um lado).
- **Bloco não pareado a nenhuma borda**: seu T_y é definido como a metade da média dos valores de T_y calculados para os blocos que foram pareados às suas bordas com sucesso.

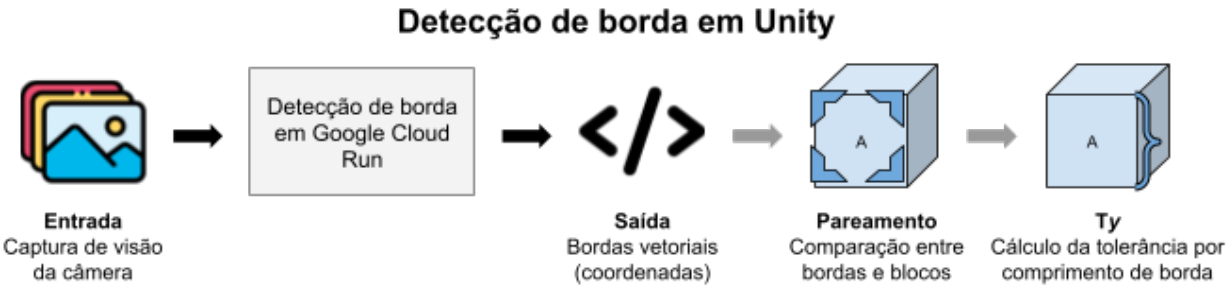

Figura 3.8: Cadeia de transformações para detecção de bordas na classe auxiliar no projeto da Unity.

3.4.1.4 Simulação de código executado

A definição do escopo de programação no PyBlox priorizou códigos cujos resultados na saída padrão (`stdout`) apresentassem relevância pedagógica. Especificamente, o projeto em sua versão final direciona-se a códigos cuja saída seja uma sequência de **valores numéricos plotáveis** (números de ponto flutuante), o que permite a visualização de conceitos como progressão de variáveis, sequências matemáticas ou coordenadas. A simulação consiste em interpretar essa saída e representá-la visualmente em uma *grid* 2D, permitindo que o usuário observe o efeito concreto do código programado por ele. Essa possibilidade de visualização concreta dos efeitos do código criado consolida ainda mais o aprendizado da lógica de programação.

O design arquitetural para o gerenciamento da execução de código e da subsequente simulação é baseado em um padrão **Orientado a Eventos**, utilizando a classe `UnityEvent` da Unity. Essa abordagem desacopla o módulo `PythonExecutor` (Executor de Python), responsável pela execução, do módulo `PythonSimulationController` (Controlador da Simulação de Python), responsável pela visualização, garantindo flexibilidade e facilitando futuras expansões.

O `PythonExecutor` define quatro eventos estáticos:

- `OnPythonExecutionInitiated` (Execução de Python Iniciada): disparado antes do envio da requisição HTTP ao Google Cloud Run, informando qual código foi enviado.
- `OnPythonExecutionCompletedSuccessfully` (Execução de Python Finalizada com Sucesso): disparado após o recebimento de uma resposta bem-sucedida do servidor, contendo a saída (`output`) do código Python.
- `OnPythonExecutionCompletedWithFailure` (Execução de Python Finalizada com Falha): disparado em caso de falha de rede ou erro de execução do Python (`error`), fornecendo a mensagem de erro formatada.
- `OnClearPythonExecution` (Limpeza de Execução de Python): utilizado para solicitar o *reset* da simulação ou de outros elementos de UI.

O `PythonSimulationController` atua como um *listener* (ouvinte) central, subscrevendo ao evento `OnPythonExecutionCompletedSuccessfully` no método `Awake`. Após a

detecção de uma execução bem-sucedida, o controlador inicia a *pipeline* de simulação através do método `HandlePythonExecutionComplete`, que chama a função `ParseOutput` para processar a *string* de saída e, em seguida, inicia uma `Coroutine` (`CreateSimulationPoints`) para a visualização animada dos dados.

3.4.1.4.1 Algoritmo de mapeamento e visualização O processo de simulação de código executado envolve duas etapas principais dentro do `PythonSimulationController`: o **processamento da saída** e o **mapeamento matemático para coordenadas 2D**.

1. Processamento da saída (`ParseOutput`) O código C# espera que a saída padrão do Python (`output`) consista em múltiplas linhas, onde cada linha representa um valor numérico a ser plotado. O método `ParseOutput` realiza:

- **Divisão por linha:** a *string* de saída é dividida em um *array* de linhas.
- **Tentativa de parse:** para cada linha, é realizada uma tentativa de conversão para o tipo `float` (ponto flutuante) usando `float.TryParse`.
- **Coleta de dados:** os valores válidos são armazenados na lista `parsedValues`, que servirá como a fonte de dados para a plotagem.

2. Mapeamento matemático (`CreateSimulationPoints`) O objetivo do mapeamento é transformar o conjunto de dados (índice i e valor Y_i) em coordenadas (P_x, P_y) no sistema de coordenadas da tela de simulação. Isso garante que os pontos sejam plotados de forma consistente, preenchendo o espaço disponível no `backgroundImage` de maneira proporcional.

Primeiramente, são estabelecidas as dimensões da área de plotagem, aplicando-se um *padding* definido por `PaddingPercentage` ($P = 0.1f$) para que os pontos não toquem as bordas do plano de fundo.

