UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

ANDRÉ SCHNEIDER GUIMARÃES MONTRESOR PIMENTA
Orientador: Prof. Dr Saul Emanuel Delabrida Silva

UMA ARQUITETURA DE MODULARIZAÇÃO DE COMPONENTES PARA DESENVOLVIMENTO DE JOGOS SÉRIOS NA UNITY ENGINE

Ouro Preto, MG
2022

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

ANDRÉ SCHNEIDER GUIMARÃES MONTRESOR PIMENTA

UMA ARQUITETURA DE MODULARIZAÇÃO DE COMPONENTES PARA DESENVOLVIMENTO DE JOGOS SÉRIOS NA UNITY ENGINE

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr Saul Emanuel Delabrida Silva

Ouro Preto, MG
2022

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

P644a Pimenta, Andre Schneider Guimaraes Montresor.
Uma arquitetura de modularização de componentes para desenvolvimento de jogos sérios na Unity Engine. [manuscrito] / Andre Schneider Guimaraes Montresor Pimenta. - 2022.
39 f.: il.: color..

Orientador: Prof. Dr. Saul Emanuel Delabrida Silva.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Graduação em Ciência da Computação .

1. xAPI. 2. Jogos Sérios. 3. Unity Engine. 4. Ensino. 5. Pacote. 6. Test-Driven Development. I. Silva, Saul Emanuel Delabrida. II. Universidade Federal de Ouro Preto. III. Título.

CDU 004

Bibliotecário(a) Responsável: Luciana De Oliveira - SIAPE: 1.937.800

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

André Schneider Guimarães Montresor Pimenta

UMA ARQUITETURA DE MODULARIZAÇÃO DE COMPONENTES PARA DESENVOLVIMENTO DE JOGOS SÉRIOS NA UNITY ENGINE

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 24 de Outubro de 2022.

Membros da banca

Saul Emanuel Delabrida Silva (Orientador) - Doutor - Universidade Federal de Ouro Preto
Rodrigo Geraldo Ribeiro (Examinador) - Doutor - Universidade Federal de Ouro Preto
Rafael Alves Bonfim de Queiroz (Examinador) - Doutor - Universidade Federal de Ouro Preto

Saul Emanuel Delabrida Silva, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 24/10/2022.

Documento assinado eletronicamente por **Saul Emanuel Delabrida Silva**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 27/10/2022, às 14:40, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0416019** e o código CRC **D79F5138**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: 3135591692 - www.ufop.br

Este trabalho foca em manter continuidade ao projeto iniciado por Nunes (2022) e Melo (2022)

Resumo

Um dos principais objetivos de um jogo sério é fornecer um ambiente adequado para ensino, porém, não necessariamente este precisa estar relacionado a apenas um tema. Este trabalho surgiu através da proposta de criar uma plataforma modular e genérica para a criação de jogos sérios, e para isso, são necessários componentes que sejam programados sem particularidades específicas que flexibilizam o desenvolvimento de jogos digitais. A proposta é desenvolver um pacote para disponibilizar funcionalidades e dependências necessárias para a implementação de um jogo didático, registrando cada interação do aluno com a sala de aula virtual desenvolvida no mesmo. Este pacote deve ter flexibilidade para receber conteúdo adicionados por educadores. Analogamente, o trabalho em questão não necessariamente está vinculado apenas a um jogo alvo, podendo ser usado por diferentes projetos de jogos sérios dentro da *Unity Engine* com intenções diversas.

Palavras-chave: xAPI. Jogos Sérios. Unity. Ensino. C#. Programação Orientada a Objetos. Pacote. Test-Driven Development.

Lista de Ilustrações

Lista de Abreviaturas e Siglas

LMS	Learning Management System
LRS	Learning Records Store
IDE	Integrated Development Environment
API	Application Program Interface
xAPI	Experience API
POO	Programação Orientada a Objetos

Sumário

1 Introdução

Jogo sério é um termo existente para identificar jogos com objetivos diversos que não centralizados à diversão ou a distração do jogador. Portanto, um jogo sério é um software interativo, com elementos que caracterizam um jogo, mas com um propósito voltado para assuntos mais sérios, como educação, marketing, exploração, entre outros. (BUSCH, 2014).

A portabilidade de um jogo, tal qual a capacidade do mesmo ser o mais abrangente possível em termos de acessibilidade, é algo a ser levado em consideração na confecção de um produto na área da computação. Afinal, quanto maior a portabilidade do produto, maior o público alcançado e portanto maior a influência que o seu objetivo pode alcançar. Em jogos sérios, a importância de uma alta acessibilidade pode implicar em uma ferramenta didática mais interessante em termos de aprendizado, visto que mais alunos e professores poderiam usar tal ferramenta em plataformas diversas.

Outro aspecto a ser levado em consideração quando se trata de desenvolvimento de aplicações na *Unity Engine* é a metodologia de desenvolvimento de pacotes. Um pacote (*package*) nada mais é que um conjunto de *scripts* e conteúdos que formam um módulo separado em um projeto. Levando isso em consideração, é possível criar um conjunto de módulos sem nenhuma dependência para com o jogo principal, proporcionando um potencial interessante para futuras implementações de sistemas customizáveis em diferentes tipos de jogos sérios. Um ambiente customizável por sua vez pode proporcionar ao usuário um controle maior sobre o que é desejado dentro da aula sem a necessidade de alterar o código fonte do produto.

Jogos em realidade virtual possuem algumas características especiais, tais quais a imersão em um ambiente altamente interativo, repetível e customizável. Isso pode ser benéfico para a implementação de um jogo sério que tem como um dos objetivos principais ser didático, fazendo com que o aluno aprenda de forma prática e empírica os fundamentos a serem ensinados pelo jogo em questão. Esse sentimento de imersão e interação é fundamental para uma experiência de aprendizado aprimorada (MIKROPOULOS; NATSIS, 2011), e foi levada em consideração no desenvolvimento e confecção dos componentes desenvolvidos nesse projeto.

Portanto, dito tudo isso, o desenvolvimento de um jogo sério com o objetivo de simular uma sala de aula específica foi iniciado, utilizando ferramentas para a criação de jogos, como o *Unity* e o *Steam VR* (NUNES, 2022). O jogo por sua vez utiliza conceitos de xAPI, uma forma modular e didática de armazenar dados pontuais, para que seja possível a integração com um serviço de LMS (*Learning Management System*) e LRS (*Learning Record Store*) os quais servem como uma forma prática de armazenar aulas e rastrear cada interação do jogador, no caso o aluno, para com o jogo em si, respectivamente (MELO, 2022). Contudo, o jogo em questão possui um baixo grau de customização ou modularidade em seus componentes, isto é, muitas

funcionalidades estão enraizadas dentro do código fonte do jogo, impossibilitando o reuso dos seus componentes em outros projetos com diferentes contextos.

Para a implementação de um sistema customizável, é necessário que cada elemento de ensino seja programado com o intuito de funcionar individualmente. Neste artigo será explicado alguns pontos fundamentais sobre a importância que tais componentes modulares proporcionam para o projeto como um todo, a necessidade dos mesmos serem separados do jogo principal, além de mostrar o propósito e a metodologia por trás do código dos novos componentes. Além disso, será explicado como a arquitetura adotada pelos componentes programados foi pensada e adaptada para acomodar as particularidades de um pacote da *Unity*.

1.1 Problema

Na construção de um jogo com propósitos de aprendizado, é importante refletir qual o impacto social que tal jogo pode alcançar em diferentes níveis de acessibilidade, tal qual a capacidade de adaptação do produto para as diferentes demandas dentro de um certo escopo, que no caso são aulas virtuais. Um jogo estático, sem nenhum aspecto customizável, pode implicar em um público alvo altamente específico, o que entra em conflito com o propósito geral do projeto. Além disso, certas alterações provindas dos usuários em relação ao conteúdo da aula que atualmente não podem ser especificadas na LMS, como por exemplo um número customizável de áudios a serem reproduzidos ou diferentes telas de apresentação de slides, teriam de ser corrigidas através de alterações diretamente no código fonte do jogo, o que pode causar trabalho desnecessário e alta dependência para com o desenvolvedor.

1.2 Justificativa

A criação de componentes individuais com alta portabilidade pode facilitar muito a implementação de um posterior sistema de customização para o usuário dentro da LMS. Componentes parametrizados cujas características podem ser definidas com dados concretos e configurações simples, além de diferentes tipos de componentes, podem ser caracterizados como uma das dependências da implementação de uma futura customização modular.

Além disso, a disponibilização dos componentes em um pacote separado pode ajudar a popularizar o conceito de jogo sério e *Learning Analytics*, uma vez que os módulos desenvolvidos são independentes e podem ser instalados em qualquer projeto, facilitando o desenvolvimento de novos trabalhos e aplicações com o intuito de aprimorar o ensino através de jogos.

1.3 Objetivos

O objetivo deste trabalho é a implementação de componentes dinâmicos e reutilizáveis que tem como público alvo desenvolvedores de jogos em *Unity*. A plataforma deve ser utilizada para aplicações focadas em registro de experiência de usuário em jogos. Paralelamente, tais componentes foram desenvolvidos com a preocupação de possuirem uma alta portabilidade, incluindo, porém não se privando apenas ao meio da Realidade Virtual.

Como objetivos específicos, temos:

- Confeccionar uma arquitetura que diz respeito à implementação dos componentes de jogo de maneira modular e independente, ao mesmo tempo acomodando as particularidades e exigências para que tais componentes estejam em um pacote da Unity;
- Adaptar os componentes já existentes no jogo confeccionado por (NUNES, 2022) à nova arquitetura;
- Criar um repositório voltado para armazenar o pacote que contém os componentes;
- Formular e programar uma interface para conectar os componentes do jogo com a LMS feita por (MELO, 2022) e qualquer LRS.

1.4 Organização do Trabalho

Neste capítulo, foi apresentada a introdução do trabalho, bem como a justificativa e os objetivos gerais e específicos. No Capítulo 2, é feita a revisão bibliográfica contendo a fundamentação teórica e os trabalhos relacionados. Logo após, no Capítulo 3 é descrito o funcionamento da arquitetura proposta e quais foram os componentes selecionados para integrá-la. O Capítulo 4 descreve as considerações finais do projeto e quais possibilidades de trabalhos futuros podem ser consideradas.

2 Revisão de Literatura

2.1 Fundamentação Teórica

2.1.1 Realidade Virtual

Realidade virtual, uma mídia que tem como principal ponto forte a imersão e interação do jogador para com o ambiente, é a inserção do usuário em um ambiente gerado computacionalmente através de dispositivos vestíveis. Vem tomando cada vez mais espaço no campo de pesquisa na área da computação, principalmente em desenvolvimento de jogos. O conceito de realidade virtual foi a princípio colocado em prática na Universidade de Utah, por volta de 1970. Ainda rudimentar, foi possível criar um headset experimental devida a criação de hardwares de rastreamento com um preço mais acessível nos anos 70 (FUCHS, 2006). Nos anos 90, a tecnologia começou a ficar mais conhecida, mas ainda não o suficiente devido a qualidade ainda baixa dos hardwares para a demanda (XIONG et al., 2021).

2.1.2 Jogos Sérios

A primeira definição para o conceito de jogos sérios foi criada por Abt (1970) constatando o seguinte: "Analisamos jogos sérios de maneira que estes tenham um propósito educacional explícito e cuidadosamente pensado, além disso, não se destinam a serem jogados primariamente por diversão". Em outras palavras, segundo ele, jogos sérios são definidos por dois aspectos, que são intrinsecamente ligados ao intuito ao qual o jogo foi criado, sendo este, didático.

Existem diversas definições no campo científico, e assim como de costume em situações similares, existe uma definição específica que é mais aceita do que as outras: "Jogos que não possuem entretenimento, diversão ou prazer como seu principal objetivo"(MICHAEL; CHEN, 2005). Um problema especial que se enfrenta em definir uma explicação para o que caracteriza um jogo sério, é determinar se dito jogo seria jogado por outro propósito que não didático por parte da maioria de seus jogadores. Isso por sua vez é algo complicado a se mensurar, já que requer um tempo especial para testes e pesquisas que envolvam teste de jogabilidade, abrindo espaço na área científica para trabalhos que focam em criar uma definição mais coesa (LAAMARTI; EID; SADDIK, 2014).

Jogo Sério é um conceito que vem crescendo no mercado, a medida que mais pesquisas relacionadas vêm sendo realizadas e o entendimento aumenta. Na década passada foi constatado que o crescimento do valor de mercado em cima de jogos sérios chegou a marca de aproximadamente 100% ao ano (DJAOUTI et al., 2011; LAAMARTI; EID; SADDIK, 2014; BUSCH, 2014).

2.1.3 Programação Orientada a Objetos

É um estilo de programação e de linguagens que existe há um tempo relativamente grande nas proporções de tempo dentro da área de computação. Em meados de 1980, diversas pesquisas foram realizadas para estudar e entender mais como uma linguagem orientada a objetos pode trazer benefícios para a programação moderna.

