Componentes Governados por Dados, Shell um Framework para Unity 3D

Trabalho de Conclusão de Curso

Engenharia da Computação

Felipe Jorge Pereira

Orientador: Prof. Thiago Farias

Felipe Jorge Pereira

Componentes Governados por Dados, Shell um Framework para Unity 3D

Monografia apresentada como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco - Universidade de Pernambuco.

Engenharia da Computação

Escola Politécnica de Pernambuco

Universidade de Pernambuco

Orientador: Prof. Thiago Farias

Recife - PE, Brasil

julho de 2018

Felipe Jorge Pereira
Componentes Governados por Dados, Shell um Framework para Unity 3D/ Felipe Jorge Pereira. – Recife - PE, Brasil, julho de 2018-
46 p.
Orientador: Prof. Thiago Farias

Trabalho de Conclusão de Curso – Engenharia da Computação
Escola Politécnica de Pernambuco
Universidade de Pernambuco, julho de 2018.

1. Unity 3D. 2. Game. 2. Gamedev. 3. Shell. 4. Data. 5. Framework. I. Prof. Thiago Farias II. Universidade de Pernambuco. III. Escola Politécnica. IV. Componentes Governados por Dados, Shell um Framework para Unity 3D

Este trabalho é dedicado à minha família, meus amigos e aqueles que me ajudaram ao longo do caminho.

Escola Politécnica de Pernambuco
Graduação em Engenharia de Computação
Projeto de Final de Curso

ESCOLA POLITÉCNICA
DE PERNAMBUCO

MONOGRAFIA DE FINAL DE CURSO

Avaliação Final (para o presidente da banca)*

No dia 13 de julho de 2018, às 11:00 horas, reuniu-se para deliberar a defesa da monografia de conclusão de curso do discente **FELIPE JORGE PEREIRA**, orientado pelo professor **Thiago Souto Maior Cordeiro de Farias**, sob título **Componentes Governados por Dados, Shell um Framework para Unity 3D**, a banca composta pelos professores:

Byron Leite Dantas Bezerra
Thiago Souto Maior Cordeiro de Farias

Após a apresentação da monografia e discussão entre os membros da Banca, a mesma foi considerada:

☒ Aprovada ☐ Aprovada com Restrições* ☐ Reprovada

e foi-lhe atribuída nota: 8,0 (oito)

*(Obrigatório o preenchimento do campo abaixo com comentários para o autor)

O discente terá 5 dias para entrega da versão final da monografia a contar da data deste documento.

BYRON LEITE DANTAS BEZERRA

THIAGO SOUTO MAIOR CORDEIRO DE FARIAS

* Este documento deverá ser encadernado juntamente com a monografia em versão final.

Agradecimentos

Sou grato pela minha família, em especial aos meus pais porque foi graças a seus esforços que cheguei onde estou. Agradeço ao meu irmão por ter me apoiado e ter sido compreensível.

Ao longo do curso tive ótimos professores que me ensinaram e me mostraram o caminho. Também fiz grandes amizades; foram bons momentos que levarei para sempre comigo.

À minha família e amigos aos meus professores. Completar um curso de engenharia foi uma tarefa árdua e teria sido impossível sem a ajuda de todos.

"Para ter algo que você nunca teve,
é preciso fazer algo que você nunca fez."
Chico Xavier

Resumo

Essa monografia destina-se a explicar o desenvolvimento do *framework* Shell, para Unity 3D, que une vários padrões empregados na criação de objetos de dados e como podem ser utilizados em componentes de lógica. Tais padrões foram criados com a intenção de auxiliar na estruturação e desenvolvimento de código para jogos, que dada sua natureza imprevisível precisa de um código flexível. Como metodologia para selecionar os padrões, foram escolhidos os padrões mais recorrentes utilizados na produção de vários jogos. Esses padrões listados foram apresentados nos últimos dois anos em conferências, e são utilizados na indústria por desenvolvedores independentes e grandes estúdios. Com essa monografia conclui-se que utilizar objetos de dados para parametrizar e interligar partes chaves da lógica do jogo permite escrever lógica modular, criando grandes oportunidades de reaproveitamento e casos de testes unitários.

Palavras-chave: Unity 3D. Desenvolvimento. Jogos. *Framework*. Shell. Lógica Modular. Reaproveitamento.

Abstract

This monograph aims to compile and develop the Shell framework for Unity 3D, which unites several standards used in creating data objects and how they can be used in logic components. Such standards were created with the purpose of assisting in the design and development of game code, which, given its unpredictable nature, requires flexible code. As a methodology for selecting standards, the most recurrent patterns used in the production of a game were chosen. These standards listed have been presented to us last two years at conferences, and are used in the industry by independent developers and large studios. It is concluded with this monograph that using data objects to parameterize and interconnect key parts of the game logic allows: easy editing of the behavior of the logic by means of parameter adjustments, writing modular logic creating great opportunities for reuse and unit test cases.

Keywords: Unity 3D. Game development. Game. Framework. Shell. Logic. Modular .Reuse.

Lista de ilustrações

Lista de abreviaturas e siglas

HP	Health Points
FPS	Frames Per Second
DI	Dependency Injection
HUD	Head's Up Display
GUID	Globally Unique IDentifier
GPU	Graphics Processing Unit
URI	Uniform Resource Identifier
BHT	BeHavior Tree
API	Application Programming Interface
GC	Garbage Collector
ECS	Entity Component System

Sumário

1 Introdução

Essa monografia destina-se a desenvolver e discutir um projeto de framework chamado Shell, que utiliza objetos dados para interligar os mais diversos sistemas e componentes de um jogo.

Do ponto de vista da engenharia criar jogos assemelha-se com à criação de qualquer outro tipo de programa. Porém, desenvolver jogos possui seus próprios desafios, que exige um grande esforço de várias outras disciplinas como: artes plásticas e música.

1.1 Motivação e Caracterização do Problema

Durante sua produção, os jogos, mudam constantemente para atender ao propósito de ficarem mais divertidos. Essas constantes mudanças representam um grande desafio do ponto de vista técnico da produção de um jogo.

Visando acomodar as constantes mudanças, que um jogo sofre ao longo de seu desenvolvimento, surgiram nos últimos dois anos novos conceitos e idéias. Como é o caso dos princípios S.O.L.I.D. (1), conceito já estabelecido durante o desenvolvimento de software. Do Toy Boxing batizado por Janson Story e apresentado por Elder (3). E finalmente os três pilares de arquitetura para jogos proposto por Hipple (2), que são: Modularidade, Editabilidade e Depurabilidade.

É crucial compilar esses novos conceitos para que desenvolvedores de jogos os utilizem de forma coerente. Para assim, melhorar seus fluxos de trabalho e a qualidade de seus jogos.

1.2 Objetivo Geral

Essa monografia tem como objetivo definir um framework conciso baseado em ideias de Hipple (2), Elder (3) e nos príncipios S.O.L.I.D. (1). Que são dedicadas a organizar código de jogos utilizando estruturas de dados.

1.3 Objetivos Específicos

O projeto envolve os seguintes tópicos:

- Entender como ativos contendo apenas dados podem facilitar o desenvolvimento de jogos;
- Definir o *framework* Shell;
- Melhorar os inspetores do Unity 3D para que suportem interfaces e campos abstratos;
- Verificar como os conceitos de Shell podem ser aplicados em outras *game engines*;

1.4 Estrutura da Monografia

O documento está dividido em quatro grandes seções: Primeiro, Fundamentação Teórica, que descreve as principais ideias, conceitos e ferramentas envolvidas com Shell; Segundo, Padrões e técnicas que lista todos os padrões implementados no Shell, explicando suas nuances e como podem ser utilizados; Terceiro, Inspetor avançado, que explica o conceito do inspetor como sistema de Dependency Injection (DI);

Quarta, Outras *Engines*, que apresenta implementações dos padrões propostos por Shell, nas game engines *Unreal* e *Godot*.

Por fim, essa monografia finaliza discutindo sobre o impacto de Shell na produção de jogo publicado comercialmente e sobre o que ainda resta a ser desenvolvido.

2 Fundamentação Teórica

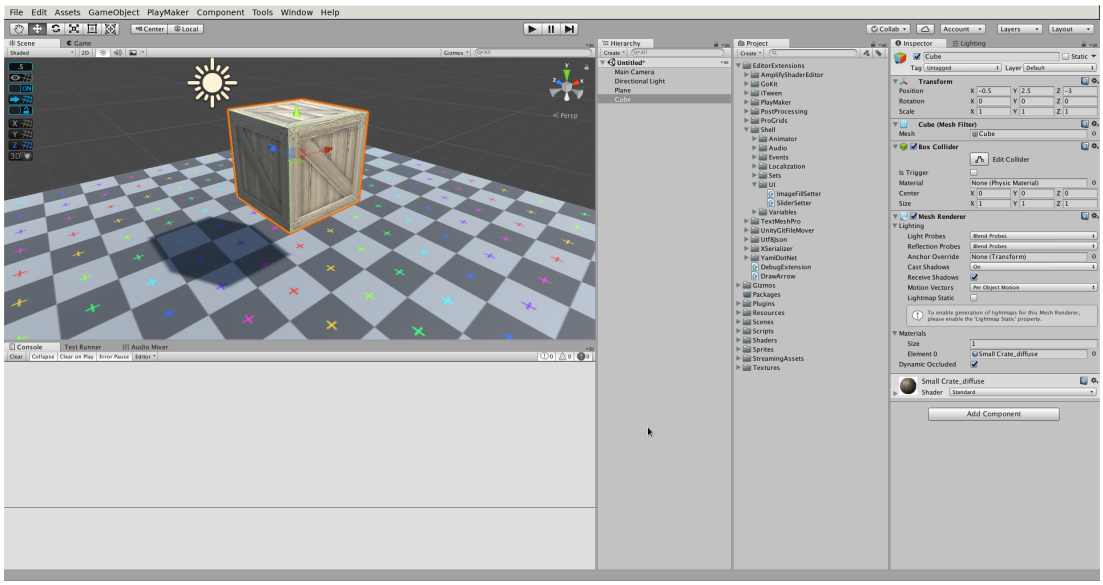
Essa seção apresenta conceitos que serão utilizados durante todo o trabalho. É importante salientar que as próximas seções não são destinadas a explicar profundamente cada um desses conceitos. Mas destina-se em prover uma fundamentação básica para o entendimento do trabalho.