Seja:

- W: Largura do `backgroundImage` (`bgWidth`)
- H: Altura do `backgroundImage` (`bgHeight`)
- P: Porcentagem de *padding* (`PaddingPercentage`)

A área de plotagem é definida por:

- Largura interna: $W' = W \times (1 - 2P)$ (`paddedWidth`)
- Altura interna: $H' = H \times (1 - 2P)$ (`paddedHeight`)
- Offset X: $O_x = W \times P$ (`xOffset`)

- Offset Y: $O_y = H \times P$ (`yOffset`)

O número total de pontos a plotar é N, dado por `parsedValues.Count`. O valor de cada ponto no eixo Y é $Y_i = \texttt{parsedValues}[i]$, com $i \in \{0, 1, \ldots, N-1\}$. Os limites dos dados são $Y_{\min} = \min(Y_i)$ e $Y_{\max} = \max(Y_i)$, e a amplitude é $\Delta Y = Y_{\max} - Y_{\min}$.

Para cada ponto i, a transformação em coordenadas (P_x, P_y) segue as seguintes etapas:

a) Mapeamento do eixo X (Índice) A coordenada P_x é determinada pelo índice do ponto i, representando a progressão temporal ou sequencial.

$$\text{Normalized}_X = \frac{i}{N-1} \quad \text{para } N > 1$$

$$P_x = (\text{Normalized}_X \times W') + O_x - \frac{W}{2}$$

O termo $\frac{W}{2}$ é subtraído para compensar a origem do sistema de coordenadas do `RectTransform` da Unity, que é centralizada.

b) Mapeamento do eixo Y (Valor) A coordenada P_y é determinada pelo valor do dado Y_i, representando a magnitude.

Primeiro, o valor é normalizado. Um caso especial é tratado: se todos os valores forem idênticos ($\Delta Y = 0$), a normalização é fixada em 0.5 para centralizar os pontos verticalmente na área de plotagem. Caso contrário, a normalização é feita linearmente:

$$\text{Normalized}_Y = \begin{cases} 0.5 & \text{se } \Delta Y = 0 \\ \frac{Y_i - Y_{\min}}{\Delta Y} & \text{se } \Delta Y > 0 \end{cases}$$

Em seguida, o valor normalizado é mapeado para a coordenada de tela:

$$P_y = (\text{Normalized}_Y \times H') + O_y - \frac{H}{2}$$

Finalmente, os pontos (`pointPrefab`) são instanciados e posicionados em (P_x, P_y) dentro do *canvas* de simulação, com um pequeno atraso (`delayPerPoint`) entre cada ponto, criando um efeito visual de **animação sequencial**. Um exemplo de simulação pode ser verificado na imagem 4.4.

3.4.2 Interface

3.4.2.1 Imagens usadas como identificadores de blocos

A fase de conceptualização do PyBlox estabeleceu a necessidade de utilizar identificadores únicos e com baixo potencial de memorização pelos usuários, cujas motivação e veredito foram formalizadas no *Game Design Document* e exploradas na seção 3.3. A proposta aceita foi inspirada pelo uso difundido de *QR Codes*, código de barras bidimensional

criado em 1994 pela empresa japonesa Denso Wave, como marcadores capazes de acionar experiências de AR. Sua vasta aplicabilidade abrange diversos setores, como o educacional, para projeção de modelos biológicos integrados a livros didáticos; o comercial, para simular como móveis complementariam ambientes decorados; e o artístico, para criação de instalações e obras interativas.

Em busca de adaptar esta referência ao escopo do PyBlox, o primeiro ciclo de criação concentrou-se na criação de cinco identificadores de bloco utilizando uma grade de dimensão 3×3 para gerar imagens binárias (compostas estritamente por *pixels* pretos e brancos). O conjunto resultante, intitulado **"minimalista"**, impôs como restrição que todas as imagens fossem únicas e não pudessem ser derivadas umas das outras por rotações ou inversões.

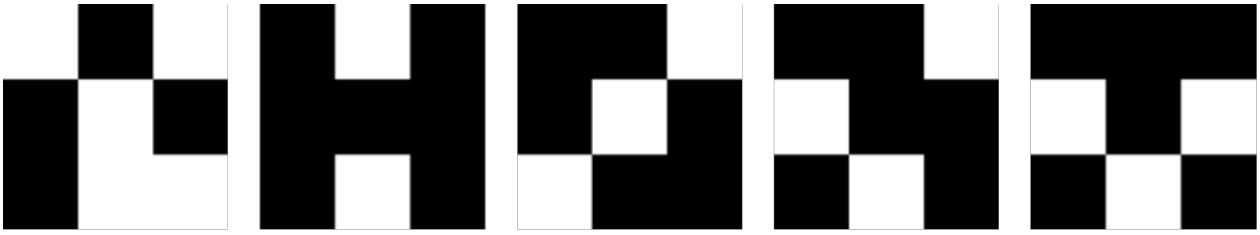

Figura 3.9: Conjunto de identificadores **"minimalista"**.

Contudo, ao tentar integrar tais identificadores na cena virtual montada para testes internos na Unity, foi descoberto um erro durante a inicialização do *asset* de `ReferenceImageLibrary`, causado pela insuficiência de *Feature Points* encontrados nas imagens carregadas na biblioteca. Pesquisas subsequentes sobre a criação de identificadores ideais indicaram a utilização de um *script* de avaliação de imagens, fornecido pelos desenvolvedores da biblioteca ARCore [3], que serviu como guia metodológico para as próximas iterações.