Uma linguagem orientada a objetos foca em separar um código em componentes chamados de *classe*, onde cada classe pode gerar um *objeto*. Um objeto por sua vez é uma instância de dados que possui seu próprio escopo, de forma que forneça ao programador meios internos de tratar os dados que dita instância armazena (RENTSCH, 1982).

2.1.4 Acoplamento em códigos

Um código coeso e com baixo acoplamento é fundamental em qualquer software que possui um planejamento a longo prazo e capacidade de crescimento. Acoplamento é uma inter-dependência de relações entre múltiplos pacotes, módulos ou componentes dentro do código fonte de um software. Isso por sua vez é algo muito difícil de se mensurar, porém, pode ser claramente percebida pelos desenvolvedores de um software que possui altos graus de acoplamento. Vários estudos inclusive já foram feitos com o intuito de pesquisar sobre maneiras mais eficientes de se medir o grau de acoplamento de um software (BRIAND; DALY; WUST, 1999).

2.1.5 Reusabilidade

Reusabilidade é um conceito importante na área de engenharia de software, uma vez que a programação orientada a objetos é amplamente usada nos dias de hoje. Ela permite que sejam programados componentes genéricos que podem ser re-usados em diferentes contextos, poupando assim tempo de desenvolvimento para realizar a manutenção e posterior melhoria do projeto em questão. Contudo, desenvolver um componente reusável prova-se um desafio, que se bem feito pode trazer inúmeros benefícios no trabalho de um projeto. (PADHY; PANIGRAHI; BABOO, 2015).

2.1.6 Singleton

A definição de um *Singleton* pode ser explicada como uma espécie de restrição de classe. Uma classe pode ser caracterizada como um Singleton caso haja alguma restrição que faça com que haja apenas uma (ou um número limitado) de objetos existindo ao mesmo tempo. Em variações de Singleton mais famosas, a instância única geralmente é acessível através de um atributo estático ou variável global.

Existem algumas variações de Singleton, onde uma pode ser mais apropriada que a outra dependendo do contexto do projeto. A variação mais utilizada é a **Lazy Instantiation**,

que se baseia em implementar um método que checa se a instância já existe, e caso não exista, automaticamente cria uma nova e aloca na variável estática (ou global) que referencia essa instância (STENCEL; WĘGRZYNOWICZ, 2008).

O projeto que será apresentado nesse texto utiliza o padrão de design de Singleton para definir algumas classes, mais especificamente a variação **Lazy Instantiation** explicada acima.

2.1.7 Test-Driven Development

O *Test-Driven Development*, ou *TDD*, é uma metodologia de desenvolvimento de software que se centraliza em testes automatizados para o desenvolivivmento de novas funcionalidades no software. Esse processo traz diversos benefícios aos desenvolvedores de um projeto quanto à organização e entendimento da construção de um código, diminuindo a densidade de defeitos na aplicação e tornando a etapa de desenvolvimento de funcionalidades proativa ao invés de reativa.

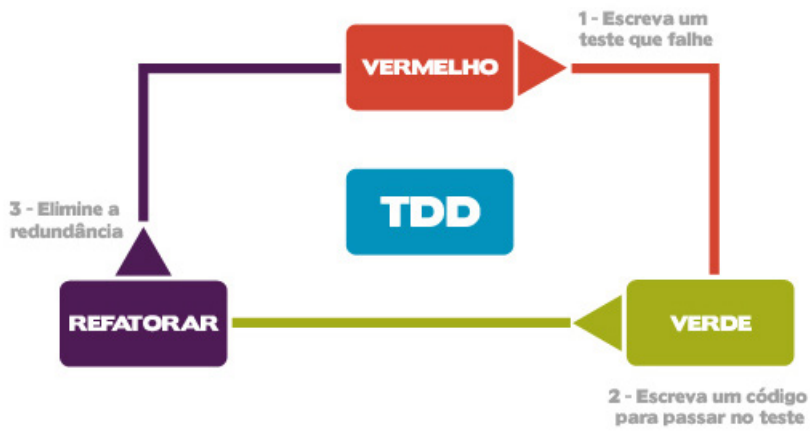

Figura 2.1 – Diagrama que ilustra o fluxo básico do *Test Driven Development*

<https://www.devmedia.com.br/tdd-fundamentos-do-desenvolvimento-orientado-a-testes/28151>

É uma metodologia relativamente simples de se entender e pode ser resumida em três etapas diferentes, onde o conjunto das mesmas representa o ciclo de desenvolvimento que idealmente é repetido durante toda a etapa de construção de uma aplicação (Figura 2.1). A primeira etapa se baseia em construir um ou mais testes, ou seja, confeccionar a ideia de uma funcionalidade juntamente com o que ela retornará. Uma vez feito isso, parte-se então para a etapa de construção de código, onde a funcionalidade será programada visando passar nos testes construídos para a mesma. Por fim, na etapa de refatoração são eliminadas todas as possíveis duplicações que o código desenvolvido na segunda etapa possa conter (BECK, 2003).

2.1.8 C#

É uma linguagem de programação relativamente nova, que mistura conceitos de C e Java de forma a focar fortemente em Orientação a Objetos. Os programas feitos em *C#* executam em .NET, um framework de código livre criado pela *Microsoft* (MICROSOFT, 2022b). É a principal linguagem utilizada no *Unity Engine*.

2.1.9 Unity Engine

Unity é uma ferramenta e um ambiente integrado de desenvolvimento (*IDE*) comumente usada para a criação de jogos virtuais. Sua primeira versão foi lançada no ano de 2005, com o objetivo de criar uma *game engine* acessível para desenvolvedores independentes, fornecendo funcionalidades que outrora só estariam disponíveis em ferramentas pagas na época (HAAS, 2014).

É uma das plataformas de criação de jogos mais populares do mundo por diversos motivos, sendo estes a gratuidade de uso para desenvolvedores independentes, o suporte para um número alto e diverso de plataformas, excelente renderização de gráficos e participação da comunidade em fornecer diversas ferramentas para outros desenvolvedores através da *Unity Asset Store*. [1]

Grande parcela dos projetos feitos em Unity é composta de jogos programados em *C#*, fazendo com que um conhecimento prévio sobre os fundamentos da orientação a objetos seja algo interessante a se possuir ao programar um jogo nessa ferramenta.

A seguir serão explicados alguns elementos que complementam o entendimento do que o Unity pode proporcionar e o seu funcionamento propriamente dito. As informações a seguir foram baseadas na documentação da ferramenta (UNITY, 2022).

2.1.9.1 Objeto de Jogo

É um conceito estritamente utilizado dentro do *Unity Engine*. Pode ser definido como um objeto que tenha qualquer tipo de comportamento dentro do jogo, seja uma fonte de luz, um personagem, uma casa, ou até mesmo algo invisível ao jogador, mas que de alguma forma altera ou complementa a execução de scripts específicos.

Cada objeto de jogo (*game object*) possui necessariamente 1 ou mais componentes, e estes por sua vez ditam como esse objeto vai se comportar dentro do programa. Cada componente geralmente é definido através de um arquivo de classe no formato "*.cs*", herdando da super classe *Monobehaviour*, que faz parte da própria API do Unity.

[1] https://www.arnia.com/what-makes-unity-so-popular-in-game-development/

2.1.9.2 Arquivo de Definição Assembly

Um assembly é um arquivo usado no ambiente de desenvolvimento do framework *.NET*, o qual a linguagem C# usa. Um arquivo de definição assembly é um conjunto de recursos que dizem respeito às funcionalidades de um código, caracterizando assim uma biblioteca de vínculo dinâmico, ou DLL *(Dynamic Link Library)* (MICROSOFT, 2022a).

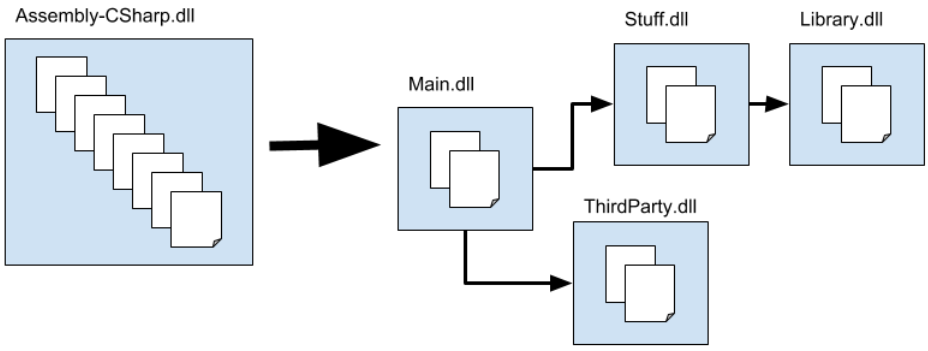

Figura 2.2 – Diagrama que ilustra como o código de um projeto na Unity pode ser separado utilizando arquivos de definição assembly (UNITY, 2022)

DLLs são utilizadas no projeto deste artigo para separar os módulos do código do pacote e fornecer um ambiente controlado de desenvolvimento dentro da Unity Engine (Figura 2.2), uma vez que arquivos de definição assembly ajudam a organizar e modularizar o código, além de tornar a compilação do mesmo significativamente mais rápida (UNITY, 2022).

2.1.9.3 Pacote

Um pacote (ou *package*) no contexto da *Unity Engine*, é um container que armazena conteúdos como scripts, bibliotecas, objetos de jogo e arquivos de gráficos. A engine fornece funcionalidades práticas para o carregamento dos pacotes, disponibilizando assim oportunidades para que desenvolvedores de diversas aplicações ou bibliotecas criem seus próprios pacotes para acomodar sua implementação dentro da Unity Engine.

As informações fundamentais de um pacote se encontram em um arquivo chamado *package.json*, que lista e armazena todas as características de dito pacote, como nome, versionamento, dependências, descrição, entre outros.

2.1.9.4 Prefabs

É um conceito estritamente utilizado dentro do *Unity Engine*. É um arquivo que armazena as informações que compõem o essencial que um objeto de jogo (*Game Object*) possui. Informações como configurações específicas de tamanho, scripts e valores pré-definidos. Seu uso é focado principalmente em fornecer objetos pré-montados ou objetos exemplo para que o

usuário que está utilizando os scripts possa ter a opção de implantação rápida e direta, sem a necessidade de montar objetos de jogo do zero.

2.1.9.5 Corrotina

Uma corrotina é uma maneira diferente de executar uma função durante a execução de um jogo. Normalmente a *engine* executa todas as funcionalidades necessárias para o funcionamento constante do jogo em um quadro (*frame*), porém, em certas ocasiões podem acabar ocorrendo travamentos caso o processamento seja muito caro ou demorado. As corrotinas por sua vez permitem que o programador especifique a função que sua execução pode ser pausada para o quadro atual, e resumida em um quadro futuro especificado. Isso faz com que execuções que dependam de conexões com a internet ou cálculos complexos por exemplo, possam ser fatiadas em múltiplos quadros, evitando travamentos indesejados.

2.1.10 xAPI

A xAPI (*Experience Application Programming Interface*) é uma tecnologia que torna possível o registro de diferentes experiências que o usuário teve com uma mídia específica. A forma que ela captura os dados que dizem respeito a isso é uma maneira fácil de entender por humanos e computadores, por possuir um vocabulário simples e coeso [2].

É construída a partir de declarações (*statements*), e essas por sua vez são subdivididas em alguns elementos que se assemelham à construção de uma frase convencional da língua inglesa (e por sua vez, portuguesa), sendo subdivididas em ator, verbo e objeto (LIM, 2015). "Hugo visualizou Slide 4"é um exemplo de declaração xAPI, onde "Hugo"é o ator, "visualizou"é o verbo e "Slide 4"é o objeto.

2.1.11 Learning Management System

Um *LMS* (*Learning Management System*) é um software que auxilia no gerenciamento de elementos de ensino, permitindo ao usuário criar, gerenciar, organizar e fornecer material didático ao seus estudantes. Um dos mais usados mundialmente é a plataforma *Moodle* [3].

É geralmente um sistema baseado em *web* que fornece uma plataforma coesa tanto para os educadores quanto para os estudantes, seja em qualquer contexto didático. Alguns LMS também podem fornecer ferramentas especiais para a análise dos dados gerados pelos estudantes durante o uso da ferramenta (BERKING; GALLAGHER, 2013).