2.1 Unity 3D

Unity 3D é uma *game engine* moderna escrita em C++ com uma camada de *script* em C# que é utilizada para criar a lógica do jogo. Ela é capaz de exportar jogos para as plataformas: Windows, Mac, Linux, Android, iOS e consoles dos quais estão inclusos *Xbox*, *Playstation* e *Switch*. A figura 1 mostra uma foto do Unity 3D. O Editor está disponível para Windows, Mac e Linux.

O modelo estrutural de uma *game engine* descreve como novas funcionaliades são adicionadas nas entidades do jogo. O Unity 3D tem o modelo estrutural baseado em componentes. Assim novas funcionalidades são adicionados nas entidades por meio da adição de componentes. No caso do Unity 3D essas entidades são chamadas de *GameObject*. Não existe limite para o número de componentes que um *GameObject* pode ter, porém todo *GameObject* possui o componente *Transform* que é responsável por representar a localização espacial da entidade no nível.

Figura 1 – Interface do Unity 3D.



Fonte: o autor.

Todos os objetos e arquivos utilizados para a criação do jogo são denominados ativos ou em inglês *Assets*. No Unity 3D, cada um dos ativos é gerenciado por meio de um arquivo de meta-dados, que guarda dentre outras informações uma *GUID*. Internamente o Unity 3D garante que um mesmo ativo, carregado em partes diferentes do código referencie o mesmo objeto em memória.

Dentre os tipos de ativos que um projeto pode ter, dois tipos são de interesse particular dessa monografia, são eles: *GameObjects* e *ScriptableObjects*.

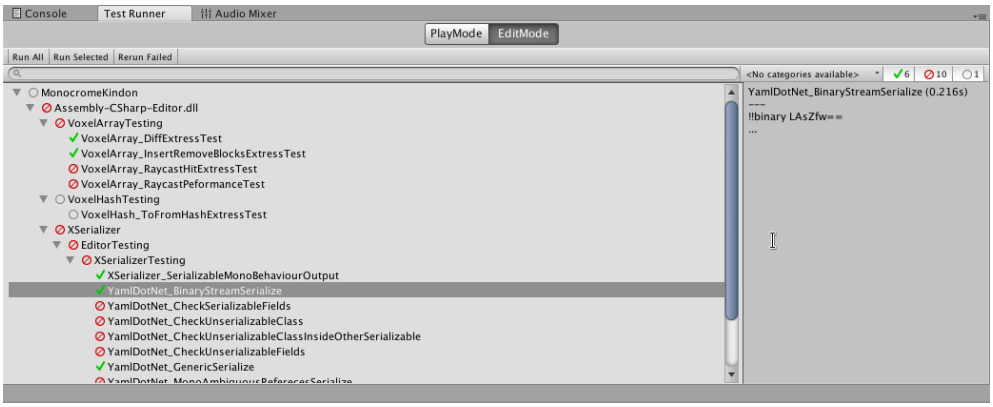
Os *GameObjects* são as entidades do Unity 3D. Um *GameObject* representa qualquer objeto que pode ser colocado e exibido em um nível. Eles também podem existir como ativo no projeto no formato de *Prefabs*.

Prefabs são protótipos de *GameObject*, i.e. objetos pré-fabricados utilizados para instanciar tipos recorentes de *GameObject*.

ScriptableObject é a classe base de qualquer ativo que pode ser customizado.

O Unity 3D possui um sistema de testes chamado *Test Runner*, figura 2, que permite acessar as funções internas da *game engine*, como criação de níveis, controle de animações entre outras.

Figura 2 – *Test Runner*, painel de testes do Unity.



Fonte: o autor.

É importante salientar que o Unity 3D precisa controlar os construtores de todas as classes derivadas de *UnityEngine.Object* ou seja, todos os ativos do jogo. A príncipal consequência disso é que sistemas convencionais de *Dependency Injection* (DI) e serialização não podem ser utilizados diretamente no Unity 3D.

2.2 Data Binding

Data Binding remete a uma técnica empregada para simplificar o desenvolvimento de software. Essa técnica mantêm duas fontes de dados sincronizadas, abstraindo a lógica necessária para enviar as informações entre os diferentes domínios da aplicação.

2.3 S.O.L.I.D. e Dependency Injection

S.O.L.I.D. é um acrônimo para cinco princípios de design destinados à criação de software. Tornando-a mais robusta, flexível e sustentável; são eles:

1. *Single Responsibility Principle*, classes que agregam muita funcionalidade no projeto são confusas e difíceis de depurar, portanto uma classe deve possuir apenas uma responsabilidade ou uma única razão para falhar.

2. *Open/Closed Principle*, classes devem ser fechadas para modificação, mas abertas para extensão, isso significa que uma vez que a classe for escrita ela não deve ser reescrita, caso contrário, todos

os testes realizados serão invalidados e é muito provável que novos bugs sejam inseridos durante a modificação;

3. *Liskov Substitution Principle*, prevê que quando duas classes possuem funcionalidades significativamente diferentes, uma não deve herdar ou estender da outra. Em outras palavras, quando duas classes não podem ser substituídas uma pela outra sem alteração das propriedades desejadas, uma não deve ser subclasse da outra ou o contrário;

4. *Interface Segregation Principle*, interfaces muito grandes devem ser separadas em unidades menores, assim classes podem se concentrar na implementação dos métodos necessários;

5. *Dependency Inversion Principle*, significa que um código deve depender de uma abstração ou interface e não de uma classe concreta. A utilização desse princípio permite abranger situações onde a adição de novas classes com novos comportamentos seja simples e não quebre funcionalidades de outras classes. Além disso, fica mais fácil escrever testes para um tipo de abstração e testar todas suas implementações.

S.O.L.I.D. é comumente utilizado com algum framework de DI que é uma de forma abstrair a construção e atribuição de campos abstratos e interfaces, tornando simples a adição de novas implementações ao código existente.

3 Revisão do Estado da Arte

Os parágrafos subsequentes apresentam um resumo dos autores referenciados nessa monografia.

A apresentação de Thomas Kinnen intitulada Data-Driven and Component-Based Game Entities (4) propõe um modelo arquitetural baseado em componentes que são unidades isoladas com comportamentos específicos, que se comunicam entre si por meio de um barramento de mensagens. Em sua proposta, o autor afirma que esses componentes são facilmente testados e portanto fáceis de serem mantidos, pois são flexíveis, e permitem uma divisão clara de trabalho entre as entidades. O modelo prevê a separação entre dados e código. Dessa forma uma entidade é definida apenas pelos dados que ela carrega.

Em sua tese, Wallentin (5) analisa a implementação teórica de sistemas com entidades baseadas em componentes, ao passo que analisa suas diferentes implementações para discutir como cada uma delas afeta o desempenho, a estruturação do projeto, o design e a interação dos jogos.

Em sua apresentação, Shumaker (6) aponta que muitos jogos não são concluídos e entre aqueles que são finalizados, muitos não atingem sucesso comercial. Segundo o autor, um jogo não é mais tecnicamente difícil de ser criado do que qualquer outro programa de computador, porém, para um jogo, tecnologia funcional é apenas um dos requisitos para o sucesso. A maior parte do tempo de desenvolvimento de um jogo é gasto no seu conteúdo e gameplay.

Comumente um jogo precisa de objetos para guardar valores padrões e estado compartilhado. Desenvolvedores que utilizam o Unity 3D comumente usam *Prefabs* para este fim, porém, isso é um abuso do conceito de *Prefab* que causa uma série de problemas deixando o projeto confuso para os próprios desenvolvedores e causando *overhead* já que um *Prefab* possui outros objetos associados a ele. Dessa forma, Fine (7) foca seus esforços em elucidar como e quando *ScriptableObjects* podem ser utilizados, exemplificando situações e padrões úteis.

Hipple (2) apresenta uma arquitetura para jogos utilizando *ScriptableObjects*; segundo ele o que falta nos desenvolvedores é o entendimento de como cada pequena parte de seus projetos irão se juntar para formar um jogo. O autor então descreve sua abordagem para uma arquitetura baseada em três pilares: modularidade, editabilidade e depurabilidade. Os exemplos mostrados por Hipple (2) segue na mesma linha de raciocínio de Fine (7), onde ambos utilizam *ScriptableObjects* para estruturar de maneira simples e coesa padrões úteis e reutilizaveis.

No seu vídeo, Elder (3) apresenta o conceito de *Toy Boxing* um estilo de codificação que vem ganhando muito apelo na comunidade Unity. O *Toy Boxing* se concentra em criar *Prefabs* (ou *Toys* como são chamados) reutilizáveis e autossuficientes, i.e. *Prefabs* que funcionam independentemente de qualquer outra entidade.

Em suas apresentações na conferência Unite 2017, Ante (8) aborda pontos principais sobre o futuro do Unity 3D e como novas tecnologias como o *Job System*, *Burst Compiler* e o Unity ECS vão mudar a forma de como programar jogos no Unity 3D.

4 Desenvolvimento

Shell é um framework destinado a lidar com estruturas de dados dentro do Unity 3D, ele provê classes abstratas derivadas principalmente de *ScirptableObject*. Essas classes abstratas são por sua vez utilizadas na criação de variáveis-chave e outras estruturas de dados globais (como ativos) ou locais (no nível), que podem ser acessados de qualquer parte do projeto e servem de meio para interligar os vários componentes do jogo.