O segundo ciclo de criação teve como objetivo elevar a complexidade visual do conjunto "minimalista", de modo a alcançar o nível de detalhe necessário para o funcionamento adequado das funcionalidades de reconhecimento de imagens. Ainda assim, o novo conjunto, intitulado de **"*remake* minimalista"**, manteve os aspectos visuais dos *QR Codes*, como bicoloração e geometricidade. Apesar de ter performado satisfatoriamente na avaliação pelo *script* e nos testes da cena virtual, os novos identificadores demonstraram problemas significativos de latência e precisão nos primeiros testes com a câmera de um dispositivo Android no ambiente real. A projeção dos trechos de código sobre os identificadores de blocos associados a eles apresentava um atraso indesejável, causando disrupção na experiência do usuário. Além disso, a limitação a apenas duas cores e a predominância de formas quadrangulares fez com que os identificadores fossem confundidos pelas funcionalidades de rastreamento de imagens, culminando na projeção descasada de trechos de código sobre identificadores incorretos.

No terceiro ciclo de criação, a consciência de que obter métricas elevadas na avaliação pelo *script* era condição necessária, mas não suficiente para a interação efetiva da biblioteca de imagens com as ferramentas de AR, direcionou a estratégia. Para ampliar o espectro de variações e mitigar a intercambialidade por equívoco entre as imagens, foram empregadas capas de álbuns musicais modificadas por um efeito *pixelado*. Este novo conjunto, intitulado de **"álbuns"**, combinava a diversidade visual e a composição de cores vibrantes, fator de destaque utilizado na indústria musical e comercial, com a geometricidade do efeito *pixelado*,

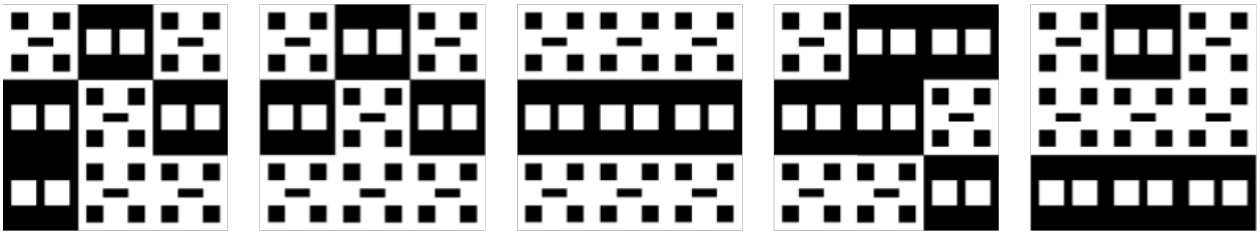

Figura 3.10: Conjunto de identificadores **"*remake* minimalista"**.

que mantém a alusão aos *QR Codes*. Esse arranjo resultou em uma excelente performance em ambos os testes virtuais e reais.

Figura 3.11: Conjunto de identificadores **"álbuns"**.

O bom desempenho, entretanto, mostrou-se efêmero com a introdução da funcionalidade de detecção de bordas, descrita na seção 3.4.1.3. Como o algoritmo de visão computacional baseia-se na variação de intensidade entre *pixels* para a identificação de bordas quadrilaterais, a detecção excessiva de bordas internas, devido ao efeito *pixelado*, comumente ofuscava a detecção de bordas externas, que envolvem por inteiro os identificadores. Esse fenômeno eleva o consumo de memória e dificulta a medida precisa das dimensões dos identificadores na captura da câmera, desencadeando a necessidade de mais um ciclo de criação.

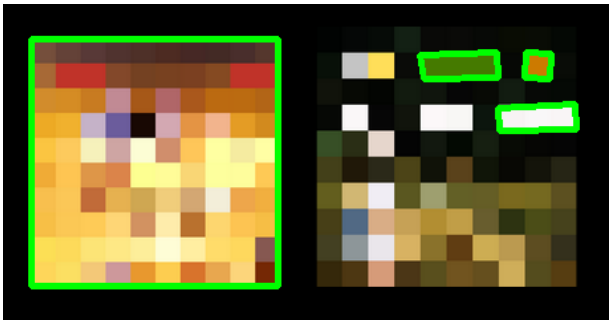

Figura 3.12: Detecção de bordas internas ofuscando bordas externas do conjunto de identificadores **"álbuns"**.

Diante da falha persistente dos identificadores inspirados em *QR Codes*, o quarto conjunto foi elaborado de modo a espelhar a estética de *identicons*, representações imagéticas de valores de *hash* criadas em 2007 por Don Park. Essa mudança de perspectiva no processo de criação coexistiu com o obedecimento de restrições reveladas nas iterações anteriores, resultando no conjunto de identificadores intitulado de **"*identicons*"**. Sua composição, que incluía formas geométricas não-quadradas e um esquema de cores singular para cada imagem, contribuiu para a obtenção de excelentes métricas na avaliação de *Feature Points* pelo script

e evitou a interferência de elementos intrínsecos das imagens nos resultados da detecção de bordas. Surpreendentemente, este conjunto foi reprovado nos testes reais em virtude da alta simetria das imagens em relação a pelo menos um de seus eixos. Como consequência, a funcionalidade de rastreamento era incapaz de identificar estavelmente a rotação assumida pelos identificadores, causando *glitches* na projeção dos trechos de código devido à orientação instável a cada ciclo de renderização da aplicação.