[2] https://xapi.com/overview/
[3] https://moodle.com/news/what-is-an-lms-learning-management-systems-explained/

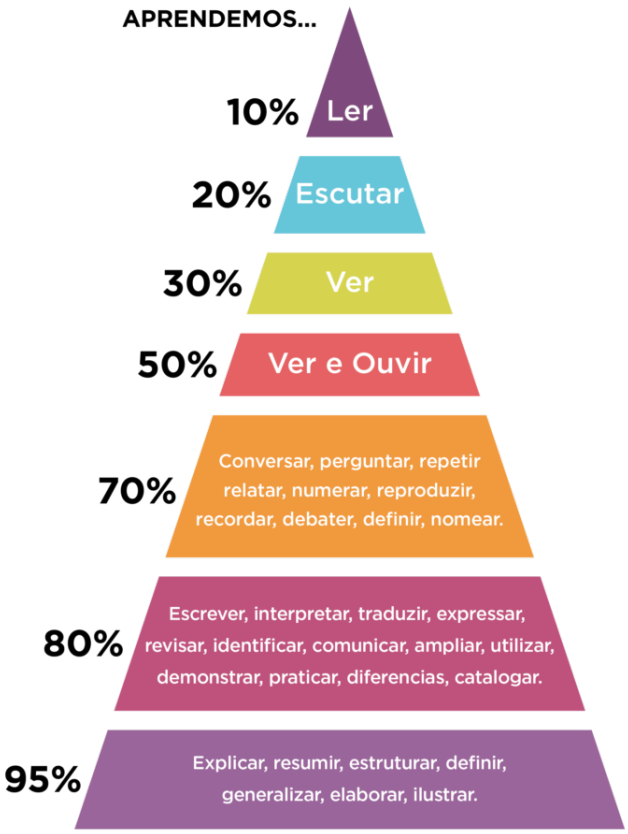

Figura 2.3 – Piramide de aprendizagem proposta por William Glasser.
<https://bm.edu.br/piramide-de-aprendizagem/>

2.1.12 Learning Records Store

Um *LRS* (*Learning Records Store*) pode ser definido como "um servidor que recebe, armazena e fornece acesso ao *Learning Records*". Pode ser considerado como uma espécie de registro que armazena os dados gerados por alguma mídia didática, para que assim possam ser analisados posteriormente (ADL, 2021).

2.1.13 Pirâmide de Aprendizagem

A ciência da aprendizagem é um campo de pesquisa que vem sendo estudado por décadas e seus avanços contruíbuiram muito para as concepções modernas das estratégias de aprendizagem que conhecemos hoje (COBERN; AIKENHEAD, 1997).

Na área de ciência da aprendizagem, diversas pesquisas na história procuraram definir de forma ampla como podem ser classificadas os vários métodos de ensino usados na sociedade moderna, e a mais conhecida e citada é a Pirâmide de Aprendizagem criada por Glasser (1999).

Na teoria de Glasser, os vários meios de aprendizado são sub-divididos em dois tipos:

Passivo, onde o aluno não participa diretamente na conversa e troca de informações com o educador, e Ativo, onde as ações do próprio aluno fazem parte do próprio processo de ensino. Segundo o psiquiatra, o aprendizado ativo tende a ser muito mais efetivo do que o passivo.

Dito isso, Glasser propôs uma ilustração em forma de pirâmide para classificar cada método de ensino relevante usado atualmente (Figura 2.3). A metade de cima da pirâmide (Os 4 últimos andares) representa os métodos de aprendizagem passivos, e a metade de baixo (3 primeiros andares) representa os métodos de aprendizagem ativos. As porcentagens dizem respeito a taxa de aproveitamento de aprendizado que cada método fornece ao ser praticado.

As ideias principais do projeto de desenvolvimento dos componentes explicadas neste trabalho tem como proposta aplicá-los a um ou mais andares da pirâmide, fazendo com que seja possível assim medir a efetividade propriamente dita em questão de aprendizado para cada um dos componentes desenvolvidos.

2.2 Trabalhos Relacionados

Os trabalhos relacionados a este projeto entram no campo de criação de jogos sérios dentro da *Unity Engine* e/ou trabalhos que focam em desenvolver uma arquitetura com baixo acoplamento de código para algum software específico, além dos trabalhos que complementam o projeto de desenvolvimento para jogos sérios ao qual este trabalho faz parte.

Nunes (2022) desenvolveu um jogo sério em Realidade Virtual na Unity Engine que gera uma sala de aula virtual com conteúdos diversificados, com o propósito de popularizar e incentivar a educação através de mídias interativas (Figura 2.4). O ganho que um jogo sério bem formulado pode representar para tarefas de aprendizado, principalmente na Realidade Virtual é algo a se levar em consideração. O jogo em questão é composto por uma interface de menu principal e uma cena de laboratório. Durante o carregamento, o jogo se comunica com uma plataforma web que por sua vez fornece uma interface para o usuário customizar o conteúdo das aulas do jogo de maneira dinâmica.

Melo (2022) confeccionou uma arquitetura aplicada em uma plataforma de gerenciamento de dados de ensino (*Learning Management System*) (Figura 2.5), além de um banco de dados que armazena a experiência do usuário através de declarações xAPI, que é caracterizado como um LRS (Learning Records Store). A ideia do desenvolvimento de ambas plataformas é principalmente de fornecer um ambiente amigável e eficiente para o usuário, provendo maneiras de customizar o conteúdo de uma aula e, posteriormente, assim conseguir os dados referentes à experiência dos alunos em relação a essa mesma aula, utilizando a tecnologia xAPI. Contudo, este trabalho não é independente e faz parte de um projeto com um escopo maior, complementando as funcionalidades as quais o trabalho de Nunes (2022) acessa para carregar os dados da respectiva sala de aula virtual no jogo desenvolvido.

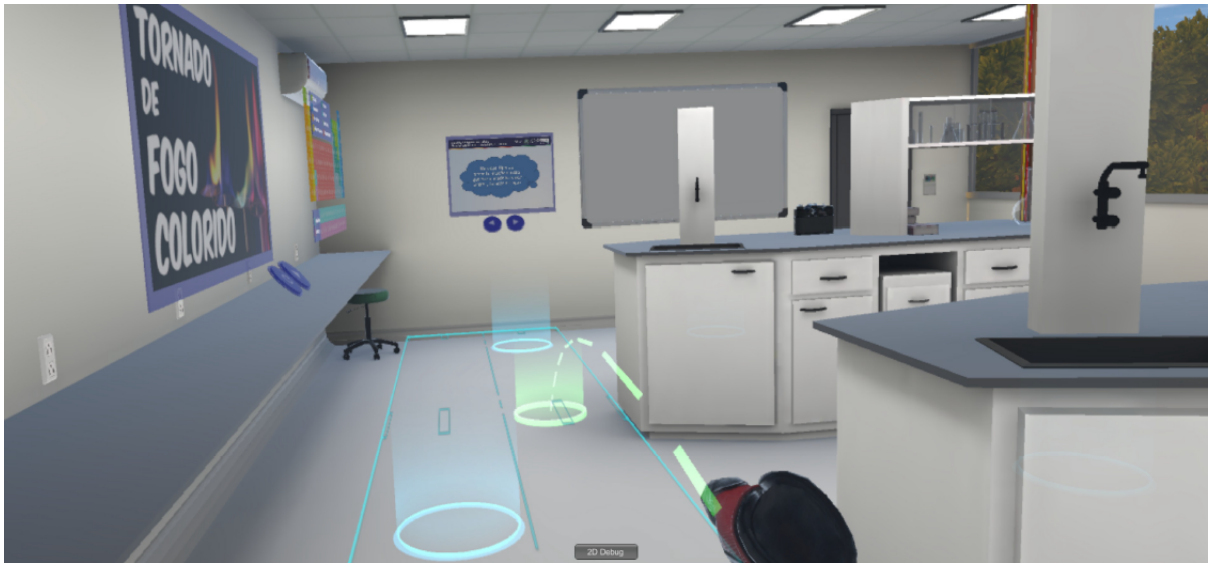

Figura 2.4 – Visualização do laboratório virtual em RV construído através da Unity por Nunes (2022)

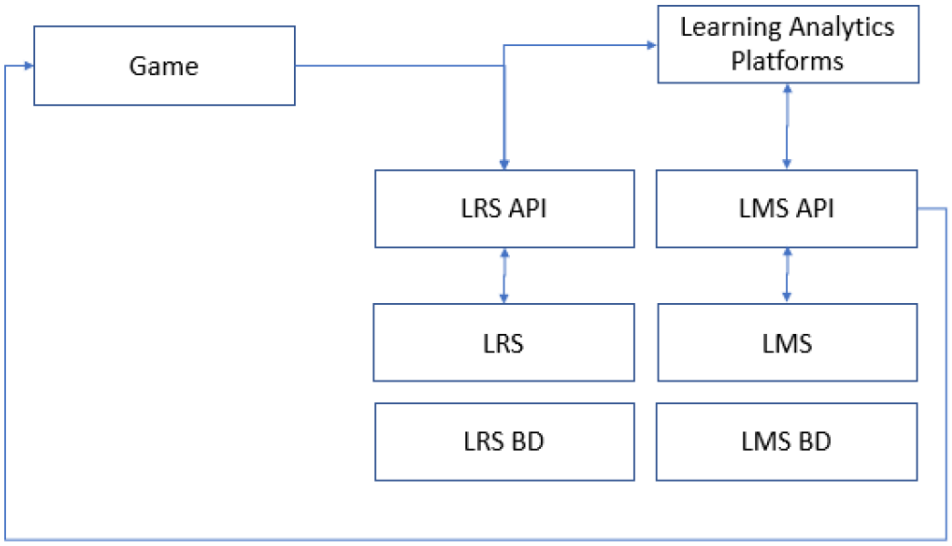

Figura 2.5 – Arquitetura proposta por Melo (2022) para o fluxo dos dados e experiência de usuário

Vidakis et al. (2019) estudaram e desenvolveram uma biblioteca especial que implementam a xAPI, voltado a analisar dados educacionais criados através de um jogo feito dentro da *Unity Engine*. É explicada com detalhes como a arquitetura dessa biblioteca foi pensada, com a portabilidade que o *Unity Engine* oferece em mente. É uma biblioteca que foca em gerar dados consistentes para uma posterior e mais facilitada análise dos mesmos, chamada de *learning analytics*.

Houliston et al. (2016) desenvolveram uma nova arquitetura de código para a comunicação entre módulos de um robô humanóide. O interessante desse trabalho é o estudo aprofundado feito para focar em melhorias relacionadas ao código que já existia, porém mostrava claros gargalos

em seu design em termos de modularidade, atrasando significativamente o desenvolvimento do projeto. A arquitetura que foi proposta pelos autores foca em diminuir o acoplamento entre os componentes das estruturas do código fonte do software, e sua concepção foi construída através de um estudo de diferentes arquiteturas de software como *Whiteboard*, *Messaging* e *Blackboard*, apontando os prós e contras de cada uma, e como seria possível criar uma nova que se aproveitasse dos seus respectivos pontos positivos.

3 Desenvolvimento

A modularização e separação dos componentes do jogo para um pacote são o foco principal desse trabalho. É importante ressaltar que o código fonte do jogo já existe e está em constante evolução em diversos aspectos (MELO, 2022; NUNES, 2022), contudo, o jogo no momento é programado exclusivamente para uma cena específica sem uma capacidade elevada de adaptação para outros tipos de aula, os quais requerem um grau de customização mais notável, como várias telas de slide, diferentes áudios e questionários na mesma lição.

Foi então proposta uma arquitetura de organização dos componentes dentro da *Unity Engine* na versão *2020.3.30f1 (LTS)* visando a independência de cada elemento didático que um jogo sério possa conter. Essa arquitetura por sua vez é baseada em metodologias de desenvolvimento de pacotes da *Unity* e *Test-Driven Development*, focando em garantir um código livre do contexto de um projeto específico, o uso seletivo de componentes que o usuário possa julgar necessário para sua sala de aula virtual, além de também fornecer uma pequena interface para envio de declarações xAPI a uma LRS cadastrada.

Após confeccionada, a arquitetura é então aplicada nos componentes em questão, desacoplando-os do código fonte do jogo principal e alocando-os em um repositório separado. Os componentes são:

- Reprodução de Áudio
- Tela de Slides
- Questionário
- Interface LMS
- Interface LRS

A seguir, será explicado em tópicos cada etapa do trabalho.

3.1 Arquitetura

A arquitetura que envolve os componentes programados nesse projeto foca principalmente em isolar o código dos mesmos e introduzi-los em contextos próprios e independentes. Tal estratégia, se aplicada corretamente, pode facilitar a manutenção de cada componente específico do projeto, seja em correção de bugs ou melhorias, já que a alteração é feita apenas no escopo do componente em si, sem levar em consideração o ambiente em que ele é usado (BRIAND; DALY; WUST, 1999). Além disso, para os propósitos de facilidade da montagem das salas de

aula, componentes individuais independentes podem ajudar muito na programação de diferentes salas com diferentes temas, uma vez que seu conteúdo é genérico e pode ser customizado por quem está utilizando-os de maneira simples e direta através das interfaces do *Unity Editor*.

A programação dos mesmos foi com o tempo adaptada para acomodar testes unitários, onde cada componente possui um arquivo que define um ambiente de teste para suas funcionalidades mais importantes. A implementação de testes unitários é uma etapa muito importante no desenvolvimento de um software ou API, uma vez que provê garantias ao programador de que o código funciona da maneira esperada em um ambiente controlado (HOCK; SCHITTKOWSKI, 1980). A etapa de testes pode dar ao programador também uma boa noção das características do código fonte do projeto, como o acoplamento das funcionalidades ou modularidade das responsabilidades de cada função, que está potencialmente ligado à dificuldade de programar testes para as diferentes funções do software (RAMLER; MOSER; PICHLER, 2016). Dito isso, é interessante que futuros componentes sejam implementados na metodologia de implementação *Test-Driven Development*, diminuindo drasticamente a chance de um código com más práticas de desenvolvimento de software seja confeccionado (BECK, 2003).