A principal vantagem dos padrões que Shell oferece é a capacidade desacoplar componentes que antes eram fortemente interligados. Como exemplo, suponha o cenário onde é preciso exibir o nível de vida do jogador na tela. Nessa situação, a classe *HUD* dependeria da classe *Player* para exibir o valor de *HP* (do inglês *Health Points*). Porém, com Shell, a variável *HP* pode ser transformada em um ativo que será utilizado por ambas classes *Player* e *HUD*. Uma vez desacoplados, fica fácil testar isoladamente qualquer componente do jogo.

O ativo *HP* ou qualquer outro pode ser ajustado diretamente por designers sem a necessidade de engenheiros. Do ponto de vista do engenheiro adicionar e remover funcionalidades, também torna-se mais simples, o que incentiva a experimentação, que é crucial para desenvolver um jogo de sucesso.

Mudanças sempre ocorrem e são muito naturais no desenvolvimento de um jogo; a mudança pode ser a diferença entre o sucesso e o fracasso.

4.1 Padrões e Técnicas

Essa secção é destinada aos vários padrões derivados da classe *ScriptableObject*, utilizados para criar estruturas de dados que podem ser gerenciadas por designers e impactam diretamente no comportamento dos componentes presentes no jogo.

4.1.1 Variáveis Ativas

São chamadas variáveis-chave qualquer variável que é de importância direta para a mecânica do jogo. Variáveis como vida do jogador e dano de ataque dos inimigos são exemplos de variáveis-chave; ambas impactam diretamente na jogabilidade e na dificuldade do jogo.

Variáveis-chave controlam a sensação que jogo vai passar. Por isso é importante que durante o desenvolvimento do jogo essas variáveis estejam disponíveis para que designers possam ajustar seus valores. Formas comuns de expor variáveis no Unity 3D são mostradas na figura 3.

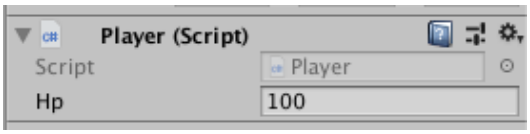
Figura 3 – Exemplos de variáveis serializadas.

```
1       [SerializeField]
2       private float m_Hp = 100;
3
4       ou
5
6       public float Hp = 100;
```

Fonte: o autor.

Os exemplos de códigos mostrados na figura 3 se colocados dentro de qualquer *MonoBehaviour* serão expostas no editor e poderão ser alteradas por qualquer um. A única diferença entre esses dois exemplos é que a váriavel *m_Hp*, no primeiro exemplo, não pode ser alterada diretamente através de código. Esse padrão é utilizado para encapsular variávies sem relevância fora da classe. A figura 4 mostra o editor das variáveis.

Figura 4 – Inspetor da classe *Player*, HP é um exemplo de variável serializada.



Fonte: o autor.

Durante o desenvolvimento de qualquer jogo é comum surgir a necessidade de acessar certas variáveis de outras partes do código. Estes acessos tornam o código acoplado. A dependência forçada de dois componentes com funções distintas, que muitas vezes estão em entidades diferentes, dificulta tanto os testes, quanto a construção de novos níveis para o jogo.

Então, Hipple (2) apresenta o conceito de Variáveis Ativas que define qualquer tipo de variável como sendo um ativo. Esse ativo por sua vez pode ser acessado por qualquer componente sem nenhuma lógica adicional. A implementação abstrata da Variável Ativa pode ser observada na figura 5.

Figura 5 – Implementação da Variável Ativa

```
1  public abstract class Variable<T> : ScriptableObject {
2    [Multiline]
3    public string DeveloperDescription = "";
4
5    [SerializeField]
6    private T m_Value;
7    public virtual T Value {
8      get { return m_Value; }
9      set { Value = value; }
10   }
11
12   public void SetValue(T value) {
13     Value = value;
14   }
15
16   public void SetValue(Variable<T> value) {
17     Value = value.Value;
18   }
19 }
```

Fonte: o autor.

Unity 3D não permite a criação de *ScriptableObjects* e nem a serializarão de genéricos (salvo o tipo *System.Collections.Generic.List*). Portanto, para que a variável possa ser criada é preciso criar uma implementação concreta da classe, mostrada na figura 6. Variaveis Ativas para tipos comuns como: *float*, *int*, *string* e *Vector3* já são implementadas por Shell.

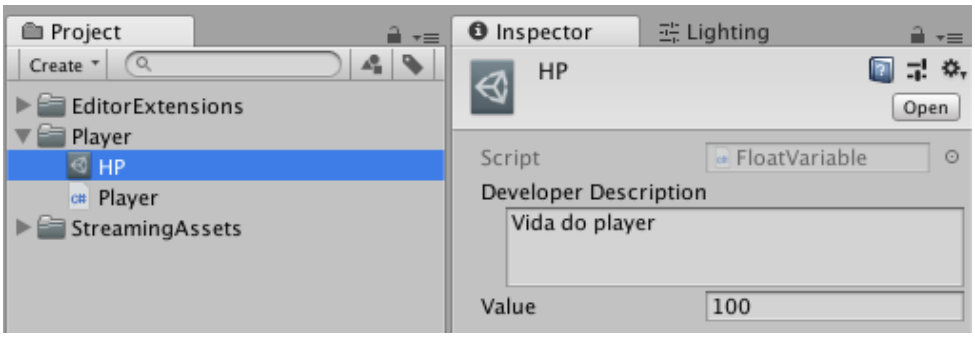
Figura 6 – Implementação concreta da classe FloatVariable uma Variável Ativa

```
1 [CreateAssetMenu]
2 public class FloatVariable : Variable<float>
3 { }
```

Fonte: o autor.

No código da figura 6, o atributo *CreateAssetMenu* adiciona um atalho para a criação do ativo. A figura 7 mostra um exemplo da classe *FloatVariable* instanciada como ativo.

Figura 7 – Exemplo de Variável Ativa.



Fonte: o autor.

Cada Variável Ativa funciona em conjunto com uma classe intermediária específica. Essa classe intermediária é utilizada para referenciar um tipo expecífico de Variável Ativa. A figura 8 mostra a implementação da referência da classe *FloatVariable* da figura 7. O propósito da classe intermediária é eliminar a necessidade de criar uma Variável Ativa para cada um dos campos de uma classe. Para tal, cada uma dessas classes intermediária mantêm um valor local do mesmo tipo que a Variável Ativa. Esse valor local pode ser utilizado caso o designer julgue que uma Variável Ativa não é necessária para a situação.

As figuras 9 e 10 mostram o inspetor customizado da classe *FloatReference*. Como apontado por Hipple (2), é importante customizar o editor para deixar o fluxo de trabalho (do inglês *workflow*) sempre intuitivo.

É importante ressaltar que Variáveis Ativas não devem ser utilizadas como substitutas para todas as variáveis de uma classe. Pelo contrário, elas devem ser utilizadas em situações específicas como conveniência para lidar com variáveis-chave que são acessadas por outros componentes.

A classe *Variable<T>* implementa Variáveis Ativas para operação em *Polling*, isto é, com checagens regulares, normalmente a cada atualização de quadro. Funções com alto custo computacional devem ser resguardadas por uma checagem para identificar se o valor realmente foi alterado desde a última execução.

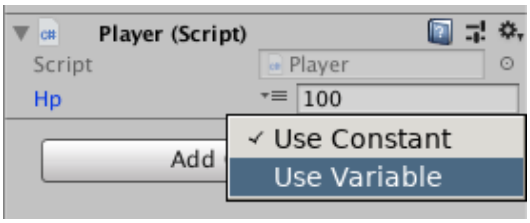
De uma maneira mais sofisticada a Variável Ativa pode ser modificada para verificar mudanças e acionar eventos, como é de costume em frameworks de Data Binding tradicionais. Porém, vale ressaltar que código baseado em *Polling* é normalmente mais simples de ser escrito, mais robusto e fácil de ser otimizado. Esse recurso permite manter a taxa de quadros por segundo (do inglês, FPS) consistente, já que a sequência de funções acionadas pela técnica é previsível antes da execução do programa. Como todas

Figura 8 – Referência para uma FloatVariable

```
1  [Serializable]
2  public struct FloatReference {
3    public bool UseConstant;
4    public float ConstantValue;
5    public FloatVariable Variable;
6
7    public FloatReference(float value) {
8      UseConstant = true;
9      ConstantValue = value;
10     Variable = null;
11   }
12
13   public float Value {
14     get {
15       return UseConstant ? ConstantValue : Variable.Value;
16     }
17   }
18
19   public static implicit operator float(FloatReference reference) {
20     return reference.Value;
21   }
22 }
```

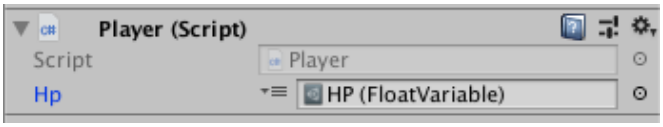
Fonte: Hipple (2).

Figura 9 – Inspetor da referência para uma Variável Ativa, tanto o valor constante quanto a referência podem ser utilizadas.



Fonte: o autor.

Figura 10 – Referência atribuída para a Variável Ativa.



Fonte: o autor.