Figura 3.13: Conjunto de identificadores **"*identicons*"**.

Por fim, a elaboração do quinto e final conjunto, intitulado de **"*half-and-half*"** evitou o obstáculo da simetria ao criar duas variações de cores para cada identificador do conjunto "*identicons*" e utilizar a metade, repartida na diagonal, de cada uma das variações na composição dos identificadores definitivos.

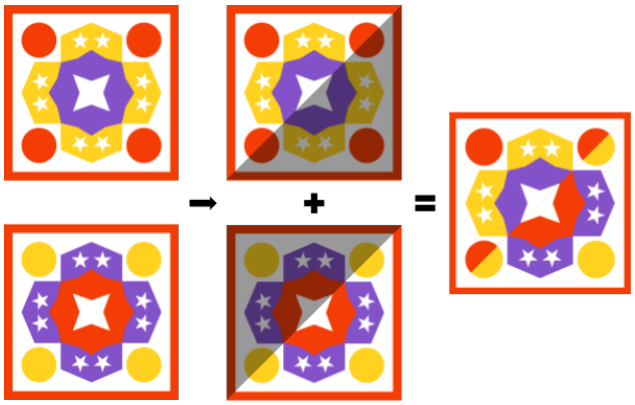

Figura 3.14: Composição do conjunto de identificadores **"*half-and-half*"** utilizando metades diagonais das variações de cores dos identificadores "*identicons*".

Esta estratégia reaproveitou a padronagem visual do conjunto anterior ao mesmo tempo que introduziu uma característica para ser utilizada como marcador de orientação. Como consequência, os identificadores "*half-and-half*" foram os mais bem sucedidos em todos os experimentos: na avaliação de *Feature Points* e em ambos os testes virtual e real.

Figura 3.15: Conjunto de identificadores **"*half-and-half*"**.

3.4.2.2 Separação da tela

A evolução da interface do usuário do PyBlox foi orientada por considerações de usabilidade e clareza pedagógica, resultando em uma arquitetura visual que prioriza a compreensão intuitiva do fluxo de trabalho da aplicação. A versão do produto inicial da interface apresentava uma abordagem mais minimalista, onde a tela era integralmente ocupada pela visualização da câmera em tempo real, permitindo ao usuário observar continuamente o ambiente físico e os blocos sendo manipulados. Quando o usuário acionava o botão de simulação, o resultado da execução do código Python era diretamente sobreposto à imagem da câmera, criando uma camada de informação textual sobre a captura visual. Essa interface pode ser visualizada na figura 4.2.

Contudo, essa implementação inicial levantou problemáticas durante os testes de usabilidade realizados com usuários sem conhecimento prévio em programação. A sobreposição direta dos resultados da simulação sobre a imagem da câmera gerava confusão visual e dificultava a compreensão do processo de transformação dos blocos físicos em código executável. Os participantes dos testes demonstraram dificuldade em estabelecer a conexão lógica entre o arranjo físico dos blocos, o código Python resultante, que não era explicitamente renderizado, e a saída da execução, elementos fundamentais para o processo de aprendizagem proposto pela aplicação.

Diante dessas observações, foi implementada uma reestruturação completa da interface, fundamentada na segmentação lógica e visual das diferentes etapas propostas pela aplicação. A nova arquitetura divide a tela em três seções distintas e funcionalmente especializadas, ativadas quando o usuário aciona o comando de simulação.

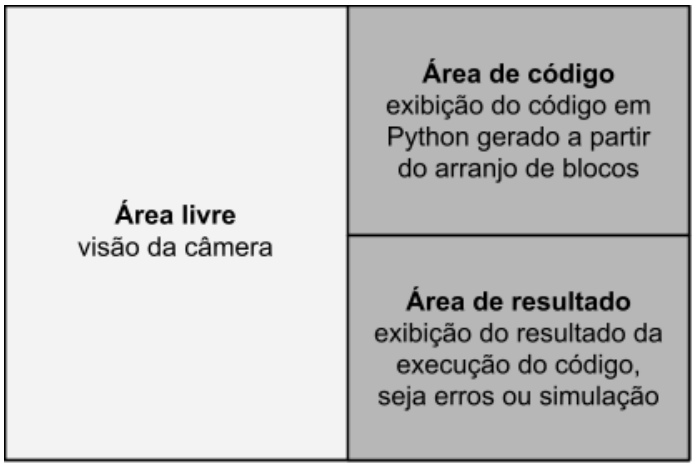

Figura 3.16: Resultado da reformulação de separação da tela.

A seção esquerda da tela mantém a **visualização da câmera**, preservando a conexão visual do usuário com o ambiente físico e permitindo ajustes contínuos no arranjo dos blocos. Essa área ocupa aproximadamente dois terços do espaço total da tela, garantindo que o contexto físico permaneça acessível durante todo o processo de simulação.