O pacote possui um *namespace* geral chamado *SeriousGameComponents*, e cada componente que segue a arquitetura proposta possui um namespace específico dentro do escopo do *SeriousGameComponents*. Todo e qualquer código novo criado para definir o comportamento do componente em questão deve estar dentro de um namespace em comum. Isso faz com que seja possível o uso de arquivos de script com nomes padronizados entre os componentes, já que a *Unity Engine* por natureza não permite classes com o mesmo nome em um mesmo projeto. De qualquer forma, um namespace é apenas uma abstração usada pela linguagem *C#* para tornar até certo grau, transparente certos prefixos de nomes de classes para o programador.

Além dos namespaces, conceitos fundamentais de Programação Orientada a Objetos (POO) foram aplicados na base da arquitetura, já que o código é construído em *C#*, uma linguagem orientada a objetos. Um projeto pode se beneficiar bastante com uma arquitetura baseada em heranças de classes, trazendo um código compacto e reduzido, além de aumentar a reusabilidade das classes mais altas na herança. Isso pode acabar poupando tempo valioso de desenvolvimento e é um dos principais fortes que a POO pode trazer (PADHY; PANIGRAHI; BABOO, 2015).

Dito isso, foram confeccionadas classes abstratas que fornecem as funcionalidades básicas que todos os componentes seguindo a arquitetura proposta deverão possuir (Figura 3.2). O benefício que isso traz é enorme, pois futuras melhorias ou correções que não estão ligadas a um componente apenas, podem ser implementadas nas classes abstratas em questão.

O código do pacote é sub-dividido em diferentes diretórios, de forma que os requisitos para que o repositório em questão seja reconhecido como um pacote da *Unity Engine* sejam devidamente cumpridos (Figura 3.1). Os *scripts* de funcionamento dos componentes se encontram dentro de um diretório chamado *"Runtime"*, que por sua vez contém um arquivo de formato *.asmdef (arquivo de definição assembly)* para manter o código dentro do contexto fechado do

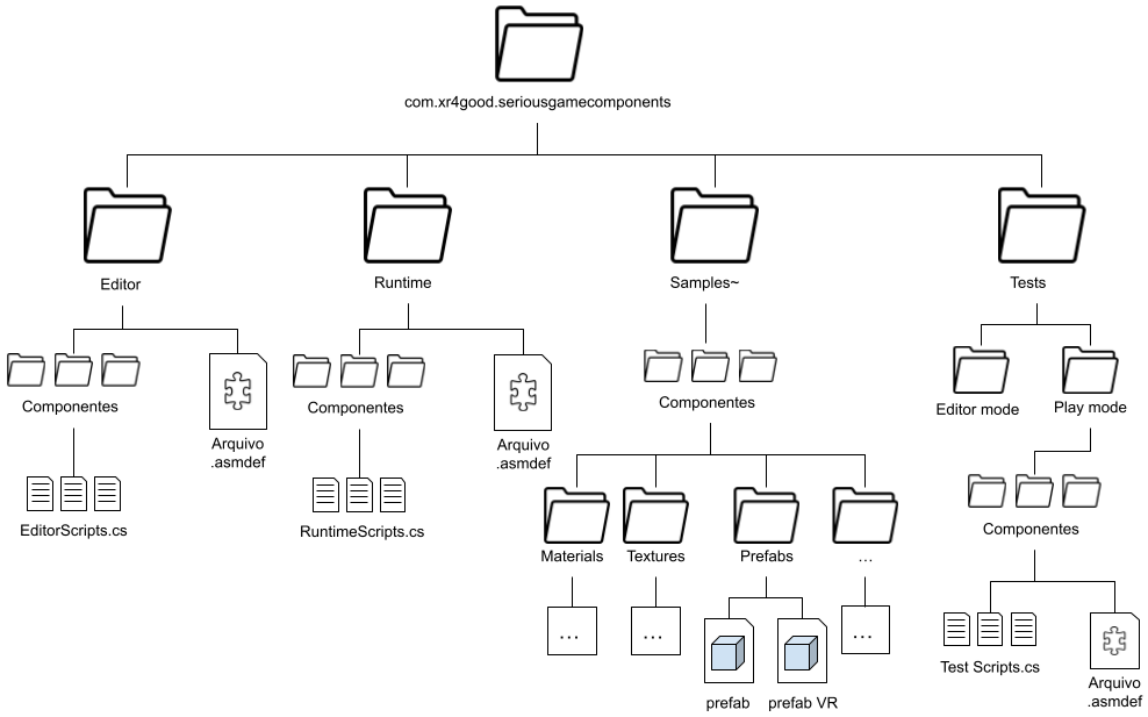

Figura 3.1 – Estrutura de diretórios do repositório.

pacote e diferentes diretórios que dizem respeito aos diversos componentes presentes no projeto, os quais contém os seus respectivos *scripts* em C#.

Por fim, para auxílio na construção das declarações de xAPI que serão geradas, pequenas classes estáticas chamadas *LogVerb, LogActivity e LogExtension* foram criadas para guardar algumas informações específicas de cada elemento da declaração xAPI que o componente envia a uma LRS cadastrada.

3.1.1 Base Statement Sender

Uma classe abstrata que fornece as principais funcionalidades de comunicação com a LRS. As classes que a herdam vão ter a disposição métodos específicos para o envio das declarações xAPI de maneira simples, sem que haja a necessidade de programar eles do zero para cada componente individual. Todo componente obrigatoriamente deve possuir ao menos um script que herde, para que seja possível o envio das declarações xAPI.

A classe utiliza conceitos de programação paralela para executar as funcionalidades fornecidas pela *TinCan API*, uma *API* com funções exclusivas para a montagem das declarações *xAPI* e comunicação com qualquer LRS baseada nas mesmas. A funcionalidade de envio dos dados para a *LRS* depende diretamente de uma resposta do servidor, isso apresenta um problema importante no contexto de aplicações de jogos, uma vez que as mesmas são baseadas em chamadas repetidas de função a cada quadro (*frame*) executado no jogo. Caso a resposta do servidor necessite de um tempo mais longo para ser recebida, a execução do jogo pode travar e apresentar

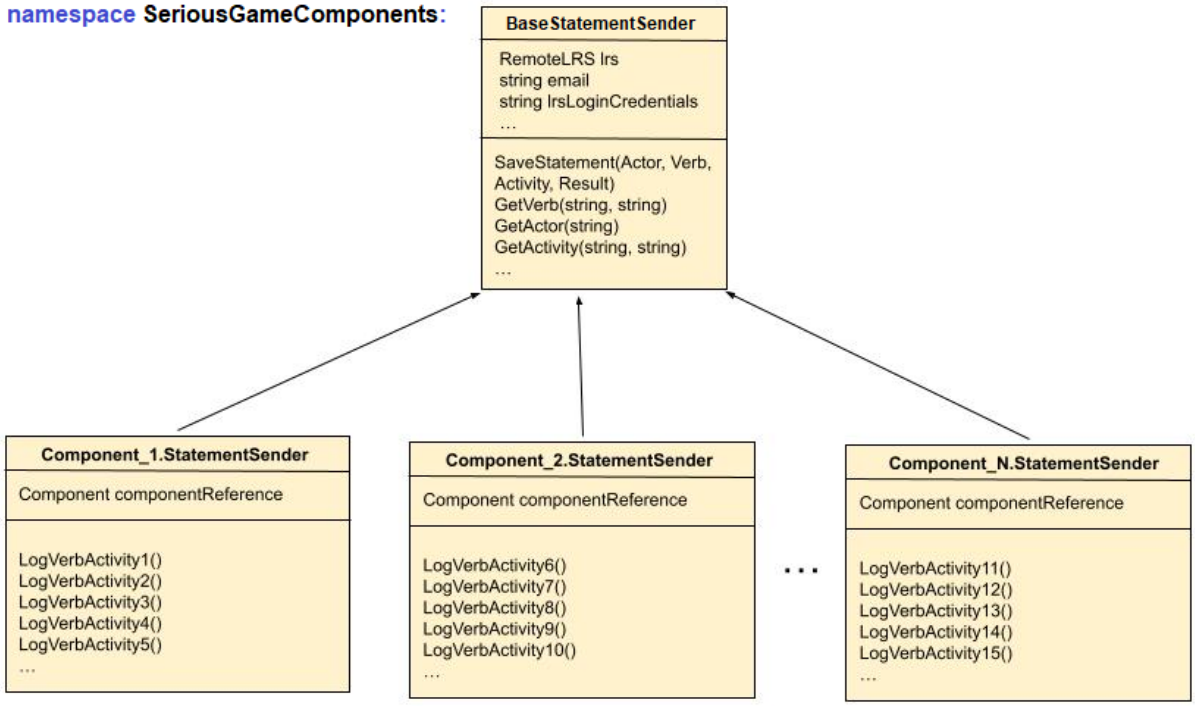

Figura 3.2 – Esquema UML da relação que cada componente tem com a classe *BaseStatement-Sender*.

complicações na jogabilidade, por isso, toda e qualquer funcionalidade das classes de Statement Sender executam as funções de comunicação e recebimento de resposta com a *LRS* em uma thread diferente da principal, com o auxílio de funções assíncronas para mandar a requisição e corrotinas para esperar a resposta.

A função que gerencia a montagem de informações e realiza o envio da declaração xAPI é por sua vez uma corrotina, ou seja, é possível postergar sua execução para um quadro (frame) futuro, dito isso, foi então criada uma tarefa paralela para executar a comunicação com a LRS e enquanto sua execução acontece, a corrotina entra em modo de espera através de um comando de laço (Figura 3.3), onde no caso da *thread* ainda não ter terminado sua execução, é realizada uma nova checagem no próximo quadro do jogo, evitando assim engasgos e queda de quadros por segundo durante a execução da aplicação.

3.1.2 Button Helper Abstract

É uma pequena classe abstrata que facilita a programação do comportamento de elementos interagíveis no componente. Sua utilização não é obrigatória para a integridade da arquitetura. Fornece alguns métodos para desativar ou ativar botões interativos relacionados ao componente.

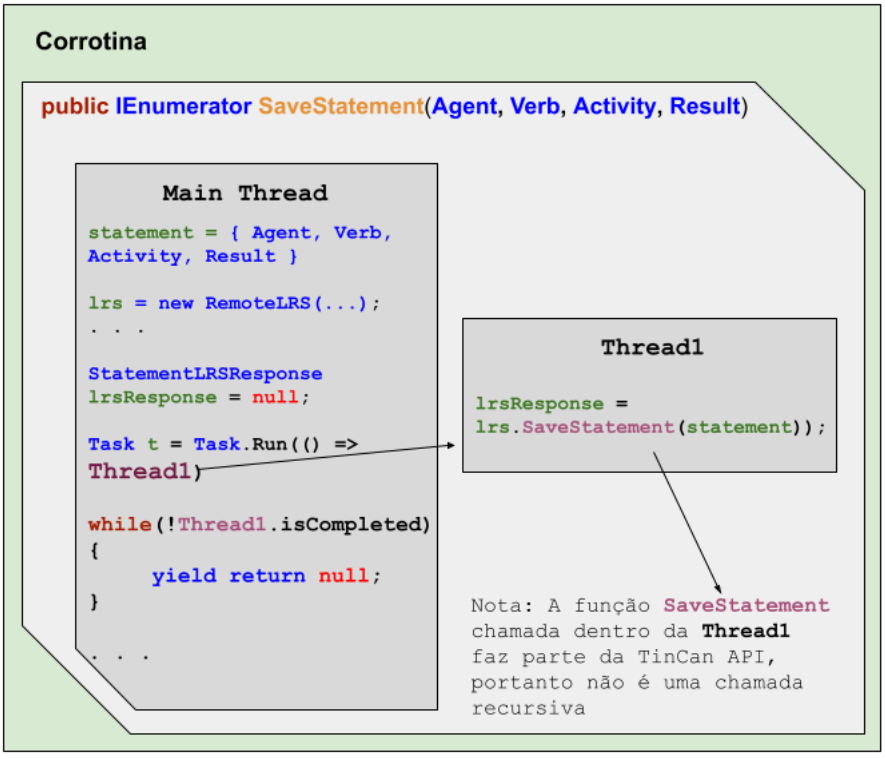

Figura 3.3 – Ilustração demonstrando a execução do código para mandar a declaração xAPI à LRS e esperar a resposta do servidor sem travar a execução do jogo

3.2 Componentes

Já construída a arquitetura conceitual, foi então realizada a implementação propriamente dita dos componentes. Os componentes foram pensados com a ideia da pirâmide de aprendizado em mente (GLASSER, 1999). Cada componente pode ser definido com um ou mais métodos de aprendizado específicos, e este pode ser passivo ou ativo. Vale ressaltar que os componentes se usados em conjunto, podem melhorar significativamente a taxa de aprendizado do aluno segundo a ilustração da pirâmide (Figura 2.3).

