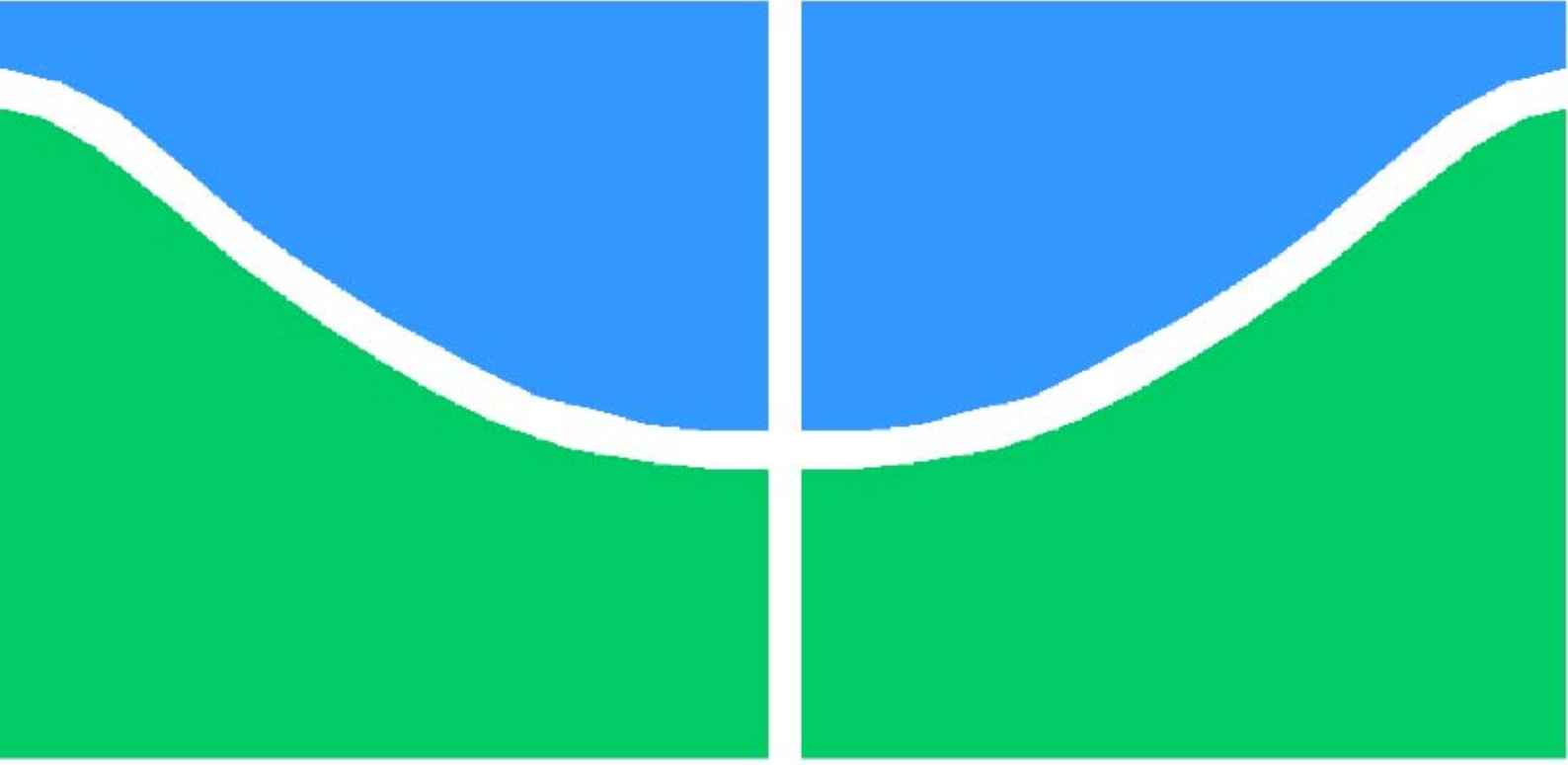

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de *Software*

NTeraction: um *Storytelling Engine* para *Unity3D*

Autor: Vítor Makoto Matayoshi de Moraes
Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF
2014

Vítor Makoto Matayoshi de Moraes

NTeraction: um *Storytelling Engine* para *Unity3D*

Monografia submetida ao curso de graduação em Engenharia de *Software* da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de *Software*.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Coorientador: Prof. Msc. André Luiz Peron Martins Lanna

Brasília, DF

2014

Vítor Makoto Matayoshi de Moraes

NTeraction: um *Storytelling Engine* para *Unity3D*

Monografia submetida ao curso de graduação em Engenharia de *Software* da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de *Software*.