A seção direita da tela foi subdividida em duas áreas funcionalmente complementares. A área superior direita é dedicada à **exibição do código Python** gerado a partir do

arranjo de blocos detectado. Essa funcionalidade surgiu como resposta direta às dificuldades observadas nos testes iniciais, onde os usuários não conseguiam compreender como seus arranjos físicos eram interpretados pelo sistema. A visualização explícita do código intermediário estabelece uma ponte conceitual essencial entre a manipulação física e a execução digital, permitindo que o usuário valide se sua intenção foi corretamente interpretada pelo algoritmo de detecção e ordenação de blocos.

A área inferior direita é reservada para a **renderização da simulação visual dos resultados da execução do código**. Conforme descrito na seção dedicada à simulação da aplicação, essa área apresenta a plotagem dos valores numéricos gerados pelo código Python em uma *grid* 2D, proporcionando *feedback* visual imediato sobre o comportamento do programa construído. A interface da nova estruturação de tela pode ser vista na figura 4.4.

Essa segmentação tripartite da interface não apenas resolve as questões de usabilidade identificadas nos testes iniciais, mas também alinha a experiência do usuário com o fluxo pretendido: de manipulação física à interpretação em código à execução e visualização de resultados. A separação visual clara entre essas etapas facilita a compreensão do processo de **programação como uma sequência lógica de transformações**, contribuindo para o objetivo educacional da aplicação de tornar tangíveis os conceitos abstratos da lógica de programação.

Capítulo 4

Resultados

4.1 Versão inicial

Na versão inicial do projeto, o intuito foi criar uma interface mais minimalista, que permitisse a verificação e validação das *features* mais cruciais da aplicação. Nesta versão, o resultado da execução do código em Python montado pelo usuário era renderizado diretamente na tela do dispositivo, sobrepondo a visualização da câmera. As figuras abaixo demonstram exemplos da interface em cenários de falha e sucesso de execução, respectivamente:

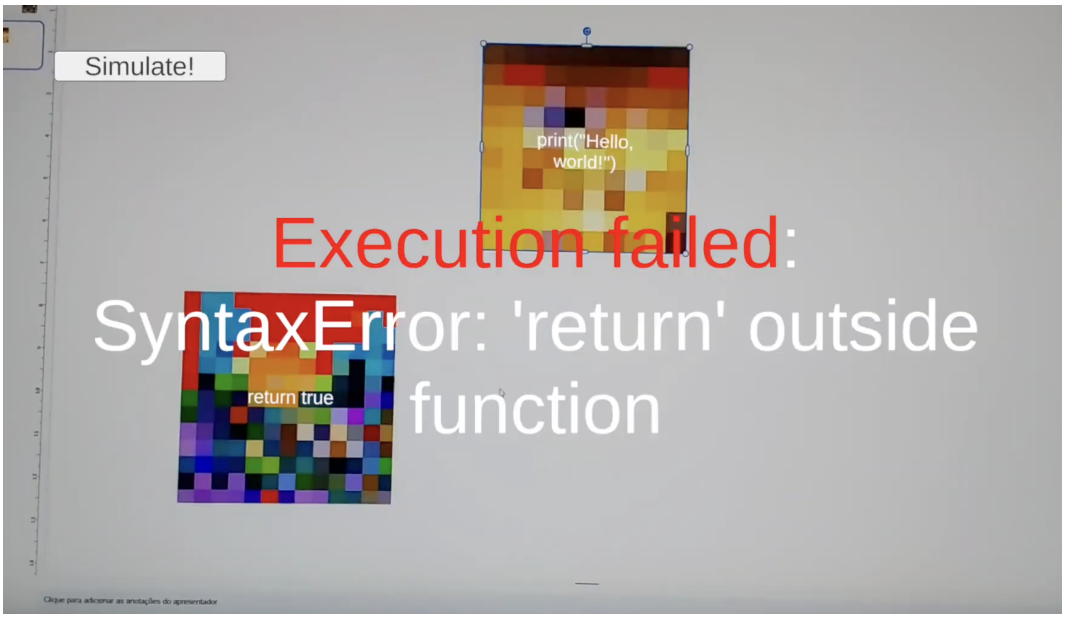

Figura 4.1: Tela do dispositivo mostrando os blocos com pedaços de código em camadas sobrepostas a eles, com o resultado de uma execução falha do Python sobrepondo centralmente a visualização.