Os componentes possuem um arquivo que especifica um objeto já montado dentro do *Unity Engine* (*prefab*), que se encontra dentro do diretório *Samples/<nome do componente>*, facilitando o uso rápido do pacote sem que o usuário precise obrigatoriamente montar do zero o objeto de jogo com os scripts, texturas, modelos *3D* e objetos interativos para conseguir usá-lo apropriadamente no jogo. Esses diferentes conteúdos podem todos ser importados separadamente a desejo do usuário no processo de instalação do pacote.

Apesar da arquitetura fornecer um conjunto de regras a serem seguidas para a programação de cada componente didático do jogo, eles ainda possuem suas particularidades que não necessariamente vão estar especificadas na definição da arquitetura. É importante ressaltar também que os componentes são completamente independentes, por isso, no caso do uso de dados a partir de uma *LMS* diferente da definida pelo componente *LMSLoader* presente no pacote,

as particularidades desta LMS em questão devem ser programadas e destrinchadas através de um código externo para que os componentes possam usar estes dados, ou seja, a interface de comunicação com a web e download de dados criada nesse projeto foi feita com apenas uma LMS específica em mente.

Além disso, todos os componentes possuem suporte para Realidade Virtual utilizando a biblioteca *SteamVR*, porém não é obrigatório, ou seja, sua programação não foi feita com exclusividade para RV em mente, podendo se adaptar a diferentes projetos com diferentes graus de acessibilidade.

A seguir será explicado em tópicos cada particularidade dos componentes implementados no pacote, de maneira a esclarecer o que pode ser definido como novo em cada componente e o que deve ser seguido para se manter dentro das definições da arquitetura em questão em caso de adições futuras.

3.2.1 Componentes de Funcionamento

Componentes que não são notáveis para o jogador e auxiliam no funcionamento dos Componentes de Reprodução para certas funcionalidades

3.2.1.1 Componente de Interface com a LMS

É um componente de funcionamento que gerencia a comunicação com a *XR4Good LMS*, ou seja, foi programado para baixar os dados de uma lição específica e disponibilizá-los para que os componentes de reprodução possam mostrá-los ao jogador de maneira interagível.

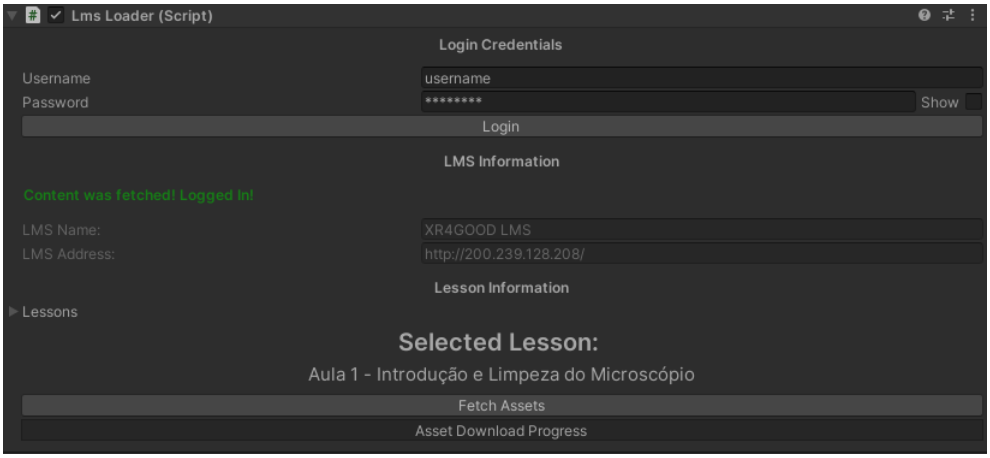

Figura 3.4 – Interface do inspetor dentro da *Unity Engine* para o script *LMSLoader.cs*

Possui um autenticador com usuário e senha para realizar o login na LMS, além de interpretadores para cada tipo de dado disponível para a lição escolhida em formato *Json*, como imagens para slides, questionários e áudios. Esses dados por sua vez são armazenados em variáveis estáticas acessíveis por todos os componentes, tornando possível o acesso dos dados

sem referenciar o objeto em questão, e assim facilitando implantação dos componentes em qualquer cena da *Unity* (Figura 3.5). Vale notar que como o *LMS Loader* se trata de uma interface que se comunica diretamente com a LMS, qualquer alteração na formatação de retorno do *Json* esperado deve ser reavaliada pela API disponibilizada no script *LmsLoader.cs*, que diz respeito ao funcionamento base desse componente.

As estruturas de dados que armazenam o conteúdo que a LMS aponta possuem uma característica que foi pensada para acomodar múltiplos componentes do mesmo tipo em uma única cena específica. Os dados baixados são armazenados em dicionários onde a chave *(Key)* representa um dado do tipo *string*, e seu significado diz respeito a uma palavra que identifica um componente na cena de jogo. É importante ressaltar que uma vez baixado o conteúdo da lição, não é mais necessário que o componente *LMS Loader* exista na cena atual, possibilitando assim trocas de cenas dentro do jogo sem se preocupar com perda de dados ou autenticar por múltiplas vezes.

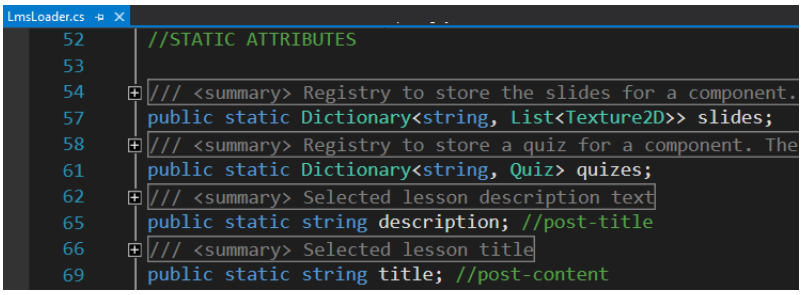

Figura 3.5 – Estruturas de dados que armazenam o conteúdo baixado

Isso permite um grau de customização interessante para o usuário do pacote, uma vez que o mesmo consegue especificar em grupos os dados que certos componentes executarão. Vale ressaltar no entando que essa funcionalidade ainda não foi implementada na *LMS*, e por enquanto é exclusiva na aplicação do cliente, portanto a princípio todos os componentes usam uma palavra padrão como identificador (Figura 3.6).

Foi utilizado o conceito de *Singleton* para programar este componente, isto é, poderá existir apenas uma classe instanciada do mesmo em uma cena no *Unity*. Isso foi feito para evitar conflitos de download e mudanças de variáveis estáticas ao mesmo tempo, que naturalmente ocorreria caso houvesse mais de um objeto com este componente instanciado em uma cena alterando as mesmas variáveis globais.

3.2.1.2 Componente de Interface com a LRS

Similar ao componente de interface com a *LMS*, este é um *Singleton* e também é um componente de funcionamento, isto é, pode apenas existir uma instância na cena e sua existência não é diretamente aparente ao jogador. Sua função é simplesmente guardar as informações das

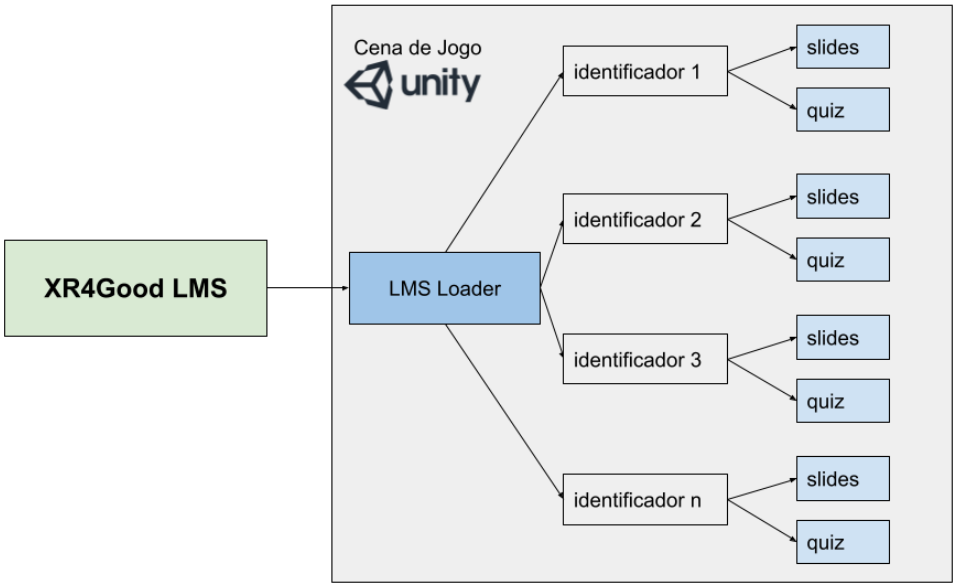

Figura 3.6 – Organização dos componentes na cena usando identificadores

credenciais da LRS desejada em atributos estáticos para que os Componentes de Funcionamento possam acessá-los de maneira global.

Recebe como entrada a *URL* da *LRS* e as chaves credenciais para login na plataforma *scorm*, para o armazenamento das declarações xAPI (Figura 3.7).

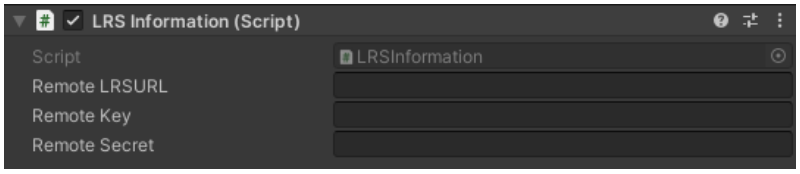

Figura 3.7 – Interface do inspetor dentro da *Unity Engine* para o script *LRSInformation.cs*

3.2.2 Componentes de Reprodução

3.2.2.1 Componente de Áudio

O componente de áudio possui um script central (*AudioPlayer.cs*) que gerencia quais áudios serão reproduzidos, podendo baixá-los a partir de uma *URL*, carregá-los através do componente *LMS Loader* ou reproduzi-los localmente sem a necessidade de internet. Ele armazena uma lista de áudios que podem ser reproduzidos, pausados ou reiniciados referenciando os respectivos métodos detalhados na documentação da classe. O componente também disponibiliza uma pequena interface para testes e downloads de áudios da web de maneira dinâmica.

O componente de reprodução de áudio foi pensado para acomodar o método passivo de aprendizado descrito na pirâmide de aprendizado de Glasser (Figura 2.3), sendo esse o "Escutar".

Figura 3.8 – *Prefab* do componente de áudio dentro do jogo.

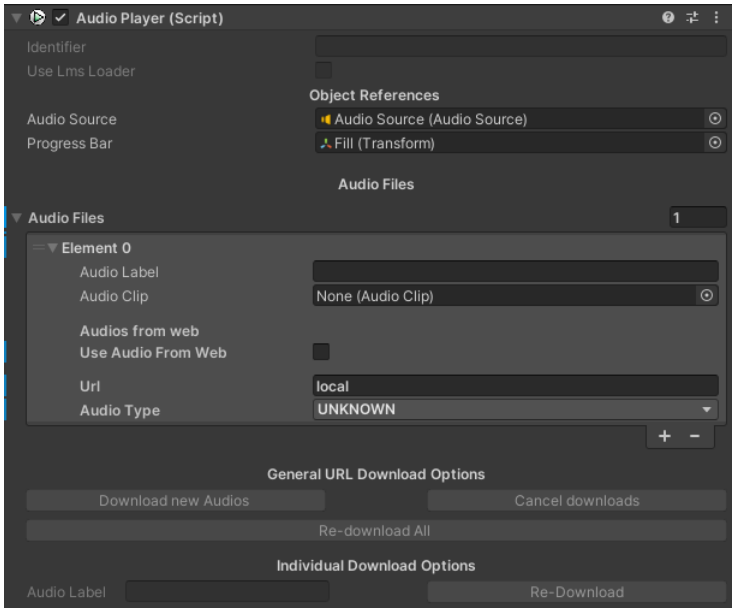

Figura 3.9 – Interface do inspetor dentro do *Unity Engine* para o script central (*AudioPlayer.cs*).

Nota-se que possui uma taxa de aprendizagem de 20% se usado de maneira individual. Essa taxa pode ou não aumentar caso seja usado em conjunto com outro componente.

Também foi implementada uma melhor interface no inspetor do Unity Engine, o que faz com que o uso desse componente não dependa que o usuário que está usando o pacote necessite fazer muitas ou sequer alguma alteração no código para satisfazer suas dependências (Figura 3.9).

É importante ressaltar que a interface de carregamento para com os dados da *LMS* não foi programada para esse componente no momento, devido ao fato de ainda não haver suporte para upload de arquivos de áudio por parte da própria interface da *LMS* (MELO, 2022), não estando assim dentro do escopo do que foi trabalhado nessa monografia, que por sua vez diz respeito a implementações de aplicação no lado do cliente, e não do servidor.

O *prefab* incluso no pacote é um pequeno objeto com 3 botões de controle de áudio e 2 botões para navegar na lista de áudios, além de uma barra de progresso para o áudio sendo reproduzido no momento (Figura 3.8).