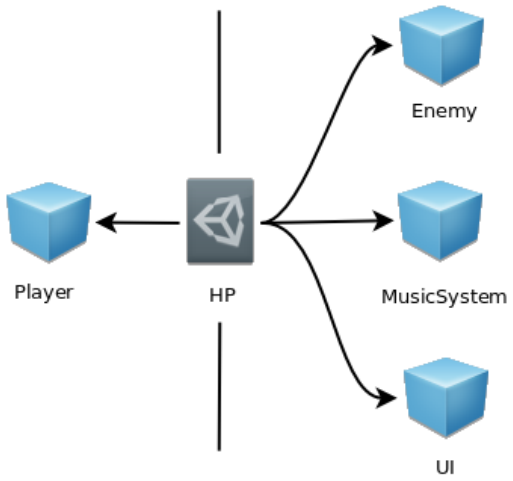
as Variáveis Ativas herdam direta ou indiretamente da classe *Variable<T>* fica fácil adicionar qualquer outra modificação caso necessário.

Por fim, o grande beneficio das Variáveis Ativas é que elas funcionam como links entre os vários componentes do jogo, e assim permitem desacoplar trechos de código que de outra forma estariam fortemente acoplados.

O desacoplamento de componentes do projeto por sua vez permite que estes mesmos componentes sejam reutilizados mais vezes pelo mesmo projeto e por outros.

A figura 11 mostra como os componentes: *Player*, *Enemy*, *UI* e *MusicSystem* interagem entre si por meio da Variável Ativa *HP*. Perceba que nenhum dos componentes à direita da figura depende diretamente do componenete *Player*. Ainda assim todos eles podem acessar livremente o ativo *HP* que é compartilhado por todos.

Figura 11 – Note que fazendo *HP* uma Variável Ativa permite desacoplar todos os sistemas e objetos que a utiliza. O ativo *HP* é compartilhado por todos os componentes na figura.



Fonte: o autor.

Além da modularidade e simplicidade de uso as Variáveis Ativas permitem que seu valor seja alterado no editor, mesmo durante runtime, o que deixa os testes muito mais simples.

Como exemplo, imagine o cenário onde sempre que a vida do jogador atingir o estado critico a música do jogo fica mais intensa. Utilizando Variáveis Ativas o sistema pode ser testado simplesmente alterando a variável responsável pela vida do *player*. Isso permite testar apenas o componente de música; nenhum outro componente precisa existir no nível.

4.1.1.1 Variáveis Locais

ScriptableObjects podem existir dentro cena da mesma forma que um *GameObject* e também podem ser criados em tempo de execução. Isso permite criar variáveis sobre demanda, desse modo fica muito prático estender a lógica para situações onde o número de variáveis pode mudar.

A função abaixo permite criar um *ScriptableObject* em tempo de execução:

```
1    ScriptableObject.CreateInstance<FloatVariable>();
```

Também é possível clonar uma Variável Ativa utilizando:

```
1    var clone = (FloatVariable)Object.Instantiate(variable);
```

Como exemplo, imagine o caso do modo cooperativo em um típico jogo *Beat'em up*[1], sempre que

[1] *Beat'em up* é um estilo de jogo criado nos tempos do acarde. Ele é caracterizado por um estilo de jogo de multiplos jogadores que podem se juntar a partida a qualquer momento e são todos do gênero de luta. Exemplo de jogos no estilo *Beat'em up*: Castle Crashers, Final Fight e Violent Storm

um novo jogador queira se juntar a partida é trivial instanciar um personagem, criar as Variáveis Ativas necessárias e atribui-las a outros compomenentes.

4.1.1.2 Outras Implementações

Em alguns casos é necessário que vários componentes do jogo modifiquem uma mesma variável simultaneamente; e.g. a vida do jogador pode ser incrementada por um power up de vida ao mesmo tempo que um inimigo a decrementa.

Graças ao modo como as Variávies Ativas foram estruturadas, é possível escrever diferentes verções apenas sobrescrevendo a propriedade *Value* da classe *Variable*. A figura 12 apresenta uma versão *threadsafe* da classe *FloatVariable*.

Figura 12 – *ClampedFloatVariable* é uma variante *threadsafe* e limitada da classe *FloatVariable*

```
1  [CreateAssetMenu]
2  public class ClampedFloatVariable : FloatVariable
3  {
4    private object lock = new object();
5
6    public override float Value
7    {
8      get
9      {
10       return Mathf.Clamp(base.Value, Min, Max);
11     }
12     set
13     {
14       lock (_lock)
15       {
16         base.Value = value;
17       }
18     }
19   }
20
21   public FloatReference Min = new FloatReference(0);
22   public FloatReference Max = new FloatReference(1);
23 }
```

Fonte: o autor

Os valores Max e Min, no código da figura 12, são colocados por conveniência para limitar o valor da variável, mantendo a responsabilidade de gerenciar seus limites na própria variável. A classe *ClampedFloatVariable* é adequada quando vários componentes alteram a variável de forma concorrente.

Observe que a implementação da figura 12, assim como qualquer outra, pode ser subsistida no campo de sua Variável Ativa base; assim é fácil utilizar a implementação certa para cada situação, sem que seja preciso alterar nenhuma linha de código.

4.1.2 Enumerações Extensíveis

Enumerações permitem criar conjuntos de constantes relacionadas e são muito versáteis e fáceis de gerenciar. Em um projeto de jogo, um típico exemplo de enumeração pode ser encontrado na figura 13.

Durante o desenvolvimento do jogo é muito provável que a enumeração da figura 13 mude. Alterar enumerações pode ser complicado no Unity 3D. Isso porque Unity 3D serializa as enumerações como valores inteiros.

Figura 13 – Maneira ingênua de declarar uma enumeração

```
1 public enum PowerUp
2 {
3   HealthBoost,
4   Invincibility,
5   SpeedBoost,
6   DamageBoost
7 }
```

Fonte: o autor

Esse esquema de serialização de enumerações faz com que adições sejam simples, mas desde que os novos elementos sempre sejam adicionados ao final da enumeração. Porém, remover elementos pode resultar na troca dos elementos que restaram. Isto é, ao remover o elemento *Invincibility*, *SpeedBoost* ficará com o mesmo valor numérico que antes era de *Invincibility* e assim subsequentemente. E como consequência da remoção todos os ativos que antes possuiam o campo de *Invincibility* agora terão o valor de *SpeedBoost*. Esse tipo de comportamento é causado porque a enumeração é implicitamente numerada.

Elementos de enumerações não podem ser deletadas, desde que seus valores sejam fixados. Sendo assim é preciso declarar os valores dos elementos da enumeração antes de remover qualquer um deles, como mostra a figura 14. Mas essa solução em si ainda não é perfeita, isso porque os ativos ainda estarão com o valor de *Invincibility* atribuido em seus campos.

Figura 14 – Elemento *Invincibility* removido de forma apropriada

```
1 public enum PowerUp
2 {
3   HealthBoost = 0,
4   SpeedBoost = 2,
5   DamageBoost = 3
6 }
```

Fonte: o autor

Assim, Fine (7) propõe um padrão chamado Enumerações Extensíveis do inglês *Extendable Enums*, que utiliza a classe *ScriptableObject* como base para criar enumerações. A figura 15 mostra a nova implementação da enumeração *PowerUp*. Note que essa implementação não declara explicitamente quais elementos ela possui.

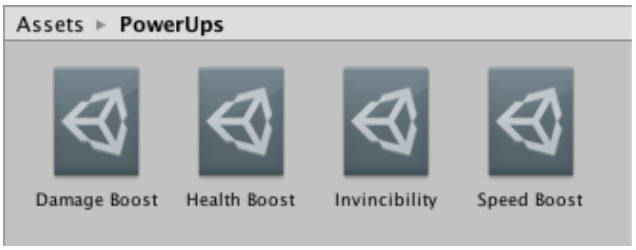
Para declarar um elemento de uma Enumeração Extensível é preciso criar um ativo da mesma. A figura 16 mostra os elementos da enumeração *PowerUp* da figura 15.

Figura 15 – Implementação de uma Enumeração utilizando *ScriptableObjects*

```
1 [CreateAssetMenu]
2 public class PowerUp : ScriptableObject
3 { }
```

Fonte: o autor

Figura 16 – Enumeração *PowerUp*



Fonte: o autor.

Ao utilizar Enumerações Extensíveis fica muito mais simples gerenciar os elementos das enumerações. Isso ocorre porque ao deletar um ativo, seu valor atribuido ao longo do projeto será igual a nulo. Também é possivel localizar os ativos que utilizam um elemento da enumeração através do *GUID* do elemento. É possivel ainda, substituir um elemento trocando seu *GUID* por um outro elemento da mesma enumeração.

Ainda utilizando o padrão de Enumerações Extensíveis, é possível transformar de forma simples uma enumeração em uma estrutura de dados mais complexa. Nos exemplos anteriores cada tipo de *PowerUp* pode conter um valor nominal para a grandeza do efeito que o mesmo tem e um tempo de duração, assim como mostra a figura 17.

Figura 17 – Extenção da Enumeração *PowerUp* para um classe de dados mais complexa

```
1 [CreateAssetMenu]
2 public class PowerUp : ScriptableObject
3 {
4   public float Value = 0; // effect magnitude
5   public float Duration = 1; // in seconds
6 }
```

Fonte: o autor

4.1.3 Singletons

Singletons são classes estáticas que sempre estão disponíveis para serem utilizadas de qualquer parte do projeto. Elas são especialmente úteis para gerenciar sistemas de áudio, gráficos, carregamento de objetos e outros.

A natureza estática dos *Singletons* criam códigos fortemente acoplados. O que por sua vez, gera uma grande dificuldade em testar o código, porque não é possivel alterar a implementação utilizada de um *Singleton* por uma classe de *Mockup*.

Sendo assim, Hipple (2) afirma que quanto menos *Singletons* existirem no projeto, menos problemas eles irão causar. Em concordância com Hipple; Fine (7) apresenta um modelo de Singleton baseado em *ScriptableObjects* para ser utilizado nas situações em que o mesmo é indispensável, e será discutido na próxima subsecção.