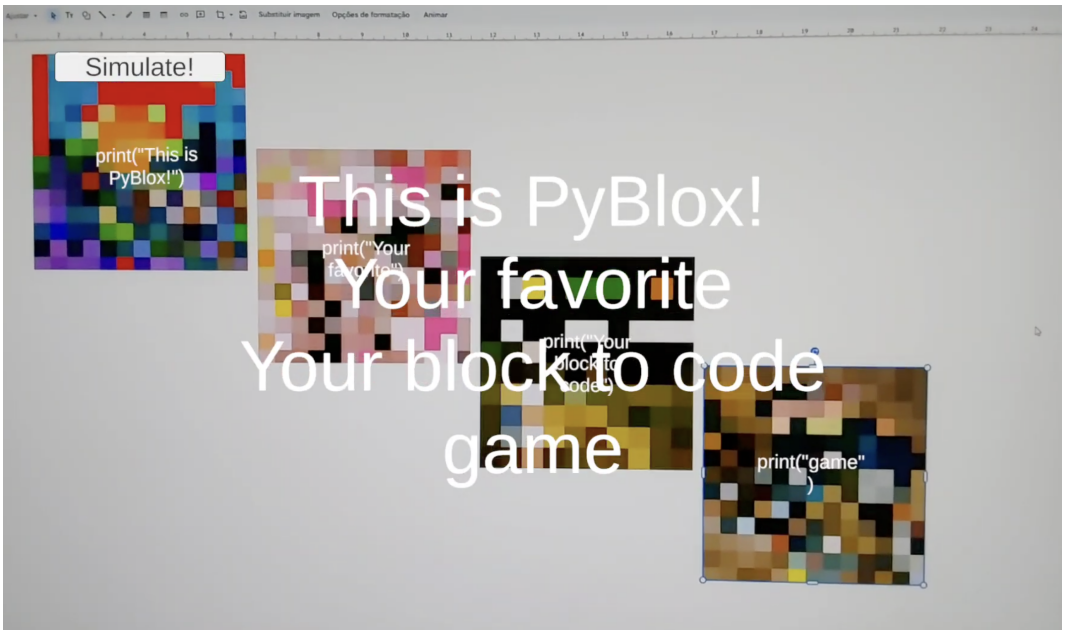

Figura 4.2: Tela do dispositivo mostrando os blocos com pedaços de código em camadas sobrepostas a eles, com o resultado de uma execução bem-sucedida do Python sobrepondo centralmente a visualização.

A interface da versão inicial demonstrou-se relativamente confusa para os usuários que participaram da gama inicial de testes do PyBlox. Para pessoas que não tinham conhecimento prévio de programação, a proposta do sistema e a interpretação dos resultados de execução não estavam suficientemente claras.

Esses resultados preliminares indicaram a necessidade de uma revisão completa do *design* da interface, buscando melhorar a usabilidade e a experiência do usuário. A seção a seguir apresenta o desenvolvimento da versão final da aplicação, que buscou mitigar esses problemas por meio de um *design* de tela mais segmentado e informativo.

4.2 Versão final

Na versão final da aplicação, foi adotado um modelo de interface segmentada no lugar da sobreposição mostrada na seção anterior. O objetivo principal foi criar um ambiente de aprendizado mais informativo, separando claramente a seção da câmera e a montagem dos blocos (mundo real) dos resultados e *feedback* do sistema (mundo digital).

Ao acionar o botão de simulação, a lateral direita da tela é segmentada horizontalmente em dois *canvas* distintos:

1. **Superior direito**: exibe o **código Python** gerado a partir da sequência de blocos montada pelo usuário.
2. **Inferior direito**: apresenta o **resultado da execução** do código gerado, seja através de uma simulação gráfica (em caso de sucesso) ou de uma mensagem detalhada de erro.

O funcionamento e o resultado final desta nova arquitetura são demonstrados nas Figuras 4.3 a 4.6, que ilustram cenários de sucesso e falha. Para os testes apresentados, foram utilizados dois blocos (vermelho e azul) com o seguinte mapeamento de código:

1. **Bloco vermelho** mapeado ao seguinte trecho de código: `a, b, c = 1, -2, 1`
2. **Bloco azul** mapeado ao seguinte trecho de código: `print('\n'.join([f"a*x**2 + b*x + c"for x in range(-9, 12)]))`

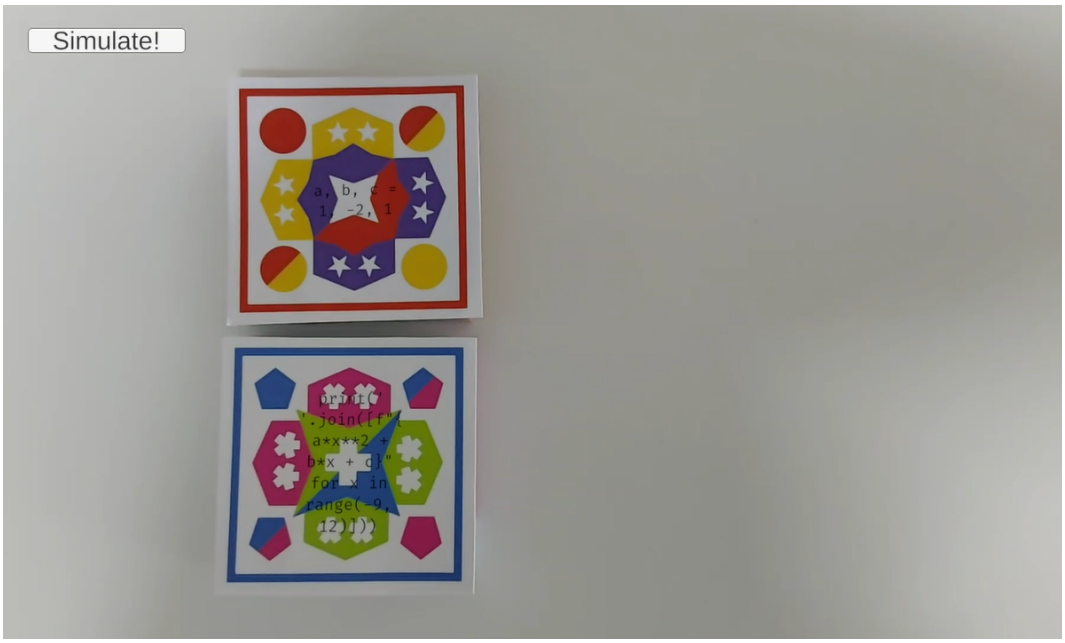

Figura 4.3: Disposição dos blocos de código que irá gerar uma execução bem-sucedida.