Ao todo, o componente de áudio pode enviar 5 verbos xAPI diferentes e 1 atividade

através do script *StatementSender.cs* para as diversas opções de interação que ele fornece ao jogador.

Verbos xAPI:

- Started (Iniciou)
- Resumed (Resumiu)
- Paused (Pausou)
- Canceled (Cancelou)
- Listened (Ouviu)
- Skipped (Pulou)
- Returned (Retornou)

Atividades xAPI:

- Audio

Pode reproduzir todos os tipos de áudio suportados pela biblioteca básica do *Unity Engine (ACC, AIFF, IT, MOD, MPEG, OGGVORBIS, S3M, WAV, XM, XMA, VAG)*.

3.2.2.2 Componente de Slides

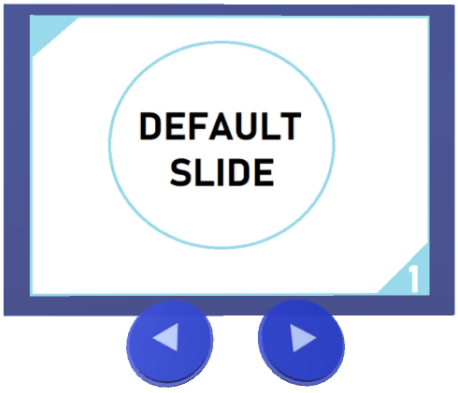

Figura 3.10 – *Prefab* do componente de slide dentro do jogo

O componente de slides possui um script controlador (*SlideController.cs*) cujo objetivo é fazer o gerenciamento dos slides a serem baixados ou carregados localmente e então mostrados ao jogador. Os slides por sua vez são arquivos de imagem projetados em um *Canvas Object* dentro do Unity Engine.

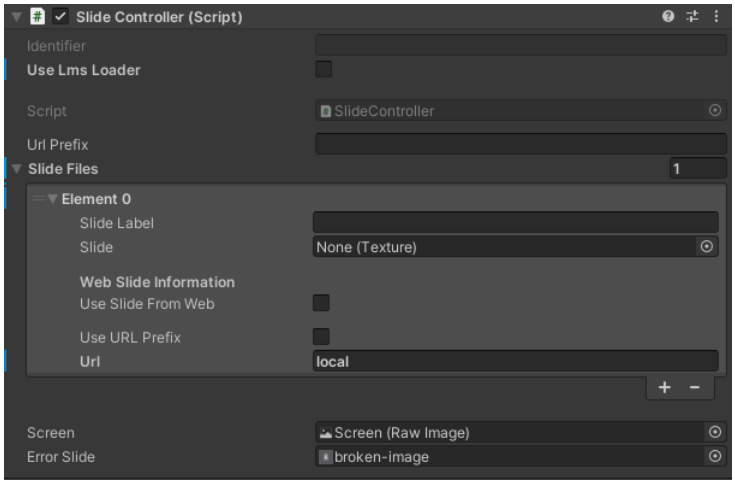

Figura 3.11 – Interface do inspetor dentro do *Unity Engine* para o script central (*SlideController.cs*)

O componente de slides foi pensado para acomodar o método passivo de aprendizado descrito na pirâmide de aprendizado de Glasser (Figura 2.3), sendo esse o "Ver". Nota-se que possui uma taxa de aprendizagem de 30% se usado de maneira individual. Essa taxa pode ou não aumentar caso seja usado em conjunto com outro componente.

O carregamento das informações baixadas através da *LMS* é ativado através de uma variável *booleana* que, se verdadeira, faz com que todos os dados e registros da classe que diz respeito ao componente em questão sejam carregados automaticamente, uma vez disponíveis, para uso. A conversão de informações é diretamente dependente do formato no qual elas foram organizadas pelo componente de carregamento da LMS (*LMS Loader*), ou seja, para que essa funcionalidade seja executada apropriadamente, o componente *LMS Loader* precisa existir na cena de jogo no qual esse componente se encontra, e ter executado uma autenticação de login bem-sucedida, além de já ter baixado o conteúdo da aula.

```
SlideController.cs
158    public void LoadDownloadedLMSData(bool reload = true)
159    {
160        ...
184        foreach(Texture2D t in LmsComponent.LmsLoader.slides[identifier])
185        {
186            slideFiles.Add(new SlideRegistry(t, "slide_" + slideCount));
187            slideCount++;
188        }
189
190        lmsDataLoaded = true;
191    }
```

Figura 3.12 – Código da funcionalidade de carregamento dos dados baixados pela LMS, que dizem respeito ao componente de slides

O *prefab* incluso no pacote são dois botões (voltar e avançar) com um canvas acima para mostrar as imagens *2D* que serão carregadas pelo componente (Figura 3.10). Ao todo, o componente de slide pode enviar 4 verbos diferentes e 2 atividades através do script *StatementSender.cs*

para as diversas opções de interação que ele fornece ao jogador.

Assim como o componente de áudio, os slides podem ser baixados através de *URLs* ou carregados localmente sem a necessidade de internet. Ele armazena uma lista de imagens que podem ser mostradas em uma tela dentro da cena do jogo, qual tela e quais objetos interativos servirão para controlar os slides cabe ao usuário definir (Figura 3.11).

Verbos xAPI:

- *Viewed* (Visualizou)
- *Completed* (Completou)
- *Skipped* (Pulou/Passou)
- *Returned* (Returnou)

Atividades xAPI:

- Slide
- SlideDeck

3.2.2.3 Componente de Questionário

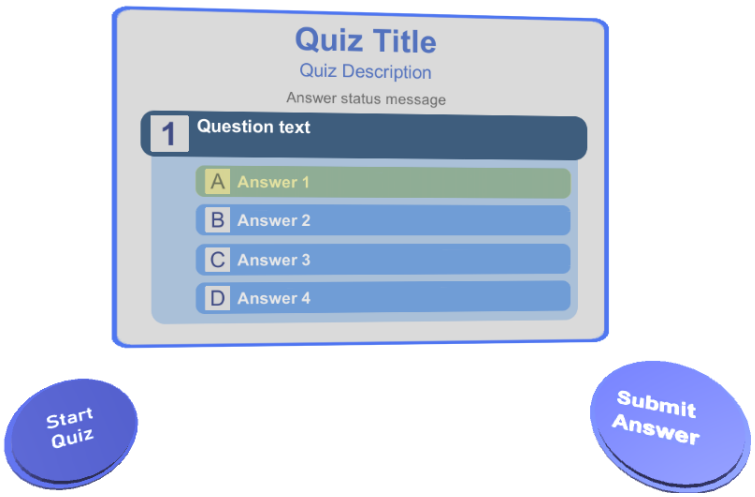

Figura 3.13 – *Prefab* do componente de questionário dentro do jogo

O componente de questionário possui um script central que controla e armazena uma ou mais perguntas criadas pelo usuário. Esse script por sua vez é uma classe que gerencia o questionário. Dentro dessa classe há algumas outras classes para definir as estruturas dos dados que ficarão guardados dentro do objeto. A subclasse *QuizQuestion* (Figura 3.15) que armazena os dados de uma pergunta, possui por sua vez, uma *struct* para armazenar dados compostos que

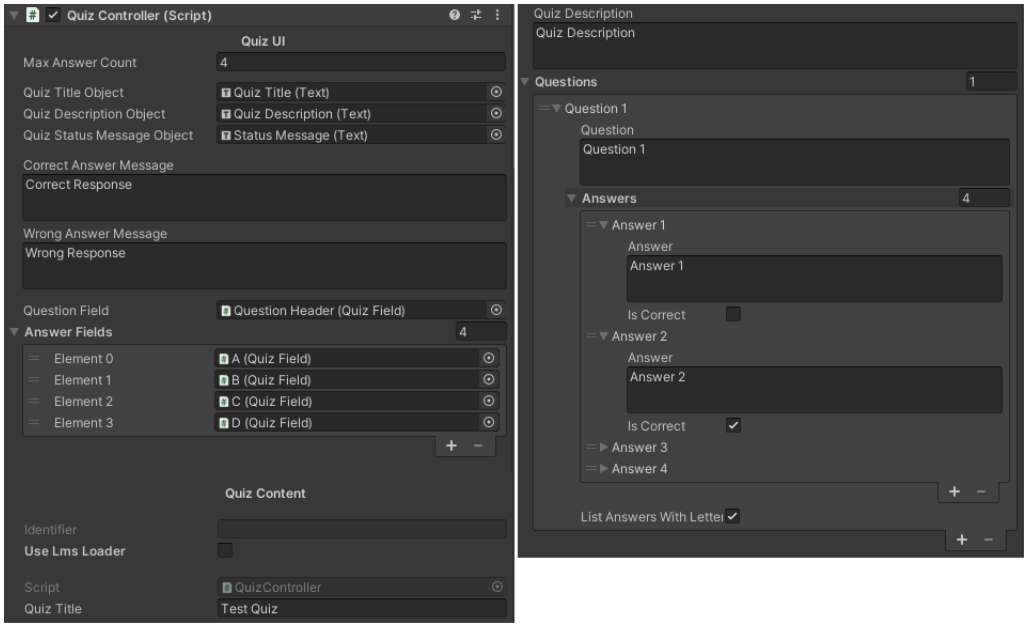

Figura 3.14 – Interface do inspetor dentro do *Unity Engine* para o script central (*QuizController.cs*)

correspondem às respostas. Cada pergunta pode ter quantas respostas o usuário quiser, desde que ele referencie todos os objetos que serão atualizados pelo script controlador, além disso, não há limite para definir as respostas corretas na pergunta.

```
QuizController.cs
23  public class QuizQuestion
24  {
25      [Serializable]
        13 referências
26      public struct QuestionAnswer
27      {
28          [TextArea]
29          public string answer;
30          public bool isCorrect;
31
            4 referências
32          public QuestionAnswer(string answer, bool isCorrect)...
37      }
38
39      [TextArea]
40      public string question;
41      public List<QuestionAnswer> answers;
```

Figura 3.15 – Código da subclasse *QuizQuestion* e da struct *QuestionAnswer*

O componente de questionário tem o intuito de acomodar o método ativo de aprendizado descrito na pirâmide de aprendizado de Glasser (Figura 2.3), sendo esse o terceiro andar. Nota-se que possui uma taxa de aprendizagem de 70% e por ser um meio ativo, pode ter uma importância especial na confecção de salas de aula virtuais.

Não é um componente que necessita de downloads ou carregamentos pesados para o seu funcionamento básico. O componente recebe apenas textos e os distribui da maneira que o usuário deseja nos vários objetos de jogo que podem ser referenciados dentro do script *QuizController.cs* (Figura 3.14). O processo de carregamento dos dados armazenados na LMS é similar ao do componente de slides, porém como o componente de questionário possui uma estrutura de classes

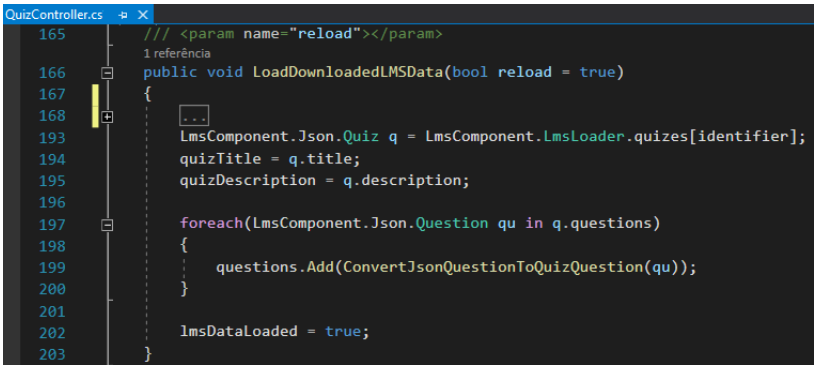

Figura 3.16 – Código da funcionalidade de carregamento dos dados baixados pela LMS, que dizem respeito ao componente de questionário

mais robusta, a interpretação dos dados deve passar por uma etapa de tradução de dados para acomodar as classes em questão.

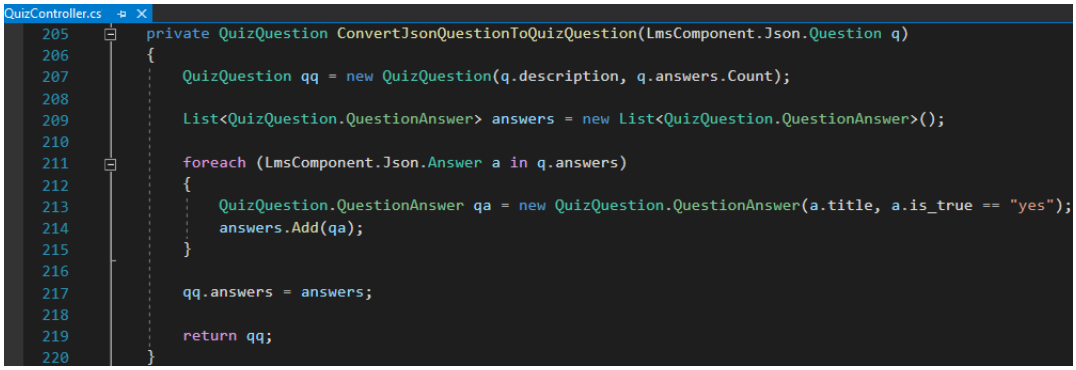

Figura 3.17 – Código que diz respeito à tradução dos dados originados do *Json* para as classes do componente do Quiz

O *prefab* incluso no pacote é um botão para começar o questionário, com um *canvas* que só aparecerá após o botão ser ativado. Dentro do *canvas* por sua vez, há um campo de texto especificando o título, subtítulo, mensagem de status para especificar se a resposta foi correta ou não, enunciado e espaço para 4 respostas (Figura 3.13).

