Desenvolvimento de um Controlador de Personagem para Jogos do Gênero RPG Através do Motor de Desenvolvimento Unity e Abstração Orientada a Objetos

Lucas Alberto de Araujo[1], André Marcos Silva[1]

[1]Centro Universitário Campo Limpo Paulista (UNIFACCAMP)
Jardim América – CEP 13231-230, Campo Limpo Paulista – SP – Brasil

lucas.alberto0609@gmail.com, andre@faccamp.br

Abstract. *This article proposes the development of a controller using the object-oriented programming paradigm and a finite state machine to manage the behavior of a character in a role-playing game, through the CShap programming language used in the Unity Development Engine, with the proposal of analyzing the practicality, advantages, and disadvantages of the support resources and methods used.*

Resumo. *Este artigo propõe o desenvolvimento de um controlador utilizando o paradigma de programação orienta a objeto e uma máquina de estados finitos para gerenciar os comportamentos de um personagem de um jogo do gênero RPG, através da linguagem de programação CShap utilizada no Motor de Desenvolvimento Unity, com a proposta de analisar a praticidade, as vantagens e as desvantagens dos recursos de apoio e métodos utilizados.*

1. Introdução

Devido à grande variedade de ações que se pode realizar ao controlar um protagonista de um jogo de RPG (Role - Playing Game), a organização e implementação do código responsável por cada ação torna-se um trabalho árduo e inviável se não for realizado de forma correta, pois diversos problemas serão oriundos de um sistema que não execute de forma assíncrona um comportamento, tendo em vista que os comportamentos executam funções distintas.

Nesse cenário, é de suma importância escolher a arquitetura correta ao implementar um controlador para que haja um fluxo dinâmico entre os estados sem a interferência indesejada de um estado em outro, já que esta tende a ser uma tarefa geralmente complexa e que se agrava à medida que aumenta a quantidade de elementos participantes e interações no jogo.

2. Objetivos e Metodologia

O objetivo do trabalho é desenvolver um modelo simples de uma máquina de estados finitos para o gerenciamento de ações, ao qual seja possível adicionar, remover e modificar um comportamento para um personagem, visando a praticidade no desenvolvimento de novas ações, e a modificação de forma independente nos estados existentes.

Para a execução do projeto foi criado um personagem do zero, assim como o cenário e afins, utilizando as ferramentas disponíveis pelo motor de desenvolvimento, com objetivo exclusivo de desenvolver a movimentação do personagem.

Para a execução do projeto foi criado um personagem do zero, através do *software* Blender foi realizada todas as etapas do processo de modelagem, sendo elas blocagem, escultura, retopologia e o recorte da malha para aplicação da textura. O Adobe Photoshop foi outro *software* utilizado para auxiliar este processo, na criação da arte conceitual e na criação e edição das texturas dos objetos presentes em cena. O cenário e afins foi construído através das ferramentas disponíveis pelo próprio motor de desenvolvimento, com objetivo exclusivo de desenvolver a movimentação do personagem.

3. Atividade Prática

Para o desenvolvimento do protótipo, foi utilizada a linguagem de programação CSharp (Microsoft, 2022) em sua versão 8 com o tempo de execução .NET 4.x, e outras ferramentas de apoio:

- Unity 2020.3.39f1 LTS: Software utilizado na criação de componentes base para jogo digital;
- Visual Studio Code: Editor de texto utilizado para a criar e gerenciar o código da aplicação;
- GitHub: Utilizado para o gerenciamento do código fonte tendo como base o sistema de versão Git;
- UnityEngine: Biblioteca oferecida pela Unity para que seja possível manipular componentes do Software através do código;
- Draw.io: Software utilizado na criação de diagramas UML;
- Blender: Software utilizado na modelagem do personagem;
- Adobe Photoshop CC2019: Editor de imagens utilizado na criação de textura dos modelos 3D;

O primeiro passo para criação de uma máquina de estado finita, foi definir a arquitetura do modelo do estado que será utilizado como base para criação dos estados do personagem, para isso pode ser utilizado uma interface, alguns métodos se fazem necessário para que haja uma transição entre os estados de forma correta sendo eles executados ao iniciar, durante execução daquele estado e no momento da transição.