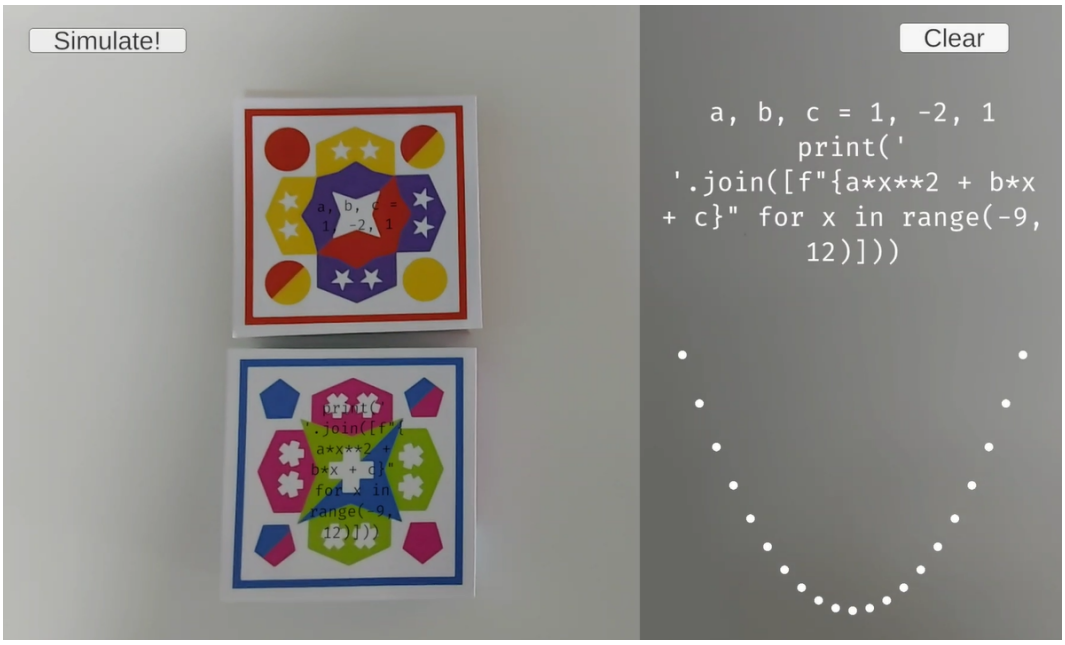

Figura 4.4: Interface após a execução bem-sucedida. O lado direito é segmentado, exibindo o código Python gerado (superior) e a simulação gráfica do resultado do código (inferior).

Figura 4.5: Disposição dos blocos de código que irá gerar uma execução falha.

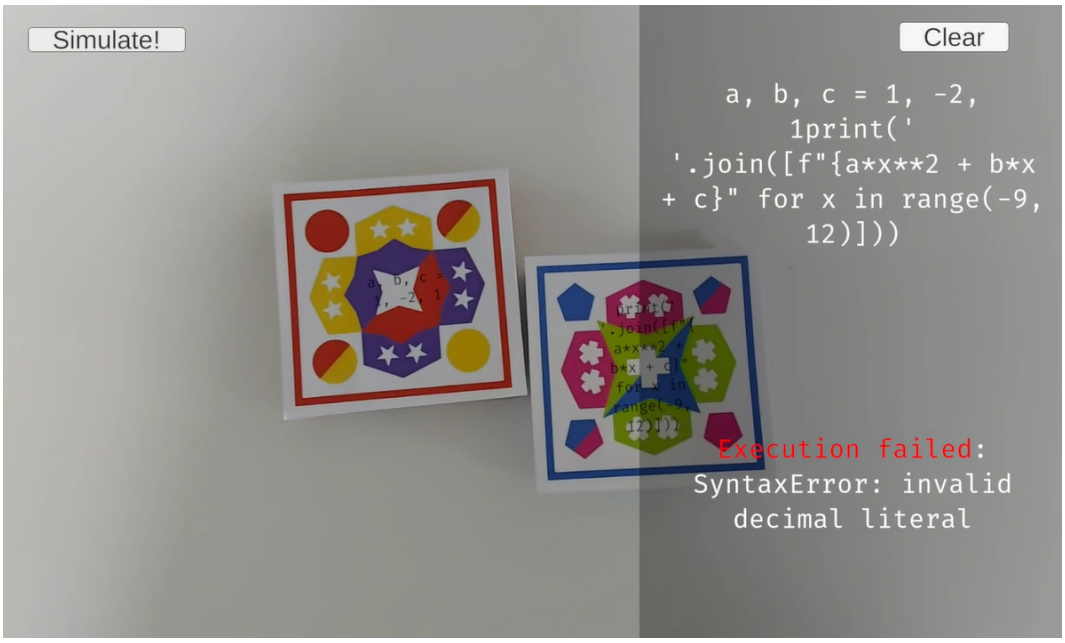

Figura 4.6: Interface após a execução falha. O lado direito exibe o código Python gerado (superior) e uma mensagem de erro detalhada no *canvas* inferior, auxiliando o usuário na identificação do problema.