Trabalho aprovado. Brasília, DF, 27 de junho de 2014:

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. Fabricio Ataides Braz
Convidado 1

Prof. Dra. Carla Silva Rocha Aguiar
Convidado 2

Brasília, DF
2014

Resumo

Diversos jogos digitais da atualidade tem sucesso baseado fortemente nas estórias contadas. Isto faz com que a qualidade exigida pelo sistema que capacita o jogo com a funcionalidade de se contar uma estória, conhecido como motor de *storytelling*, seja tão importante quanto a qualidade dos dois grandes pilares técnicos do desenvolvimento de jogos: gráfico e mecânica. Nesse contexto, um dos grandes desafios é gerar, organizar e, principalmente, testar os conteúdos não lineares que fazem parte da estória contada por um jogo. Tendo isso em vista, o presente trabalho apresenta uma proposta de um motor de *storytelling* genérico para o motor de jogos *Unity3D*. O motor proposto é adaptável às diversas necessidades de diferentes jogos. Além disso, uma importante característica do motor proposto é que a geração e o teste do conteúdo pelo *game designer* seja possível com o mínimo de intervenção de programadores.

Palavras-chaves: Motor de *storytelling*, desenvolvimento de jogos, estórias, ficção interativa, jogos.

Abstract

Many digital games today had their success heavily based on their stories, thus making the quality required by the system that enables the game with the functionality of telling a story, known as storytelling engine, as important as the quality of the two technical pillars of game development: graphics and mechanics. In this context, a major challenge is to generate, organize , and, above all, test the non-linear content that are part of the story told by a game. Taking this into account, the present work presents the proposal of a generic storytelling engine integrated into Unity3D game engine capable of being adapted for the different functionalities of any game. In addition, an important characteristic of the proposed engine is that the generation and testing of the game desinger's content should be possible with minimal intervention of programmers.

Key-words: Storytelling engine, shaders, game development, stories, interactive fiction, games.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

IDE	*Integrated development environment*
SCM	*Software configuration management*
XML	*Extensible markup language*
UML	*Unified modeling language*
NPC	*Non-player character*
ECS	*Entity-component-system*

Sumário

1 Introdução

A complexidade e a característica multidisciplinar de se fazer jogos são fatores que influenciam a necessidade de se buscar e desenvolver soluções para diversos desafios que surgem ao longo do desenvolvimento. Dentre vários outros, alguns destes desafios estão atrelados às estórias dos jogos, que vão muito além do que apenas a capacidade de um jogo de contar sua estória quando se considera o processo de desenvolvimento de jogos como um todo. Tais desafios são relacionados à criação, gerenciamento e teste dos conteúdos da estória, e estes são os problemas que o produto do presente trabalho busca solucionar.

1.1 Contexto

Um dos grandes desafios no processo de desenvolvimento de jogos digitais é a criação, organização e teste dos elementos (tais como diálogos, *cutscenes*, eventos, *quests*, etc) que compõem a estória, que é a característica de um jogo conhecida como *storytelling*.

1.2 Problema/Relevância

Segundo (M.BUCKLAND, 2007), a qualidade da *storytelling* é um importante aspecto dos jogos digitais modernos e que muitas vezes pode ser de suma importância para o sucesso ou fracasso deles. De acordo com (L.SHELDON, 2004), em jogos que são fortemente dependentes dos elementos da estória, há a necessidade de que a qualidade da *storytelling* seja, no mínimo, igual à qualidade dos gráficos, mecânicas e sons de um jogo.

Nesse contexto, é perceptível a importância de prover o suporte necessário que permita aos escritores trabalharem na criação das estórias interativas. Porém, há diversos problemas desafiadores a serem considerados no momento de se desenvolver esse suporte, que serão explicitados a seguir.