4.1.3.1 Baseado em *ScriptableObjects*

Apresentado originalmente por Fine (7), o modelo de *Singleton* da figura 18 permite que o estado do mesmo seja mantido após a execução de um *Hot Reload* (método pelo qual o Unity 3D atualiza seus scripts durante a execução do jogo).

Figura 18 – Base classe de um *ScriptableObject* Singleton

```
1  public abstract class RuntimeSingleton<T> : ScriptableObject
2    where T : RuntimeSingleton<T>
3  {
4    private static T _instance;
5    public static T Instance
6    {
7      get
8      {
9        if (!Application.isPlaying)
10       {
11         throw new InvalidOperationException("Singleton \"" +
12           typeof(T).Name + "\" is runtime only");
13       }
14
15       if (!_instance)
16       {
17         var array = Resources.FindObjectsOfTypeAll<T>();
18         if (array.Length > 0) _instance = array[0];
19       }
20       if (!_instance)
21       {
22         instance = CreateInstance<T>();
23         instance.hideFlags = HideFlags.DontSave;
24         DontDestroyOnLoad(_instance);
25       }
26       return _instance;
27     }
28   }
29 }
```

Fonte: Fine (7)

Todos os *Singletons* que herdam de *RuntimeSingleton* devem implementar seus métodos e atributos de forma não estática. Apenas campos públicos ou privados marcados com o atributo *SerializeField*, persistiram e poderão ser inspecionados durante a execução do jogo.

Assegurando que nenhum outro atributo ou propriedade do *Singleton*, salvo a propriedade *Instance*, sejam estáticas. A própria implementação do *Singleton* pode ser substituida uma classe de *Mockup*. Para tal, a classe de *Mockup* deve ser atribuida por meio de reflexão no campo *_instance* do *Singleton* em questão.

Adicionalmente, à instância do *Singleton* pode ser selecionada para inspeção no editor utilizando o seguinte snippet:

```
1 [UnityEditor.MenuItem("Singletons/My Custom Singleton")]
2 private static void Select()
3 {
4   UnityEditor.Selection.activeObject = Instance;
5 }
```

4.1.3.2 Baseado em MonoBehaviour

Baseado no *RuntimeSingleton* da figura 18 é possível derivar outros modelos de *Singleton* úteis para situações onde as classes em questão precisam receber eventos que normalmente apenas *MonoBehaviours* recebem, tais como: *Update*, *OnApplicationQuit*, *OnApplicationPause* e outros.

Esse tipo de *Singleton*, diferente do anterior que contem apenas dados, não é passivo e executa funções por conta própria.

Mesmo herdando de *MonoBehaviour* essa classe ainda consegue persistir seus dados depois de um *Hot Reload*. Isso porque o único atributo estático da classe é a referência para a instância da mesma. Após o *Hot Reload* a propriedade *Instance* é capaz de recuperar a referência para o objeto já instanciado.

Figura 19 – *Singleton* baseado em *MonoBehaviour*

```
1  public abstract class RuntimeSingletonBehaviour<T> : MonoBehaviour
2    where T : RuntimeSingletonBehaviour<T>
3  {
4    private static T _instance;
5    public static T Instance
6    {
7      get
8      {
9        if (!Application.isPlaying)
10         throw new InvalidOperationException("Singleton \"" +
11           typeof(T).FullName + "\" is runtime only");
12
13       if (!_instance)
14       {
15         var array = Resources.FindObjectsOfTypeAll<T>();
16         if (array.Length > 0) _instance = array[0];
17       }
18       if (!_instance)
19       {
20         CreateInstance();
21       }
22       return _instance;
23     }
24   }
25
26   protected static void CreateInstance()
27   {
28     if (_instance)
29     return;
30
31     var gameObject = new GameObject(typeof(T).FullName);
32     gameObject.hideFlags = HideFlags.DontSave;
33     instance = gameObject.AddComponent<T>();
34     DontDestroyOnLoad(_instance);
35   }
36 }
```

Fonte: o autor

A figura 19 contém a implementação do *Singleton* baseado em *MonoBehaviour*. Nessa implementação o método *CreateInstance* é oferecido como conveniência para ser utilizado em conjunto com o atributo *RuntimeInitializeOnLoadMethod* para criar uma instância de si mesma, assim que o jogo for iniciado.

Como exemplo de *Singleton*, pode-se apontar um sistema de *Auto-Save*, que salva o estado do jogo depois de um determinado tempo, ou sempre que o jogador sair do jogo, bruscamente ou não.

4.1.3.3 Variantes Configuráveis

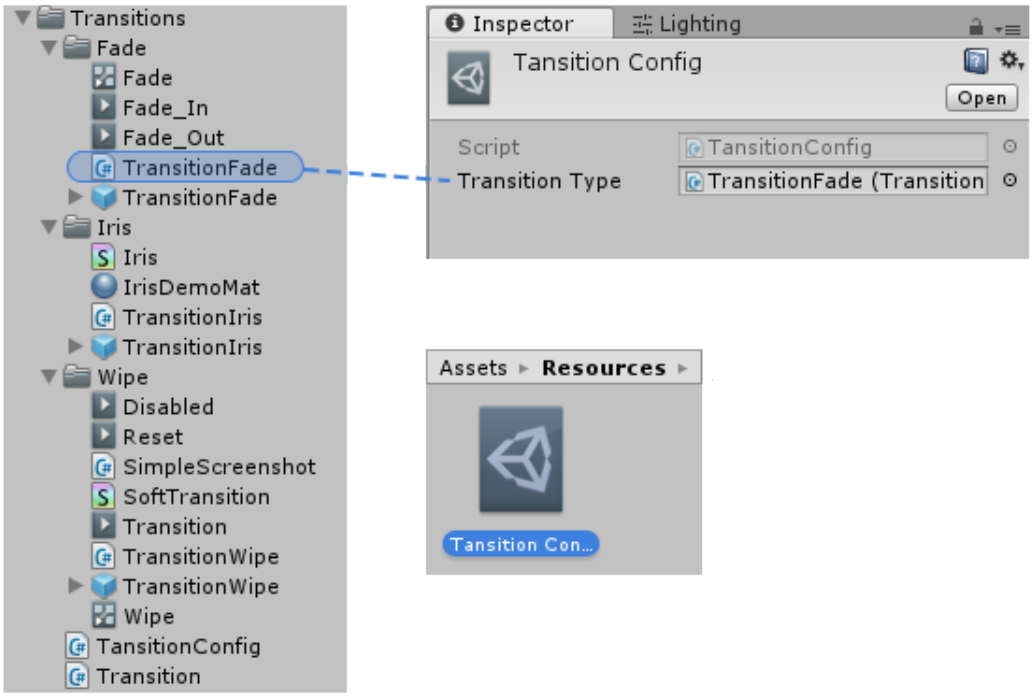
Essa variante de *Singleton* é caracterizada primariamente por criar um ativo dentro do projeto que pode ser customizado por designers. De maneira semelhante a uma Variável Ativa ou uma Enumeração Extensível, esse objeto pode ser referenciado indiretamente pelo código através de um campo serializavel ou carregado diretamente o objeto por meio da propriedade estática *Instance* ou utilizando a classe *Resources*.

O principal objetivo dessa classe é permitir a fácil parametrização e customização de valores iniciais para os *Singletons*.

A figura 20 contém a implementação da classe *RuntimeConfigurableSingleton*.

Como exemplo, imagine um sistema de transição de níveis com diferentes transições implementadas. Para saber qual transição o sistema deve utilizar por padrão, um *Singleton* Configurável é utilizado para selecionar a transição padrão. A figura 21 exemplifica a configuração de tal sistema.

Figura 21 – Transition Config é configurado para utilizar a transição Fade por padrão.



Fonte: o autor.

A implementação um de *Singleton* Configurável requer a implementação de dois métodos extras, descritos nos próximos dois parágrafos:

O primeiro método *private static void Init()* com o atributo *InitializeOnLoad*, utilizado apenas para criar o ativo configurável na pasta de Resources, que é feito chamando a função CreateDefaults.

Figura 20 – Singleton Configuravel

```
1  public abstract class RuntimeConfigurableSingleton<T> : ScriptableObject
2    where T : RuntimeConfigurableSingleton<T>
3  {
4    protected static void CreateDefaults() {
5  #if UNITY_EDITOR
6      var name = SplitCamelCase(typeof(T).Name);
7      var instance = Resources.Load<T>(name);
8      if (!instance) {
9        if (!UnityEditor.EditorUtility.DisplayDialog("Create␣Defaults",
10         "Create␣defaults␣for␣" + typeof(T).FullName + "␣inside␣Resources␣forlder?",
11         "Yes", "No"))
12         return;
13
14       var path = "Assets/Resources/" + name + ".asset";
15       Debug.Log("Creating␣asset␣" + path);
16
17       instance = CreateInstance<T>();
18       instance.name = name;
19       // Create default class and if in editor save it in the resources folder
20       UnityEditor.AssetDatabase.CreateAsset(instance, path);
21       UnityEditor.AssetDatabase.SaveAssets();
22     }
23 #endif
24   }
25
26   private static T _instance;
27   public static T Instance {
28     get {
29       var reload = false;
30       if (!_instance) {
31         reload = true;
32         instance = Resources.FindObjectsOfTypeAll<T>().FirstOrDefault();
33       }
34       var name = SplitCamelCase(typeof(T).Name);
35       if (!_instance) {
36         reload = true;
37         instance = Resources.Load<T>(name);
38       }
39       if (!_instance) throw new NullReferenceException();
40       if (reload) {
41         instance.OnInstanceLoad();
42       }
43       return _instance;
44     }
45   }
46
47   protected abstract void OnInstanceLoad();
48
49   private static string SplitCamelCase(string input) {
50     return System.Text.RegularExpressions.Regex.Replace(input, "([A-Z])", "␣$1"/*,
51     System.Text.RegularExpressions.RegexOptions.Compiled*/).Trim();
52   }
53 }
```

Fonte: o autor

O segundo e último método é o *OnInstanceLoad*, útil em situações específicas onde outros recursos também precisam ser recarregados.