3.3 Testes Unitários

Cada componente desenvolvido nesse projeto também possui um conjunto de testes correspondente as suas funcionalidades. Os testes são voltados para analisar o comportamento geral de cada função da classe central do componente, fornecendo um ambiente controlado e respostas esperadas. Foi uma etapa muito importante durante o desenvolvimento da versão atual do pacote, pois permitiu a visualização clara de inconsistências no código, além de facilitar um aprimoramento de coesão do mesmo, tornando todas as funcionalidades dos componentes mais separadas e modulares do que eram inicialmente (BECK, 2003).

É importante notar que a metodologia de *Test-Driven Development* não foi utilizada para criar os componentes existentes, uma vez que a sua importância foi percebida após a implementação propriamente dita dos mesmos. Contudo, é aconselhável que futuros componentes que potencialmente serão implementados neste pacote utilizem essa metodologia, para que seja possível fazer com que as funcionalidades de cada componente permaneçam padronizadas de acordo com os componentes que já existem de maneira simples e direta.

Foi utilizado o framework NUnit para C#, que fornece as ferramentas necessárias para a programação dos testes unitários. A Unity Engine provê suporte para o framework NUnit, além de fornecer uma interface gráfica para o desenvolvedor conseguir gerenciar os testes falhos, bem sucedidos ou inconclusivos.

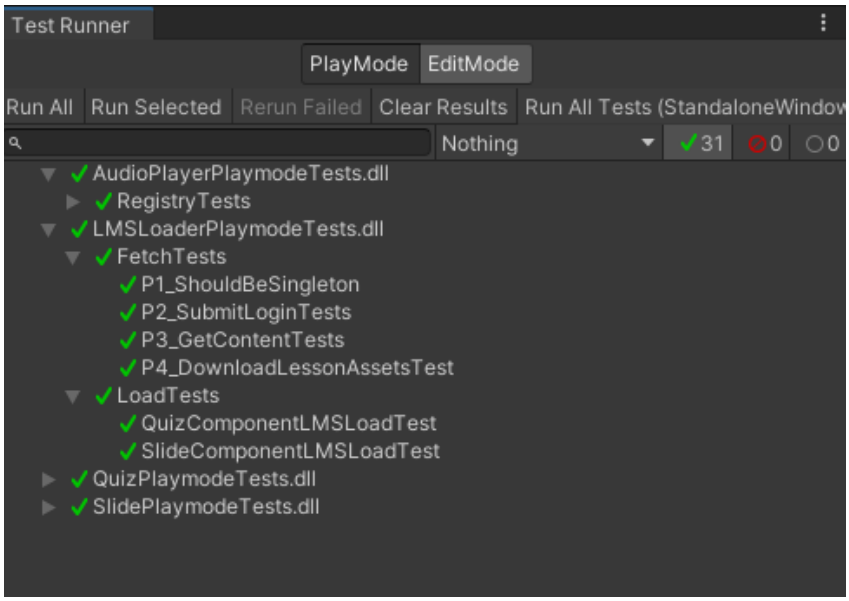

Figura 3.18 – Interface fornecida pela *Unity Engine* para visualização da execução dos testes

Os testes são subdivididos de acordo com os componentes programados no pacote e as funcionalidades implementadas nos mesmos (Figura 3.18). A *dll* que contém os testes do componente de reprodução de áudio por exemplo, testam as funcionalidades específicas para o funcionamento do componente no seu próprio escopo, como o comportamento do objeto ou as comunicações com a *LRS* através dos *StatementSenders*, ou seja, é realizado o teste para todas as funções que não possuem dependência com outro componente, mais especificamente o *LMS Loader*. O mesmo pode ser dito para os testes referentes aos componentes de apresentação de slides e questionário.

O componente de autenticação com a LMS é um pouco diferente em relação a desenvolvimento de testes, já que suas funcionalidades são muito dependentes de conexão com internet e avaliação de respostas a requisições. Dito isso, muitos testes podem retornar o status "inconclusivo"caso a conexão com a internet esteja ausente ou alguma URL esteja indisponível. O teste para autenticar o cliente com a LMS utiliza uma conta teste para realizar o mesmo. Após a

autenticação bem sucedida, é feito o download do conteúdo de uma aula teste armazenada na LMS, de ID 403. Vale ressaltar que qualquer alteração realizada na LMS para essa lição deve ser refletida nos testes implementados no pacote, caso contrário os mesmos irão falhar, pelo fato de receberem informações do servidor diferentes das esperadas.

```
[SetUp]
0 referências
public void Setup()
{
    GameObject go = new GameObject();
    testSubject = go.AddComponent<SlideController>();

    testSubject.slideFiles = new List<SlideController.SlideRegistry>
    {
        new SlideController.SlideRegistry("test slide 1"),
        new SlideController.SlideRegistry("test slide 2", "https://cdn.pixabay.com/
          photo/2014/06/03/19/38/board-361516_960_720.jpg"),
        new SlideController.SlideRegistry(new Texture2D(10, 10), "test slide 3"),
        new SlideController.SlideRegistry("test slide 4", "https://cdn.pixabay.com/
          photo/2017/06/28/10/53/board-2450236_960_720.jpg")
    };
}
```

Figura 3.19 – Função *SetUp* que cria o ambiente de testes para a execução de um teste unitário referente ao componente de apresentação de slides

Cada teste unitário necessita de um ambiente montado para ser executado, por isso, é importante que seja criado um ambiente que englobe todas as características que serão testadas. O framework *NUnit* disponibiliza uma funcionalidade chamada *SetUp*, um atributo de função que indica que a função atribuída será executado antes de cada teste unitário (Figura 3.19). Isso garante que cada teste possua um ambiente isolado sem a interferência de alterações que outros testes potencialmente podem fazer nas estruturas declaradas (MICROSOFT, 2022b).

3.4 Construção de um Componente

A construção de um novo componente deve seguir a arquitetura descrita neste trabalho, adotando a metodologia de *Test-Driven Development*, e pode ser subdividida em algumas etapas, respeitando a estrutura de diretórios definida (Figura 3.1).

3.4.1 Levantamento e planejamento das Funcionalidades do Componente

Uma etapa puramente conceitual que tem como objetivo confeccionar quais funcionalidades o componente irá conter, no que diz respeito a quais funções a classe central que gerencia o comportamento do mesmo possuirá. Usando o componente de slides como exemplo, pode-se dizer que as suas funcionalidades se baseiam em: Armazenar os slides em uma estrutura de dados adequada, fornecer formas de carregar as imagens, fornecer funções de navegação em uma apresentação de slides, fornecer suporte para download de imagens diretamente da internet utilizando URL e por fim criar declarações xAPI para o registro de experiência de usuário.

3.4.2 Desenvolvimento dos testes

Com as funcionalidades confeccionadas, parte-se para o desenvolvimento dos testes utilizando a metodologia *Test-Driven Development*, isto é, os testes para cada funcionalidade são desenvolvidos, chamando a função específica e definindo a priori quais os retornos desejados para cada uma. Inicialmente os testes irão naturalmente falhar por falta de código implementado, mas servirão de molde para o desenvolvimento controlado das funcionalidades na próxima etapa.

3.4.3 Desenvolvimento das funcionalidades e conteúdo

Cada funcionalidade desenvolvida nesta etapa deve ser criada com o objetivo de passar nos testes desenvolvidos na segunda etapa. Como cada componente possui uma natureza diferente, cabe ao desenvolvedor codificar da maneira que lhe agrada, desde que respeite os conceitos da arquitetura descritos neste trabalho. Os scripts que dizem respeito ao comportamento do objeto no jogo (*MonoBehaviour*) devem estar contidos na pasta *Runtime*, já os scripts relacionados à interface do componente para com a Unity Engine devem estar contidos na pasta *Editor*.

Nota-se que cada componente desenvolvido até aqui possui um conjunto de conteúdos gráficos que servem de auxílio para a montagem do *prefab* correspondente. O *prefab* deve ser desenvolvido de maneira que o usuário do pacote não tenha muito trabalho em configurá-lo para a condição de uso dentro do jogo.

3.4.4 Configuração das especificações do pacote

Como o projeto se trata de um pacote da *Unity*, há alguns requisitos a serem seguidos para que o pacote em si funcione de maneira adequada. No arquivo *package.json* são descritas todas as características do pacote que por sua vez são interpretadas pela Unity Engine no momento de instalação do mesmo. Caso um componente novo seja adicionado ao repositório, é necessário adicionar um novo elemento na lista "samples"do arquivo, além de atualizar o número da versão do pacote. Quaisquer especificações extras a serem adicionadas podem ser conferidas na documentação Unity (2022).

3.5 Implantação dos Componentes em um Projeto

Foram criados dois vídeos em inglês com o propósito de explicar passo a passo o processo de instalação do pacote dentro de um projeto da *Unity*, para incrementar a documentação do mesmo. [1] [2]

[1] https://youtu.be/RUUmode2vJI
[2] https://youtu.be/CnGQR6OX7aQ

4 Considerações finais

4.1 Aplicação dos Componentes

Os componentes apresentados neste trabalho foram devidamente implementados dentro do jogo desenvolvido por Nunes (2022), substituindo assim alguns aspectos de código que a versão antiga do jogo possuía. A aplicação foi feita de maneira que o componente *LMS Loader* fosse alocado na cena inicial de "Menu Principal", para coletar os dados de autenticação, autenticar o usuário e por fim baixar o conteúdo da lição escolhida durante a tela de carregamento. Com isso, os componentes de reprodução em conjunto com o componente de *LRS Information* foram implantados na cena que compõe o laboratório virtual, utilizando os dados baixados pelo *LMS Loader* durante a etapa de carregamento. É importante ressaltar também que um novo script auxiliar chamado *InterfaceHandler.cs* foi criado dentro do escopo do jogo para gerenciar a interface de menu principal do mesmo, utilizando as funções fornecidas pelo *LMS Loader* (Figura 4.1).

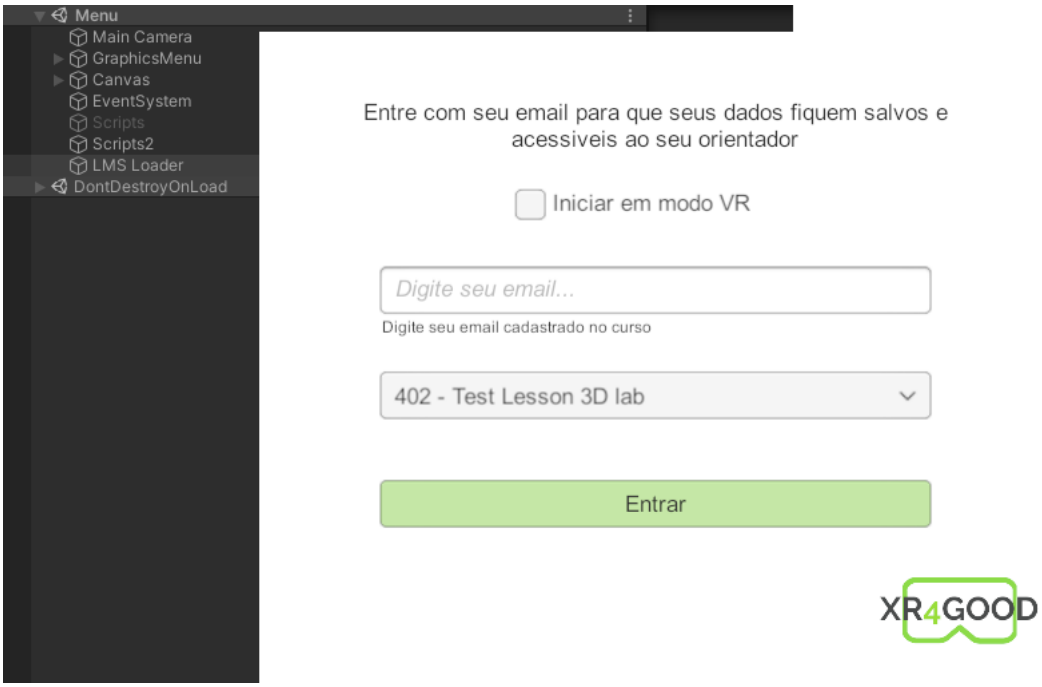

Figura 4.1 – Interface do jogo utilizando o componente *LMS Loader*

4.2 Conclusão

Um dos principais propósitos deste trabalho foi fornecer um pacote dentro de um ambiente de desenvolvimento modularizado, para que no futuro seja possível uma implementação sem

grandes complicações de um sistema de customização/criação de salas de aulas dentro do próprio jogo, sem que seja necessário um conhecimento prévio de programação ou da ferramenta *Unity*.