Na execução bem-sucedida, é possível validar que o código gerado não apresenta nenhum erro de sintaxe, representando exatamente a disposição efetuada pelo usuário ao colocar a primeira linha de código como o bloco vermelho, e a segunda linha como o bloco azul. Já na execução falha, pode-se perceber que o usuário posicionou os blocos de modo a indicar que eles estão em uma mesma linha, sendo o vermelho o primeiro e o azul o segundo nessa ordenação horizontal. É possível validar que o código gerado no canto superior direito realmente interpretou os dois trechos de código como pertencentes a uma mesma linha, o que resultou em um erro de sintaxe do Python e, consequentemente, na renderização da mensagem de erro ao usuário.

Capítulo 5

Conclusão

5.1 Considerações finais

O projeto PyBlox cumpriu com seu objetivo central ao propor e executar a criação de uma aplicação que integra a Realidade Aumentada com o ensino introdutório de lógica de programação em Python. Em resposta à crescente demanda por metodologias de educação computacional acessíveis, o PyBlox oferece uma experiência tangível e visualmente diferenciada, transformando a manipulação de blocos físicos em código executável. A solução desenvolvida mostra como é possível utilizar a imersividade da Realidade Aumentada como um catalisador para tornar conceitos abstratos de programação mais tangíveis.

A fase de desenvolvimento exigiu soluções arquiteturais robustas para superar as limitações das plataformas móveis. A decisão de empregar o **Google Cloud Run** como núcleo computacional remoto foi de grande importância, permitindo tanto a execução confiável do código Python quanto a aplicação de algoritmos complexos de Visão Computacional para medir as dimensões dos blocos. Este mecanismo de detecção de bordas garantiu a precisão do limiar de tolerância vertical, essencial para a correta interpretação da ordenação e das quebras de linha no código gerado.

Em sua versão inicial, o PyBlox renderizava o resultado do código diretamente sobre a visão de Realidade Aumentada. Contudo, testes de usabilidade conduzidos com usuários sem nenhum conhecimento prévio de programação revelaram que essa sobreposição gerava uma carga cognitiva elevada, impedindo que o usuário compreendesse a relação causal entre os blocos e a saída do programa. Para sanar esse problema pedagógico, a interface evoluiu para um modelo tripartido. Ao separar visualmente o arranjo físico (câmera) do código gerado (*canvas* superior direito) e do resultado da simulação (*canvas* inferior direito), a aplicação estabeleceu uma ponte lógica explícita entre a ação do usuário e o efeito computacional. Além disso, os experimentos demonstrados nas figuras 4.4 e 4.6 confirmam que o PyBlox é capaz de executar código Python sintaticamente correto a partir da disposição dos blocos e, em caso de falha, fornecer *feedback* que auxilia o usuário na identificação e correção de erros.

Para trabalhos futuros, o PyBlox possui um caminho claro para expansão e aprimoramento. A implementação do Tutorial, planejada inicialmente no *Game Design Document*,

pode ser uma *feature* interessante para maximizar a retenção e o aprendizado de novos usuários. Além disso, a capacidade de alterar interativamente valores das variáveis do código é outra *feature* que poderá aumentar significativamente o potencial didático da ferramenta, permitindo que os usuários explorem o impacto de diferentes dados na execução do programa sem precisar reorganizar os blocos físicos. Por fim, seria interessante a expansão do conjunto de blocos para incluir estruturas de dados e funções mais avançadas do Python, expandindo as possibilidades da ferramenta.

O PyBlox está disponível livre e abertamente no GitHub (acesse https://github.com/TCC-MAC0499/PyBlox), podendo ser transformado colaborativamente em um sistema cada vez mais enriquecedor para o ensino de programação.

Capítulo 6

Bibliografia

[1] Luciana Alvarez. Ensino de programação é aposta de colégios em todo o mundo. *Revista Educação*, Novembro 2014.

[2] ARCore Documentation. Motion tracking.

[3] ARCore Documentation. The `arcoreimg` tool. Acessado em Agosto 2025.

[4] Google Documentation. Cloud run functions documentation.

[5] Unity Documentation. Python for unity.

[6] Unity Documentation. `Camera.WorldToScreenPoint`.

[7] Unity documentation. Unitywebrequest.

[8] Paul Jansen. Tiobe index for november 2025, Novembro 2025.

[9] Yasoob Khalid. A guide to finding books in images using python and opencv, Março 2015.

[10] Annette lamb e Larry Johnson. Scratch: Computer programming for 21st century learners. *Teacher Librarian*, 28, Abril 2011.

[11] Coursera Staff. Augmented reality vs. virtual reality: What's the difference? *Coursera*, Junho 2025.

[12] OpenCV Java Tutorials. Contour features. Acessado em Outubro 2025.

[13] OpenCV Java Tutorials. Morphological transformations. Acessado em Outubro 2025.

[14] OpenCV Python Tutorials. Canny edge detection. Acessado em Outubro 2025.

[15] OpenCV Python Tutorials. Contours: Getting started. Acessado em Outubro 2025.

[16] OpenCV Python Tutorials. Mode of contour retrieval algorithm. Acessado em Outubro 2025.

[17] OpenCV Python Tutorials. Morphological transformations. Acessado em Outubro 2025.

[18] OpenCV Python Tutorials. Smoothing images. Acessado em Outubro 2025.

[19] Google AR VR. Introduction to augmented reality and arcore.

[20] Google AR VR. Unitask.