4.1.4 Eventos (GameEvents)

GameEvents são um tipo especial de ativo que foi proposto por Hipple (2) como forma de reduzir o número de premissas que cada componente deve fazer antes de realizar uma chamada para um médoto de outro componente. A implementação basica de um *GameEvent* pode ser encontrada na figura 22.

A principal função de um *GameEvent* é servir de canal para notificar componentes dos vários eventos do jogo, sem que os receptores e emissores tenham conhecimento um do outro. *GameEvents* são baseados em *UnityEvent*, que é uma classe capaz de serializar eventos, e por isso resistem ao *Hot Reload*.

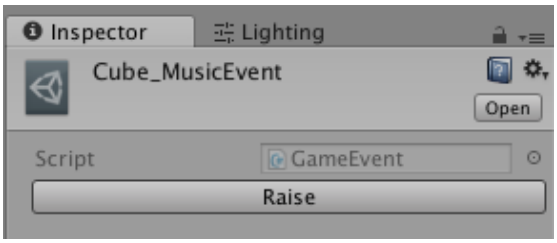
Figura 22 – Classe GameEvent

```
1  [Serializable]
2  public class GameEventCallback : UnityEvent<object>
3  { }
4
5  [CreateAssetMenu]
6  public class GameEvent : ScriptableObject
7  {
8    private readonly GameEventCallback runtimeEvents = new GameEventCallback();
9    private readonly UnityEvent runtimeEventsTypeless = new UnityEvent();
10
11   public void Raise(object argument = null) {
12     runtimeEvents.Invoke(argument);
13     runtimeEventsTypeless.Invoke();
14   }
15
16   public void RegisterListener(UnityAction<object> action) {
17     runtimeEvents.AddListener(action);
18   }
19
20   public void UnregisterListener(UnityAction<object> action) {
21     runtimeEvents.RemoveListener(action);
22   }
23
24   public void RegisterListener(UnityAction action) {
25     runtimeEventsTypeless.AddListener(action);
26   }
27
28   public void UnregisterListener(UnityAction action) {
29     runtimeEventsTypeless.RemoveListener(action);
30   }
31 }
```

Fonte: o autor

Utilizando um editor customizado, tanto designers quanto programadores podem acionar os eventos sempre que desejarem. Dessa foram torna-se incrivelmente prático testar as respostas dos componentes aos diferentes eventos. Veja na figura 23 o inspetor do *GameEvent*.

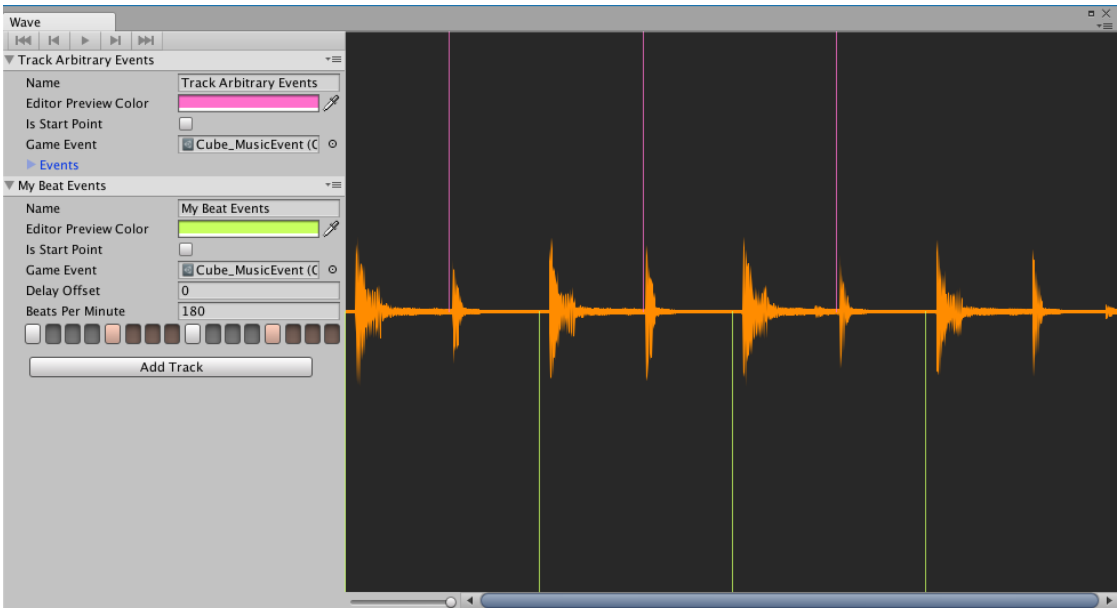
Figura 23 – Editor customizado de um GameEvent



Fonte: o autor.

Wave é um plugin de uso interno da Mental Lab[2] que é responsável por sincronizar eventos com a música; veja figura 24. *Wave* é utilizado para jogos rítmicos e para legendar trechos de diálogos. Cada ativo do *Wave* é responsavel por gerar eventos para uma música; esses eventos por sua vez são do tipo *GameEvent*, que podem ser criados e ouvidos de qualquer parte do projeto.

Figura 24 – Editor Wave, aciona de eventos sincronizados por áudio



Fonte: o autor.

4.1.5 Conjuntos (Sets)

Conjuntos de objetos são muito úteis para manter referências de outros objetos e gerenciá-los. Em um jogo de xadrez, por exemplo (veja figura 25), é preciso manter referências de todas as peças separadas de acordo com sua equipe, preto ou branco.

[2] Mental Lab é uma startup desenvolvedora de jogos recifence, <https://www.mentallab.com.br/>

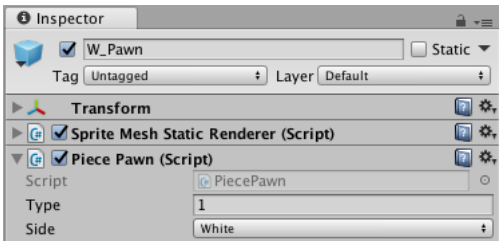
Figura 25 – Implementação trivial da classe ChessBoard

```
1 public class ChessBoard : MonoBehaviour
2 {
3   public List<Piece> white = new List<Piece>();
4   public List<Piece> black = new List<Piece>();
5 }
```

Fonte: o autor

Na implementação da figura 25, para identificar a equipe de uma dada peça é preciso pesquisar em qual das listas ela se encontra, ou de forma mais eficiente utilizar uma enumeração para marcar o time da peça. Um *GameObject* do peão desse jogo de xadrez pode ser visualizado na figura 26.

Figura 26 – Implementação ingênua do Peão em um jogo de Xadrez



Fonte: o autor.

Esse tipo de abordagem, apesar de ser pefeitamente funcional, não é robusta a mudanças do número de equipes, além de gerar um grande acoplamento de código. Hipple (2) apresenta o *RuntimeSet*, figura 27, como um substituto para gerenciar conjuntos de objetos.

Figura 27 – Classe RuntimeSet

```
1  public abstract class RuntimeSet<T> : ScriptableObject, IEnumerable<T>
2  {
3    [FormerlySerializedAs("Items")]
4    public List<T> items = new List<T>();
5
6    public void Add(T thing)
7    {
8      if (!items.Contains(thing))
9      items.Add(thing);
10   }
11
12   public void Remove(T thing)
13   {
14     if (items.Contains(thing))
15     items.Remove(thing);
16   }
17
18   public IEnumerator<T> GetEnumerator()
19   {
20     return items.GetEnumerator();
21   }
22
23   IEnumerator IEnumerable.GetEnumerator()
24   {
25     return GetEnumerator();
26   }
27 }
```

Fonte: Hipple (2)

Para adicionar ou remover equipes de meneira simplificada, cada equipe pode ser implementada como uma classe derivada de *RuntimeSet*. Assim cada equipe será identificada por um ativo único, que também contem a lista de todas as peças que fazem parte da equipe. Fazendo com que a peça de xadrez saiba diretamente qual equipe ela pertence; veja a figura 28.

Figura 28 – Implementação do Peão utilizando *RuntimeSet*

Fonte: o autor.

Os elementos do *RuntimeSet* dispensam classes gerenciadores como *ChessBoard* porque podem entrar ou sair de qualquer equipe por conta própria. O código da figura 29 apresenta um típico caso de elemento que pode entar e sair de um RuntimeSet.

Figura 29 – Snippet de como implementar um elemento autogerido de um *RuntimeSet*

```
1  public class AnRuntimeSetElement : MonoBehaviour
2  {
3    [SerializeField]
4    private RuntimeSetImpl m_Set;
5    public RuntimeSetImpl Set
6    {
7      get
8      {
9        return m_Set;
10     }
11     set
12     {
13       if (m_Set) m_Set.Remove(this);
14       m_Set = value;
15       if (m_Set && gameObject.activeInHierarchy) m_Set.Add(this);
16     }
17   }
18
19   private void OnEnable()
20   {
21     if (m_Set) m_Set.Add(this);
22   }
23
24   private void OnDisable()
25   {
26     if (m_Set) m_Set.Remove(this);
27   }
28 }
```

Fonte: o autor

4.1.6 Textos Localizados

Textos localizados são comumente criados mapeando uma chave para cada elemento de texto do projeto, então é responsabilidade do sistema de localização substituir a chave pelo texto na língua selecionada pelo usuário.

Esse tipo de implementação torna difícil de gerenciar as *strings* que estão sendo utilizadas. Um pequeno erro de digitação é o suficiente para deixar um texto sem conteúdo ou com vários pontos de interrogação.

Para contornar esse problema, as chaves podem ser substituídas por *ScriptableObjects*. A figura 30 mostra a implementação de uma chave para um texto localizado. Esse tipo objeto é uma especialização das Enumerações Extensíveis.