Atualmente, muitas das soluções para suporte na geração e manutenção dos dados de *storytelling* são feitas através de ferramentas que requerem conhecimento de programação. Segundo (M.CARBONARO, 2005) isso é problemático quando se considera que as pessoas que assumem o papel de escritores muitas vezes não satisfazem esse requisito (possuir conhecimento de programação). Fato semelhante ocorre quando é preciso alocar um programador para assumir o papel de escritor, mas nesse caso a dificuldade está relacionada às habilidades de escrita e conhecimentos de criação de estórias para se desempenhar essa função de forma eficaz.

Consequentemente, a realização da *storytelling* em jogos é feita por pessoas quali-

ficadas para escrever uma boa estória de maneira fortemente dependente de programadores, o que além de ser economicamente caro por consumir recursos humanos e tempo, cria oportunidades para possíveis erros de comunicação, erros de programação, e talvez até falta de dinamismo e controle sobre o processo no momento de se realizar manutenções ou mudanças na estória (M.MCLAUGHLIN; M.KATCHABAW, 2007).

Portanto, resumidamente podemos definir que o problema que esse trabalho busca enfrentar é:

Desenvolver uma ferramenta que permita a criação de estórias para jogos sem a intervenção e auxílio constante de programadores.

Para resolver esse problema temos que lidar com outros tipos de problemas do domínio doe jogos. Dentre esses, destaca-se um problema evidenciado por (C.OWEN, 2008) sobre a dificuldade de realizar o gerenciamento de acontecimentos não lineares da estória de uma jogo, pois à medida que o desenvolvimento do jogo vai crescendo, a estória passa a ser mais complexa.

1.3 Motivação

De acordo com (M.MCLAUGHLIN; M.KATCHABAW, 2007), é evidente a necessidade da criação de ferramentas que permitam o gerenciamento como um todo da estória com o mínimo de programação ou auxílio de programadores. Assim, tais ferramentas devem possibilitar ao escritor descrever a estória com o uso de linguagem natural, gráficos ou outras formas com níveis mais altos de abstração, enquanto, de maneira automatizada, a estória é integrada ao jogo com pouca ou nenhuma interferência humana.

Tendo isso em vista e aliado ao fato de que o *Unity3D*[1] – um motor de jogo que está em grande expansão no mercado de desenvolvimento de jogos – não tem nenhuma solução nativa relacionada às necessidades de um motor de *storytelling*, o tema de desenvolver uma ferramenta de gerenciamento de estórias integrada ao *Unity3D* surgiu naturalmente.

Além disso, a ideia geral para o tema foi iniciada durante um estágio realizado pelo autor deste trabalho em uma empresa de desenvolvimento de jogos. Uma das tarefas foi incorporar eventos e diálogos em um dos projetos. Porém, devido à fatores limitantes relacionados à recursos humanos e prazos, não foi reservado um período para se realizar um estudo minucioso dos diversos fatores que deveriam ser levados em conta no momento de se desenvolver tal extensão. Consequentemente, diversos problemas foram surgindo ao longo do desenvolvimento.

Assim, esse trabalho tem como objetivo principal:

[1] *http://unity3d.com/unity*

Propor e desenvolver uma ferramenta de criação e gerenciamento de estórias integrada de forma transparente ao Unity3D.

1.4 Organização do Trabalho

O Capítulo 2 apresenta a definição do que é um motor de *storytelling* e os tipos de estória em jogos. Além disso, são explicados os *design patterns* e padrões de gerência da configuração utilizados durante o desenvolvimento deste trabalho. O Capítulo 3 apresenta como foi feito o desenvolvimento deste trabalho abordando os seguintes pontos: escolha do tema, levantamento bibliográfico, equipamentos utilizados, configuração do ambiente, fluxo de trabalho e boas práticas, versionamento e implementação. O Capítulo 4 apresenta o *plugin NTeraction*, que é o produto deste trabalho. Por fim, o Capítulo 5 apresenta as conclusões deste texto, as contribuições desta proposta bem como os trabalhos futuros.

2 Referencial Teórico

2.1 Taxonomia

Antes de apresentar o que vem a ser um motor de *storytelling*, é preciso definir os termos e conceitos que serão utilizados ao longo do trabalho.

2.1.1 *Interaction Object* (Objetos de Interação)

Os *interaction objects* são todas as entidades de um jogo que podem disparar uma sequência de interações em uma dada situação. Alguns exemplos de situações de disparo são: conversar com um *NPC* (*Non-Player Character*), eliminar um inimigo, examinar um objeto do cenário.