A premissa por trás dessa ideia surgiu com o levantamento dos problemas que um jogo estático e exclusivo para Realidade Virtual pode trazer. Um jogo focado no ensino sem nenhum tipo de customização implementada serviria apenas para nichos pequenos de pessoas, sendo estes os que se caracterizam como usuários que usariam o conteúdo de uma aula que já foi programada, ou de usuários que já possuem conhecimento prévio de como usar a *Unity Game Engine* para programar suas próprias particularidades no jogo.

Até o presente momento, foi criada a arquitetura que os componentes a serem usados no jogo sério para a ensino, criado por Nunes (2022) usarão. Porém, os componentes não são exclusivos para esse jogo, podendo ser usados em qualquer projeto que queira implementá-los com o intuito de usar a tecnologia xAPI dentro do *Unity Engine*, e podem ser encontrados em um repositório do *GitLab* [1].

Além da arquitetura, foram criados também 5 componentes (áudio, slide, questionário, interface com LMS, interface com LRS), com seus respectivos conteúdos, scripts e testes, que por sua vez servirão de base para a implementação de futuras funcionalidades e novos componentes no repositório, seguindo a arquitetura proposta e as instruções listadas neste trabalho para desenvolvimentos posteriores.

Um dos objetivos de longo prazo é disponibilizar o pacote desenvolvido no registro de pacotes da própria *Unity Engine*, além de disponibilizar as funcionalidades programadas até aqui em diferentes versões na *Unity Asset Store* de forma gratuita, para que qualquer pessoa possa baixar os componentes e implementá-los em seu projeto do *Unity Engine*. Não foi possível realizar essa tarefa dentro do tempo do projeto devida a grande quantidade de exigências e regras a serem seguidas para postagem do pacote na plataforma, portanto é algo que pode ser considerado para trabalhos futuros.

4.2.1 Trabalhos Futuros

Essa monografia explica detalhadamente o processo de desenvolvimento e lógica da arquitetura confeccionada para a criação dos componentes aqui implementados, portanto, possíveis trabalhos futuros poderiam estar envolvidos em implementar novos componentes didáticos para compor o pacote. Além disso, quaisquer alterações na arquitetura podem também ser estudadas em busca de uma metodologia mais interessante do que a proposta nesse projeto.

[1] https://gitlab.com/seriousgamesplatform1/gamecomponents-package.git

Referências

ABT, C. Serious games. e viking press. *New York City, New York, USA, st edition*, 1970.

ADL. *What is an LRS? Learn more about Learning Record Stores*. 2021. Disponível em: <https://xapi.com/learning-record-store/>.

BECK, K. *Test-driven development: by example*. [S.l.]: Addison-Wesley Professional, 2003.

BERKING, P.; GALLAGHER, S. Choosing a learning management system. *Advanced Distributed Learning (ADL) Co-Laboratories*, v. 14, p. 40–62, 2013.

BRIAND, L. C.; DALY, J. W.; WUST, J. K. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering*, IEEE, v. 25, n. 1, p. 91–121, 1999.

BUSCH, I. C. 8th european conference on games based learning. 2014.

COBERN, W. W.; AIKENHEAD, G. Cultural aspects of learning science. 1997.

DJAOUTI, D.; ALVAREZ, J.; JESSEL, J.-P.; RAMPNOUX, O. Origins of serious games. In: *Serious games and edutainment applications*. [S.l.]: Springer, 2011. p. 25–43.

FUCHS, P. *Le traité de la réalité virtuelle*. [S.l.]: Presses des MINES, 2006. v. 2.

GLASSER, W. *Choice theory: A new psychology of personal freedom.* [S.l.]: HarperPerennial, 1999.

HAAS, J. A history of the unity game engine. *Diss. WORCESTER POLYTECHNIC INSTITUTE*, 2014.

HOCK, W.; SCHITTKOWSKI, K. Test examples for nonlinear programming codes. *Journal of optimization theory and applications*, Springer, v. 30, n. 1, p. 127–129, 1980.

HOULISTON, T.; FOUNTAIN, J.; LIN, Y.; MENDES, A.; METCALFE, M.; WALKER, J.; CHALUP, S. K. Nuclear: A loosely coupled software architecture for humanoid robot systems. *Frontiers in Robotics and AI*, Frontiers, v. 3, p. 20, 2016.

LAAMARTI, F.; EID, M.; SADDIK, A. E. An overview of serious games. *International Journal of Computer Games Technology*, Hindawi, v. 2014, 2014.

LIM, K. C. Case studies of {xAPI} applications to e-learning. In: *The twelfth international conference on eLearning for knowledge-based society*. [S.l.: s.n.], 2015. p. 3–1.

MELO, K. G. F. Uma arquitetura de integração de learning management system e learning record system para jogos sérios. 2022.

MICHAEL, D. R.; CHEN, S. L. *Serious games: Games that educate, train, and inform*. [S.l.]: Muska & Lipman/Premier-Trade, 2005.

MICROSOFT. *dotNET Documentation*. [S.l.], 2022. Disponível em: <https://learn.microsoft.com/en-us/dotnet/>.

MICROSOFT. *Um tour pela linguagem C#*. [S.l.], 2022. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/tour-of-csharp/>.

MIKROPOULOS, T. A.; NATSIS, A. Educational virtual environments: A ten-year review of empirical research (1999–2009). *Computers & education*, Elsevier, v. 56, n. 3, p. 769–780, 2011.

NUNES, A. J. d. A. Jogos sérios em realidade virtual para ensino: um caso de estudo com aulas laboratoriais em ciências biológicas. 2022.

PADHY, N.; PANIGRAHI, R.; BABOO, S. A systematic literature review of an object oriented metric: reusability. In: IEEE. *2015 international conference on computational intelligence and networks*. [S.l.], 2015. p. 190–191.

RAMLER, R.; MOSER, M.; PICHLER, J. Automated static analysis of unit test code. In: IEEE. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.], 2016. v. 2, p. 25–28.

RENTSCH, T. Object oriented programming. *ACM Sigplan Notices*, ACM New York, NY, USA, v. 17, n. 9, p. 51–57, 1982.

STENCEL, K.; WĘGRZYNOWICZ, P. Implementation variants of the singleton design pattern. In: SPRINGER. *OTM Confederated International Conferences"On the Move to Meaningful Internet Systems"*. [S.l.], 2008. p. 396–406.

UNITY. *Unity User Manual 2021.3 (LTS)*. [S.l.], 2022. Disponível em: <https://docs.unity3d.com/Manual/index.html>.

VIDAKIS, N.; BARIANOS, A. K.; TRAMPAS, A. M.; PAPADAKIS, S.; KALOGIANNAKIS, M.; VASSILAKIS, K. Generating education in-game data: The case of an ancient theatre serious game. In: *CSEDU (1)*. [S.l.: s.n.], 2019. p. 36–43.

XIONG, J.; HSIANG, E.-L.; HE, Z.; ZHAN, T.; WU, S.-T. Augmented reality and virtual reality displays: emerging technologies and future perspectives. *Light: Science & Applications*, Nature Publishing Group, v. 10, n. 1, p. 1–30, 2021.

Apêndices

APÊNDICE A – Arquivo README.md no repositório do pacote

Serious Game Components

Descrição

Esse projeto é focado na criação de diferentes componentes didáticos que ajudam o usuário a criar um ambiente de sala de aula compacto e diversificado dentro da Unity Engine.

xAPI é usada para capturar todas as interações do jogador em relação aos componentes propriamente ditos, construindo diferentes declarações xAPI de acordo com a maneira na qual o jogador interage com esses objetos, e então enviando-os para a LRS desejada.

Atualmente, há três componentes disponíveis para uso, sendo estes:

Audio Player

- Um componente que pode guardar uma lista de arquivos de áudio e reproduzí-los de acordo com a ordem da mesma. Ele captura quanto tempo o jogador ouviu a um áudio específico e também se o mesmo jogador ouviu todos os áudios.

Quiz

- Armazena um conjunto de perguntas e respostas. A quantidade de respostas pode ser especificada pelo usuário, assim como a quantidade de respostas corretas pra cada pergunta. É caputurada a quantidade de tempo que o jogador ficou em uma pergunta específica, se ela foi respondida corretamente ou não, assim como se o jogador respondeu a todas as perguntas.

Slides

- Um componente que armazena uma lista de imagens que podem representar qualquer coisa, como por exemplo uma apresentação de slides. Captura quanto tempo o jogador ficou em um slide e se o mesmo visualizou todas as imagens.

Figura A.1 – Arquivo README.md parte 1

LRS Information

- Um componente relativamente simples que faz com que as credenciais de autenticação para a LRS sejam generalizadas para uso de todos os componentes, ao invés de atribuí-las individualmente para cada um.
- Note que apenas um script de LRS Information é permitido por cena, qualquer dúplica **será** destruída assim que o jogo começar.

LMS Loader

- Um componente que gerencia a comunicação com a XR4Good LMS para os outros componentes, autenticando e baixando os dados de uma lição específica para carregamento de seus respectivos componentes.
- Um LMS Loader não precisa necessariamente existir na mesma cena dos outros componentes para que a informação baixada seja carregada, uma vez que tal informação já tenha sido baixada.
- Note também que apenas um script de LMS Loader é permitido por cena, qualquer dúplica **será** destruída assim que o jogo começar.

Instalação

- Para abrir e carregar o pacote no Unity, você precisa ter o Git instalado e configurado no seu computador.
- Vá para a aba Window e abra o **Package Manager**
- Clique o ícone de + e então selecione **Add package from git URL**
- Insira a URL desse repositório e então clique em **Add**
- Espere pela conclusão do download e instalação do pacote
- Depois disso, você pode abrir a seção **Samples** e importar quaisquer Assets dos componentes já desenvolvidos para um prefab pré-

Videos Explicativos

- Instalação e Implantação - Parte 1 (Inglês)
- Instalação e Implantação - Parte 2 (Inglês)

Utilização

Após a instalação, a utilização é bem simples. Você pode usar os prefabs disponíveis na seção Samples ou então pode criar seu próprio objeto usando os scripts dos componentes disponíveis nesse pacote.

Para usar os componentes, apenas arraste os prefabs para a cena desejada e ajuste as configurações de cada componente no seu respectivo Unity Inspector. As configurações padrão estão configuradas para carregar o conteúdo baixado da LMS pelo LMS Loader automaticamente.

Figura A.2 – Arquivo README.md parte 2

APÊNDICE B – Arquivo CHANGELOG.md especificando as mudanças realizadas no repositório até aqui

1.0.0

2022-09-02

Featuring changes

- Created package

1.0.1

2022-09-29

Featuring changes

- SaveStatement method in BaseStatementSender.cs is now async, meaning that the game will not stutter anymore when waiting for the LRS response

Bug Fixes

- Adjusted some inconsistencies in the Components prefabs
- Fixed a bug where the QuizController woudn't work if initialized with zero Questions in the registry, even if the questions were added in the future
- Fixed a bug where a NullReferenceException was being thrown at SlideController.cs in the Start method

Figura B.1 – Arquivo CHANGELOG.md parte 1

1.1.0

2022-10-09

This update mainly focused in developing a way to communicate with a pre-designed LMS and facilitate the LRS comunication by making the deployment a bit simpler.

Featuring Additions

- Added the LMS Loader component. The LMS Loader manages the project communication with the pre-designed XR4GOOD LMS. It downloads the data from a specific lesson for the components to use if they are set to do so. It is a Singleton, so only one LMS Loader script can exist in a scene.
- Added the LRS Information component. This is a simple component that stores authentication data for any wished LRS. It is a Singleton, so only one LRS Information script can exist in a scene.
- The Slide and Quiz components can load the downloaded data from the LMS by the LMS Loader automatically if they are set to do so by setting the "use LMS Loader" flag to true.
- Added a variable on the Slide Component to specify a default slide for broken images
- Added a Tests folder to the package.

Featuring Changes

- Changed Slide, Quiz and Audio components inspectors to accommodate the new LMS Loader addition.
- Changed StatementSenders inspectors to accommodate the new LRS Information addition.

Bug Fixes

- Fixed a bug where the Slide Component were loading duplicate images on the last slide

1.1.1

2022-10-10

Featuring Additions

- Each practical component now has a prefab for VR usage, using the STEAM VR Library
- Quiz Controller script now has a way to track the current selected answer

Figura B.2 – Arquivo CHANGELOG.md parte 2

1.1.2

2022-10-12

Featuring Additions

- Added tests for Audio Player Component
- Added tests for Quiz Component
- Added tests for Slide Component
- Added tests for LMS Loader Component

Bug Fixes

- Adjusted component dependencies on editor instantiated variables. Now it instantiates any uninstantiated variables in the Start method
- Fixed Audio Player component not registering what actual audio index the component is reproducing

Figura B.3 – Arquivo CHANGELOG.md parte 3