Figura 30 – Texto Localizado

```
1  [CreateAssetMenu]
2  public class LocalizedKey : ScriptableObject
3  {
4      public virtual string Key {
5          get {
6              return name; // The name of the scriptable object it self
7          }
8      }
9
10     public virtual string Text {
11         get {
12             try {
13                 return Localized.GetString(Key);
14             } catch {
15                 return DefaultText;
16             }
17         }
18     }
19
20     [Multiline]
21     [SerializeField]
22     private string m_DefaultText;
23     public virtual string DefaultText {
24         get {
25             return m_DefaultText;
26         }
27     }
28 }
```

Fonte: o autor

A classe *LocalizedKey* da figura 30, é utilizada para identificar as chaves dos textos localizados. Como conveniência todas as chaves podem ser exportadas para um arquivo *xml* ou *txt* para fácil utilização dos tradutores. Uma vez traduzidas elas serão lidas pelo sistema de localização e retornadas pela propriedade Text.

Uma das vantagens do *LocalizedKey* é que uma chave não pode ser utilizada se não for criada antes, evitando erros de digitação e habilitando a simples refatoração de chaves pré-existentes.

4.1.7 Texturas Dinâmicas e Outros Ativos

Por padrão o Unity 3D codifica as texturas e imagens importadas para um formato que pode ser utiliza facilmente pela GPU, reduzindo assim o tempo de carregamento dos níveis. Porém em certas situações é mais vantajoso guardar as imagens em um formato *lossless* com o objetivo de manter a qualidade da imagem e reduzir o tamanho do jogo exportado para plataforma alvo.

Felizmente Unity 3D oferece funções para importar texturas PNG. O processo de importação pode ser feito utilizando os métodos *Texture2D.LoadImage* passando o aquivo como uma *array* de bytes ou utilizando a classe WWW passando a URI do arquivo. Ambas as funções carregam a imagem em uma textura pré existente, isso permite que texturas sejam substistuidas facilmente.

O comportamento de sobrescrever texturas, descrito anteriormente, pode ser utilizado na localização de texturas, i.e. substituição de texturas baseado na linguá selecionada pelo jogador. No **Anexo A** encontra-se código da classe *LocalizedImage* responsável por carregar e substituir a textura que

ela controla; a textura controlada é criada no mesmo instante em que o ativo *LocalizedImage* for criado e não ocupa espaço na build final do jogo.

Unity 3D permite criar ativos de vários outros tipos como *Meshes*, Materiais e etc. Esses ativos podem ser utilizados como intermediários entre os componentes que os utilizam dos que os modificam, da mesma forma que *LocalizedImage* faz com a textura controlada.

4.1.8 Padrão para descrição de tipos complexos

Como já fora dito, Unity 3D não suporta a serialização de objetos nulos e de membros abstratos de classes que não herdam de *UnityEngine.Object*. Isso torna a serialização de tipos complexos difícil.

Pensando nisso o Unity 3D oferece a interface *ISerializationCallbackReceiver* que recebe automaticamente eventos de antes e depois que a serialização padrão do Unity tenha terminado. Esses eventos por sua vez podem ser utilizados para serializar partes do objeto que não seriam serializadas normalmente.

YSerializer é um subprojeto do Shell e consiste em um serializador YAML baseado no *SharpYaml*, modificado para seguir as mesmas regras de serialização do Unity, porém adicionado suporte para coleções como listas, dicionários e hashtables. *YSerializer* adiciona também suporte a nulos e a serialização de interfaces e campos abstratos não derivados de UnityEngine.Object, enquanto matém todas as referências para ativos gerenciados pelo Unity 3D.

Os detalhes da implementação do *YSerializer* são consideravelmente complexos e fogem do escopo dessa monografia. De maneira simplificada, *YSerializer* investiga todas as propriedades dentro do objeto a ser serializado utilizando reflexão e emitindo onde se aplica, os elementos YAML necessários para serializar o objeto.

YSerializer oferece uma API simples utilizada para deserializar e serializar objetos complexos facilmente. Essa interface é composta de dois métodos:

```
SerializerHelper.Serialize(Stream, object, List<UnityEngine.Object>)
```

e

```
SerializerHelper.Deserialize(TextReader, object, List<UnityEngine.Object>)
```

O *framework* Shell possui classes alternativas para *ScriptableObject* e *MonoBehaviour* que utilizam o *YSerializer* por padrão. A figura 31 mostra a implementação da variante *MonoBehaviour*.

Figura 31 – Alternativa para o *MonoBehaviour*, com capacidades avançadas de serialização

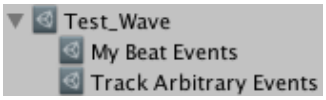
```
1  public abstract class SerializedMonoBehaviour : MonoBehaviour,
2    ISerializationCallbackReceiver
3  {
4    [YamlIgnore]
5    [SerializeField]
6    [HideInInspector]
7    private DataContainer m_SerializedInstanceData = new DataContainer();
8    public DataContainer serializedInstanceData { get { return m_SerializedInstanceData; } }
9
10   [YamlIgnore]
11   [SerializeField]
12   [HideInInspector]
13   private DataContainer m_SerializedPrefabData = new DataContainer();
14   public DataContainer serializedPrefabData { get { return m_SerializedPrefabData; } }
15
16   public void OnBeforeSerialize()
17   {
18     DefaultSerializedMethods.OnBeforeSerialize(this,
19       ref m_SerializedInstanceData, ref m_SerializedPrefabData);
20   }
21
22   public void OnAfterDeserialize()
23   {
24     DefaultSerializedMethods.OnAfterDeserialize(this,
25       ref m_SerializedInstanceData, ref m_SerializedPrefabData);
26   }
27 }
```

Fonte: o autor

De forma alternativa, objetos complexos podem ser descritos utilizando o aninhamento de *ScriptableObjects*. Onde um ativo pai é criado, responsável pelo gerenciamento dos demais ativos filhos. Mas, diferente do *YSerializer* é relativamente mais complexo e definitivamente mais passível de erros.

Os ativos filhos são aninhados com o ativo pai utilizando o método *AddObjectToAsset* da classe *AssetDatabase* e normalmente tem o seu *hideFlags* atribuído para *HideInHierarchy*. O Wave, por exemplo, utiliza esse padrão para decrever cada trilha de eventos, veja a figura 32.

Figura 32 – Tipico objeto *Wave*, modificado para exibir as trilhas que ele gerencia



Fonte: o autor.

Como nota final *ScriptableObjects* criados por componentes durante o *runtime* vivem na mesma cena do componente que os criou e são destruidos quando a cena é removida.

4.2 Inspetor de propriedades avançado

A qualidade do *workflow* aumenta muito utilizando os padrões descritos nas secções anteriores. Ainda assim é preciso utilizar padrões como os descritos por S.O.L.I.D. para estruturar corretamente o código do jogo.

Um dos padrões mais úteis do S.O.L.I.D. envolve a utilização de sistemas de DI. Porém sistemas de DI giram em torno dos programadores. Isso é ruim porque a maior parte do tempo da produção do jogo é concentrada na jogabilidade, arte, música e enredo. Esse tipo de trabalho é feito por game designers, artistas e escritores. E quanto menos dependentes de um programador melhor o trabalho deles irá se desenvolver.

Hipple (2), em sua palestra apresenta o brevemente o conceito do editor como um framework de DI. Essa ideia é particularmente interessante porque permite um maior controle nas mãos dos designers, além de utilizar um *workflow* familiar a qualquer um que usa o Unity 3D. A principal vantagem em pensar no editor como um sistema de DI é remover a necessidade de programadores terem que intervir constantemente para modificar o comportamento do componente.

A serialização padrão do Unity não suporta serialização de nulos, interfaces e certos tipos abstratos. Assim, o *framework* Shell oferece um esquema de serialização mais robusto em conjunto com um inspetor melhorado para situações onde o uso de interfaces é preferível.

O *YSerializer* como já foi mencionado é o núcleo desse sistema e oferece além da serialização duas classes: *SerializedMonobehaviour* e *SerializedScriptableObject* para substir as classes *Monobehaviour* e *ScriptableObject* respectivamente. Ambas possuem seus respectivos editores customizados capazes de exibir e editar interfaces, campos abstratos e nulos.

Semelhante às classes padrão do Unity 3D, o *Yserialzier* oferece alternativas para as classes *SerializedObject* e *SerializedProperty* de mesmo nome, dentro do *namespace Yserialized.Editor*. Elas são possuem as mesmas funções e campos das classes originais do *Unity 3D* e foram construídas dessa forma para facilitar a reimplementação de editores já existentes.

A classe abaixo pode ser serializada e inspesionda como mostra a figura 34:

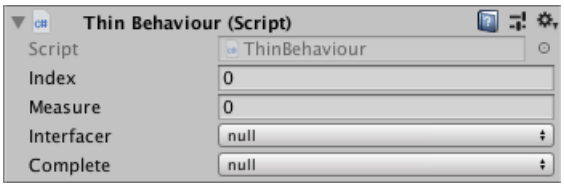
Figura 33 – Classe *ThinBehaviour*

```
1 public class ThinBehaviour : SerializedMonoBehaviour
2 {
3   public int index;
4   public float measure;
5   public IEmptyInterface interfacer; // interface field
6   public IncompleteClass complete; // abstract field
7 }
```

Fonte: o autor.