2.1.2 *Events* (Eventos)

Os *events* são valores binários que representam o estado em que a estória do jogo se encontra em um dado momento. (o valor 1 (um) representa o estado ativo, e o valro 0 (zero), o estado inativo).

2.1.3 *Interactions* (Interações)

As *interactions* são as diversas interações que podem ser desde manipulação dos estados dos *events* do jogo, até ativação funcionalidades específicas dos jogos. Alguns exemplos de funcionalidades são: abrir sistema de diálogos, popular diálogos, manipular gatilhos, alterar música de fundo, reduzir volume e etc.

2.1.4 *Interaction Tree* (Árvore de Interação)

Todo *interaction object* possui uma *interaction tree*. Essas *interaction trees* são um conjunto de sequências de interações citados na Seção 2.1.1. Quando uma interação com um *interaction object* ocorre, uma das sequências de *interaction* de sua *interaction tree* é percorrida de acordo com os estados dos *events* do jogo em um dado momento.

2.1.5 Exemplo

Com o intuito de facilitar o entendimento das terminologias definidas, a seguir são ilustrados exemplos de cenários contextualizados nas terminologias definidas.

2.1.5.1 Conversas com *NPCs*

Em diversos jogos existem *NPCs*, que são personagens que não são controlados pelo jogador. Para aumentar a imersão do jogador na estória do jogo, muitas vezes é possível conversar com esses *NPCs*. Normalmente cada *NPC* possui uma fala única em um dado momento da estória do jogo.

Neste contexto, os *interaction objects* são todos os *NPCs* que podem estabelecer uma conversação com o jogador. Alguns *events* podem ser informações atreladas ao fato de uma conversação já ter sido estabelecida com um *NPC* em particular. As *interactions* são as frases que pertencem aos diálogos com os *NPCs* e manipulações diretas nos estados dos *events*. Finalmente, as *interaction trees* são as sequências de frases que um *NPC* possui.

2.1.5.2 Eliminação de um inimigo

Em diversos jogos existem inimigos denominados chefes, que são importantes para o fluxo da estória. Quando esses chefes são derrotados diversos acontecimentos na estória são liberados para que o jogador possa ter acesso a eles.

Neste contexto, os *interaction objects* são todos os inimigos que quando derrotados devem dar continuidade de alguma forma para a estória do jogo. Os *events* são as informações relacionadas à eliminação ou não de um inimigo. As *interactions* são manipulações diretas nos estado dos *events*. Finalmente, as *interaction trees* são sequências de *events* que devem ser ativados e quais devem ser desativados quando um inimigo é derrotado.

2.1.5.3 Objetos do cenário examinados

Em diversos jogos alguns objetos que pertencem ao cenário são passíveis de interação. Um exemplo clássico são alavancas que deve ser ativadas para que uma porta seja aberta e, assim, dar continuidade à estoria do jogo.

Neste contexto, os *interaction objects* são as alavancas e as portas. Os *events* são as informações relacionadas ao estado das alavancas. As *interactions* são manipulações diretas nos estado dos *events*. Finalmente, as *interaction trees* são sequências dos *events* que devem ser ativados e dos que devem ser desativados quando uma alavanca é ativada ou desativada.

2.2 Motor de *Storytelling*

Antes de descrever o que é um motor de *storytelling*, é preciso definir bem alguns termos como *storytelling*, mecânica central e interface de usuário.

2.2.1 *Storytelling*

Em jogos convencionais, manter o jogador imerso na estória pode parecer um tanto difícil. Por diversas vezes ocorrem transições entre momentos em que a estória é contada ao jogador e momentos em que ele necessita realizar certos objetivos, fazendo com que haja uma quebra na imersão do jogador (A.ERNERST, 2010).

Entretanto, (A.ERNERST, 2010) ressalta que muitos jogos conseguem realizar a mistura entre os aspectos de contar a estória e de realizar as ações de maneira harmoniosa, o que faz com que os jogadores sintam-se dentro do jogo, com a sensação de que cada decisão está afetando o fluxo de eventos.