O passo seguinte foi criação da uma classe que faça a passagem entre as ações do personagem, para que isso ocorra de forma correta se faz necessário a finalização do estado vigente, para assim criar e iniciar um novo comportamento de forma segura.

Foi definido como método para gestão de comportamento o reconhecimento de *inputs* de *mouse* e teclado, para que através desses *inputs* o jogador possa realizar a ação desejada.

A Figura 1 apresenta um diagrama com as com possíveis transições entre os estados.

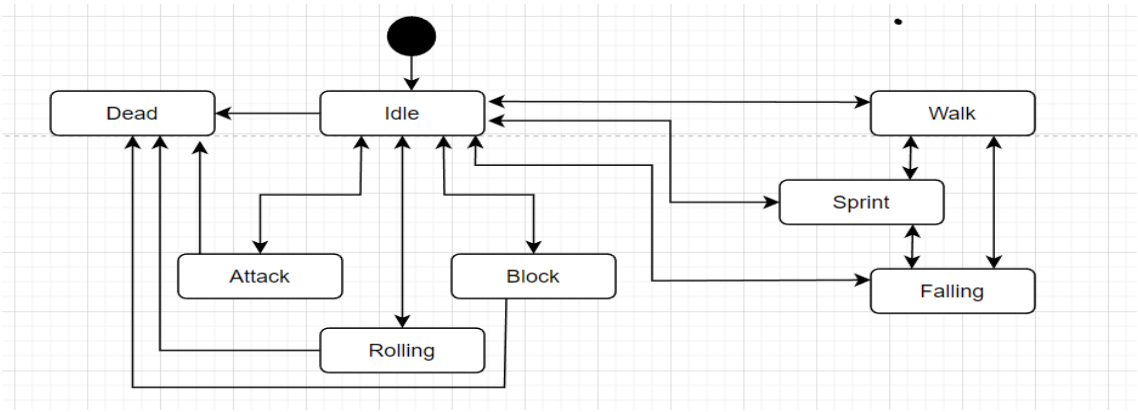

Figura 1. Diagrama com as transições de estados.

A Figura 2 apresenta um diagrama com a estrutura do método responsável pela transição entre estados.

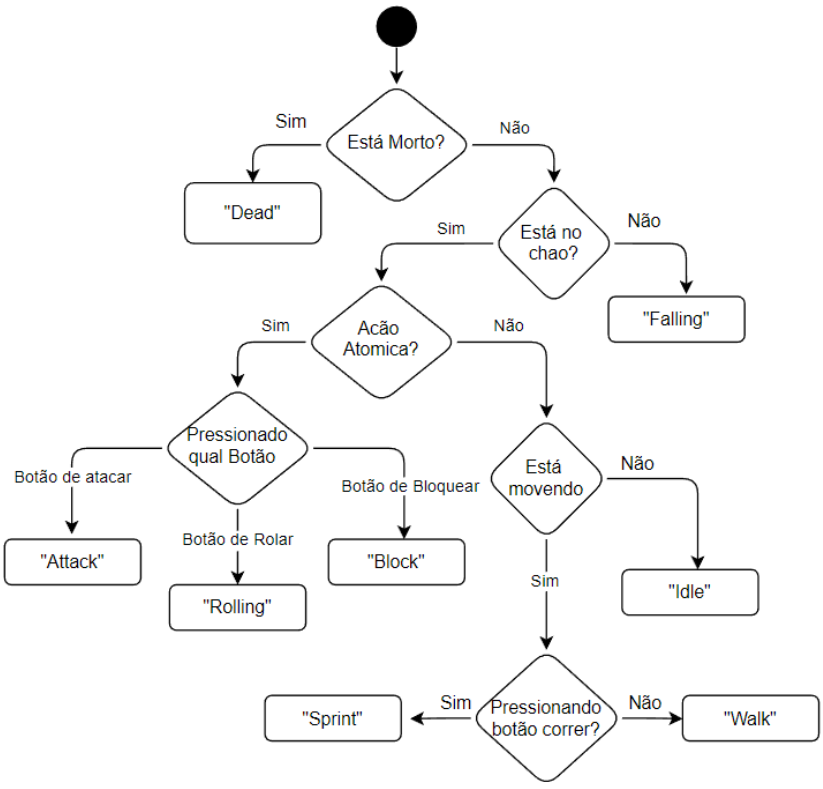

Figura 2. diagrama do método de transição.