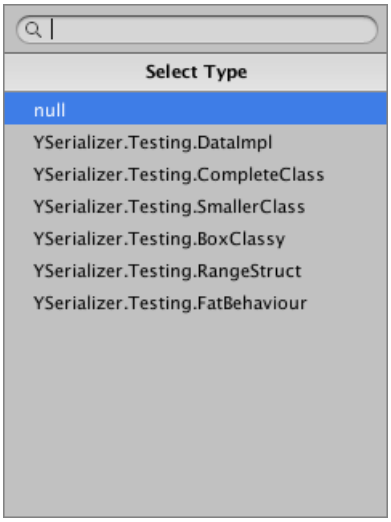
Veja a classe da figura 33; tradicionalmente os atributos *interfacer* e *complete* não seriam exibidos no editor, mas ao herdar da classe *SerializedMonoBehaviour* a serialização e inspetor avançados são utilizados. Na figura 34 perceba que ambos os campos *interfacer* e *complete* são exibidos com o valor igual a null; para instanciar um novo valor basta clicar no botão de *dropdown* para o *popup* de seleção de tipo aparecer, figura 35. No *popup* é possível escolher qualquer tipo herdado da interface ou abstração.

Figura 34 – Inspetor avançado da classe *ThinBehaviour*



Fonte: o autor.

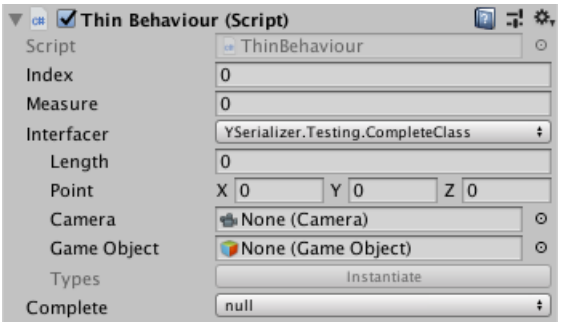
Figura 35 – Diferentes tipos podem ser selecionados utilizando esse dropdown; note que apenas os tipos que se aplicam são listados aqui



Fonte: o autor.

Uma vez selecionado um tipo todos seus campos serializáveis serão exibidos logo abaixo, assim como mostra a figura 36.

Figura 36 – Uma vez selecionado o tipo do objeto seus atributos são exibidos logo abaixo do *dropdown*



Fonte: o autor.

Utilizando as classes fornecidas por *YSerializer* é possível utilizar o conceito do editor como

sistema DI de Hipple (2) sem restrições. Agora também não existe nenhum outro empecilho para utilizar todos os padrões descritos por S.O.L.I.D. no Unity 3D.

4.3 Outras Engines

4.3.1 Unreal Engine 4

O Unreal Engine oferece a classe *UDataAsset* que desempenha o mesmo papel que *ScriptableObjects* no Unity 3D. Isso faz dela a candidata perfeita para implementar os vários padrões de Shell.

A figura 37 mostra a implementação da *FloatVariable*, no caso chamada de *FloatAsset*. Essa classe contem todas as definições necessárias para que possa ser integrada perfeitamente no sistema de *Blueprints* do Unreal Engine.

Figura 37 – *UFloatAsset* equivalente da *FloatVariable* destinada para o Unreal Engine

```
1  // UFloatAsset.h
2  UCLASS(BlueprintType)
3  class VARASSET_API UFloatAsset : public UDataAsset
4  {
5    GENERATED_BODY()
6
7    public:
8    UPROPERTY(EditAnywhere, BlueprintReadWrite)
9    float Value = 0.0;
10
11 };
12
13 //  UfloatAsset.cpp
14 #include "FloatAsset.h"
```

Fonte: o autor

4.3.2 Godot 3

Godot possui classe *Resource* que é destinada à criação de objetos dados dissociados de qualquer outra entidade, da mesma forma que *ScriptableObject* no Unity.

A figura 38 mostra a implementação de uma *FloatVariable* em *GDScript*.

Figura 38 – GDScript de uma FloatVariable

```
1  extends Resource
2
3  export(float) var value = 0.0
```

Fonte: o autor

Apesar da simples implementação, Godot não oferece um *workflow* simplificado para a criação desse tipo de ativo. Utilizando *GDScript* não permite que tipos customizados de *Resource* sejam exportados (exibidos no inspetor) o que torna difícil saber qual tipo *Resource* um dado campo espera.

5 Conclusões

É possível concluir que os padrões: *Variável Ativa*, *Enumerações Extensíveis*, *RuntimeSets*, *GameEvents* que foram apresentados nessa monografia são estruturas de dados independentes, i.e. não precisam de nenhum outro objeto para que existam. Por isso são excelentes para conectar os vários componentes e *Prefabs* de um jogo feito no Unity 3D.

Os padrões simplificam grande parte da complexidade associada ao desenvolver projetos de jogos sistêmicos, ou seja, jogos com componentes muito interligados. Uma vez que a interligação entre esses componentes são feitas por ativos como: *Variáveis Ativas*, *RuntimeSets* e *GameEvents*. Dessa forma, fazendo desses componenetes independentes uns dos outros.

O *framework* Shell compila várias classes em forma de componentes *on-the-shelf* que podem ser facilmente utilizados, editados ou estendidos. Shell foi empregado com sucesso no desenvolvimento do jogo *Castle of Awa*[1] criado pelo estúdio de jogos *Mental Lab*. Graças ao *framework* Shell, foi possível acelerar o tempo de desenvolvimento do jogo em várias semanas. Como consequência, Shell foi adotado em todos os outros projetos da *Mental Lab*.

[1] <https://www.mentallab.com.br/CoA/>

6 Trabalhos Futuros

Como sugestões de trabalhos futuros podem ser listados:

YSerializer, subprojeto do Shell, se encontra em fase alpha, portanto seu uso não é aconselhável para projetos importantes. Para levá-lo a condição de beta será preciso corrigir vários bugs; mais casos de testes precisam ser adicionados e o *workflow* com *Prefabs* precisa ser implementado. O novo sistema de *Prefabs* que será liberado na versão 2018.3 do Unity 3D também precisa ser levado em consideração antes de qualquer futuro desenvolvimento.

É difícil entender o impacto dos padrões e técnicas mostradas nesse documento, mesmo para alguém experiente na área de desenvolvimento de jogos. Essa dificuldade se dá em grande parte pelo fato deste trabalho não ilustrar com clareza exemplos. Essa monografia precisa de mais exemplos e demonstrações para que se torne fácil compreender como utilizar e qual a importância do Shell.

Referências

1 SAGMILLER, D. *S.O.L.I.D. Unity*. 2017. Unite Austin. Disponível em: <http://y2u.be/eIf3-aDTOOA>.

2 HIPPLE, R. *Game Architecture with Scriptable Objects*. 2017. Unite Austin. Disponível em: <http://y2u.be/raQ3iHhE_Kk>.

3 ELDER, C. *TOY BOXING in Unity*. 2018. Disponível em: <http://y2u.be/92Oo17K3Eoo>.

4 KINNEN, T. *Data-Driven and Component-Based Game Entities*. 2012.

5 WALLENTIN, O. *Component-Based Entity Systems, Modular Object Construction and High Performance Gameplay*. Tese (Doutorado) — Uppsala Universitet, jun. 2014.

6 SHUMAKER, S. *Techniques and Strategies for Data-driven design in Game Development*. 2012. Disponível em: <http://web.eecs.umich.edu/~soar/Classes/494/talks/Schumaker.pdf>.

7 FINE, R. *Overthrowing the MonoBehaviour Tyranny in a Glorious Scriptable Object Revolution*. 2016. Unite. Disponível em: <https://youtu.be/6vmRwLYWNRo>.

8 ANTE, J. *Writing High Performance C# Scripts*. 2017. Disponível em: <https://youtu.be/tGmnZdY5Y-E>.

ANEXO A – Textura Localizada

```
1  public class LocalizedImage : ScriptableObject, ISerializationCallbackReceiver
2  {
3    [SerializeField]
4    private Texture2D m_Target;
5
6    [SerializeField]
7    private List<LocalizedImagePair> m_Locate;
8
9    [Serializable]
10   public class LocalizedImagePair
11   {
12     public string Language;
13     public string Image;
14   }
15
16   private void OnLanguageChange(string language)
17   {
18     var locate = m_Locate.FirstOrDefault(x => x.Language == language);
19     if (locate == null) return;
20     var uri = new Uri(Application.streamingAssetsPath + "//" + locate.Image);
21     using (var www = new WWW(uri.AbsoluteUri))
22     {
23       while (!www.isDone) { } // Wait to load image
24       www.LoadImageIntoTexture(m_Target);
25     }
26   }
27
28   [RuntimeInitializeOnLoadMethod]
29   private static void Init()
30   {
31     Resources.LoadAll<LocalizedImage>("Localized");
32   }
33
34   private void OnEnable()
35   {
36     OnLanguageChange(Locate.CurrentLanguage);
37     Locate.onLanguageChange += OnLanguageChange;
38   }
39
40   private void OnDestroy()
41   {
42     Locate.onLanguageChange -= OnLanguageChange;
43   }
44
45   public void OnBeforeSerialize()
46   {
47     if (Application.isPlaying)
```

```
48      return;
49
50      m_Target.name = name; // Match names
51      m_Target.Resize(2, 2); // Reduce build size
52      m_Target.Apply();
53    }
54
55    public void OnAfterDeserialize()
56    { }
57
58  #if UNITY_EDITOR
59    [UnityEditor.MenuItem("Assets/Create/Localized Image With Proxy", false, 0)]
60    private static void Create()
61    {
62      // Create both the Asset and a proxy Texture
63      var loc = ScriptableObjectUtility.CreateAsset<LocalizedImage>();
64      var texture = new Texture2D(2, 2, TextureFormat.ARGB32, true, false);
65      texture.name = loc.name;
66      UnityEditor.AssetDatabase.AddObjectToAsset(texture, UnityEditor.AssetDatabase.GetAssetPat
67      loc.m_Target = texture;
68      UnityEditor.AssetDatabase.SaveAssets();
69      UnityEditor.AssetDatabase.Refresh();
70    }
71  #endif
72  }
```