A Figura 3 apresenta o método responsável por fazer a chamada de transição de estados com base em informações predefinidas e reconhecimento de *inputs*.

```
61  public void HandlerStates()
62  {
63      if(!isDead)
64      {
65          if(inground())
66          {
67              if(!inAction)
68              {
69                  ActionsInputs();
70                  if(inAction) return;
71
72                  if(inMove)
73                  {
74                      if(_InputHandler.SprintInput())
75                          if(CurrentState != States.Sprint)
76                              stateMachine.ChangeState(SprintState.GetSprintState(this));
77                      else
78                          if(CurrentState != States.Walk)
79                              stateMachine.ChangeState(WalkState.GetWalkState(this));
80                  }
81                  else
82                      if(CurrentState != States.Idle) stateMachine.ChangeState(IdleState.GetIdleState(this));
83              }
84          }
85          else
86              if(CurrentState != States.Falling) stateMachine.ChangeState(FallingState.GetFallingState(this));
87      }
88      else
89          if(CurrentState != States.Dead) stateMachine.ChangeState(DeadState.GetDeadState(this));
90  }
```

Figura 3. Método responsável pela transição de estados

Basicamente os estados, como por exemplo, "Walk", são definidos por uma *interface* a qual é utilizada como base para criação destes estados e uma classe de controle utilizada na transição entres eles.

3.1. Resultados obtidos

A implementação do controlador utilizando uma máquina de estados finita mostrou-se muito eficiente, pois o ganho de produtividade durante o desenvolvimento do projeto foi notório, tendo em vista a facilidade na edição do código de cada estado, assim como a organização de atributos e métodos.

A Figura 4 (a) apresenta o estado “Walk”, quando o controlador de estados reconhece a chamada para movimentação através do controlador de *inputs*. Figura 4 (b) apresenta o personagem em posição “Idle”, quando não há chamada para ação. Figura 4 (c) apresenta o personagem em posição “Block”, uma ação atômica que quando em execução não pode ser interrompida pelas outras com exceção das ações “Falling” e “Dead”.

Figura 4. (a) Personagem em estado "Walk", (b) Personagem em estado "Idle" e (c) Personagem em estado "Block".

3.2. Problemas enfrentados

Uma das dificuldades enfrentadas foi a codificação do padrão de projeto *singleton,* aonde através dos conceitos apresentados por (Gamma, E.) pode ser superada com êxito, para que a instancia de cada estado não fosse criada toda vez que houvesse a mudança entre os estados.

Houve também uma dificuldade ao implementar as classes dos estados aos quais apresentaram semelhança devido a herança de uma classe abstrata, pois os mesmos possuem comportamentos idênticos, este problema pode ser solucionado utilizando conceitos de polimorfismo e sobrescrita de métodos (Leite, 2007)

4. Conclusão

O trabalho possibilitou uma maior compreensão dos paradigmas orientados a objetos, não só no âmbito pratico, mas também na parte de arquitetura, conforme aprendizados e conceitos aprendidos ao decorrer do período de desenvolvimento.

Referência Bibliográfica

Documentação CShap (2022); "Documentação da linguagem de programação CShap". Disponível em https://docs.microsoft.com/pt-pt/dotnet/csharp, acessado em agosto de 2022.

Gamma, E. (2007). "Padrões de projeto: soluções reutilizáveis de software orientado a objetos". Ed. Bookman, ed1, Porto Alegre, 2007.

Leite, T. (2007). "Orientação a Objetos: Aprenda seus conceitos e suas aplicabilidade de forma efetiva". Ed Casa do Código, ed 1, 2016

Visual Studio Code (2022). “Site oficial da plataforma de edição de textos e código”. Disponível em https://code.visualstudio.com. Acessado em agosto de 2022.

Unity Documentation (2022); “Documentação oficial da Unity Technologies”. Disponível em https://docs.unity3d.com/ScriptReference/, acessado em março de 2022.

Unity Learn (2022); “Plataforma de ensino oficial da Unity Technologies”. Disponível em https://learn.unity.com/, acessado em março de 2022.