De acordo com (S.KIM; S.MOON; S.HAN, 2011), o conceito de *storytelling* interativa pode ser entendido como um tipo de narrativa em que as interações dos usuários influenciam o fluxo de eventos ou dão a sensação de que isso está acontecendo. Quando a *storytelling* é aplicada em jogos ele também pode ser chamado de ficção interativa.

2.2.2 Mecânica Central

Mecânica central, muitas vezes referenciada como *core mechanics*, é o coração de qualquer jogo, uma vez que ele dita como o jogo deve ser jogado. Essencialmente, a mecânica central consiste nos dados e algoritmos que definem precisamente as regras e os processamentos internos de um jogo em específico (A.ERNERST, 2010).

Uma outra forma de se entender o conceito de mecânica central é associar ele aos eventos de jogabilidade que podem ser, por exemplo, permitir que o avatar corra, permitir que o avatar ataque os inimigos, interpretar uma sequência de entradas que são entendidos como uma combinação que dispara uma ação especial, etc.

2.2.3 Interface de Usuário

A interface de usuário é um conceito familiar uma vez que ele é fundamental em qualquer *software* que tenha interação com usuário, mas quando aplicada à jogos, ela possui algumas diferenças.

Em *softwares* convencionais, a ideia da interface é tentar ser a mais eficiente possível, apresentando ao usuário a sua funcionalidade de maneira clara. Entretanto, em jogos, muitas vezes não se deseja que a interface de usuário seja tão eficiente, de modo a se privilegiar a diversão. Esta característica pode ser observada nos casos em que se deseja deixar o jogo mais desafiador ou esconder informações do usuário (A.ERNERST, 2010).

De acordo com (A.ERNERST, 2010), em um jogo a interface do usuário é o componente que realiza a mediação entre a mecânica central e o usuário, como pode ser visto na Figura 1, na qual o seguinte fluxo é representado: o usuário realiza uma entrada na

interface de usuário, essa entrada é interpretada pela interface do usuário e passada na forma de ação para a mecânica central. A mecânica central interpreta a ação e determina os efeitos no jogo, retornando assim uma informação para a interface de usuário sobre como representar o jogo para o usuário.

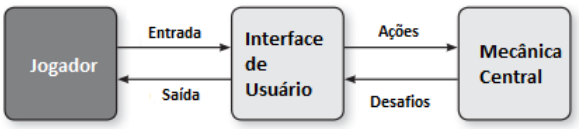

Figura 1 – Relação entre mecânica central, interface de usuário e usuário. Adaptado de (A.ERNERST, 2010).

Caso esse processo seja feito de maneira bem suave e natural, o jogador passa a associar a entrada com a ação, utilizando-a de forma automática sempre que julgar necessário.

Vale ressaltar que segundo a definição de interface de usuário de (A.ERNERST, 2010), a interface não é meramente uma geradora de saídas e receptora de entradas, mas sim, a forma como o mundo do jogo é apresentado para o usuário, o que inclui as imagens, sons do jogo e até, em casos mais específicos, as vibrações do jogo caso o recurso de vibrar o controle estiver disponível.

2.2.4 Motor de *Storytelling*

Com os conceitos de *storytelling*, mecânica central e interface de usuários entendidos, é possível descrever o que é o motor de *storytelling*.

No desenvolvimento de jogos que incluem uma estória, existe a necessidade de se realizar de forma coesa a sincronização entre os eventos de jogabilidade, gerenciados pela mecânica central, e os eventos que representam a estória, a fim de dar a sensação de uma sequência uniforme que represente uma estória e providencie entretenimento do usuário.

Portanto, é necessário uma entidade que gerencie os eventos da estória e realize a sincronização com a mecânica central. Tal entidade é chamada de motor de *storytelling*, que segundo (A.ERNERST, 2010), é o terceiro grande componente de um jogo. No entanto, esse componente nem sempre é necessário.

É de responsabilidade do motor de *storytelling* definir a forma que será utilizada para narrar o jogo. Uma das formas utilizadas é o uso de diálogos textuais, e hoje em dia é comum sincronizar os diálogos textuais com dublagens. Além disso, existem diversas outras formas de narrar como vídeos cinematográficos ou chamadas predefinidas de ações no jogo, conhecidas como *cutscenes*.

A Figura 2 apresenta como o motor de *storytelling* interage com a interface de usuário e mecânica central.

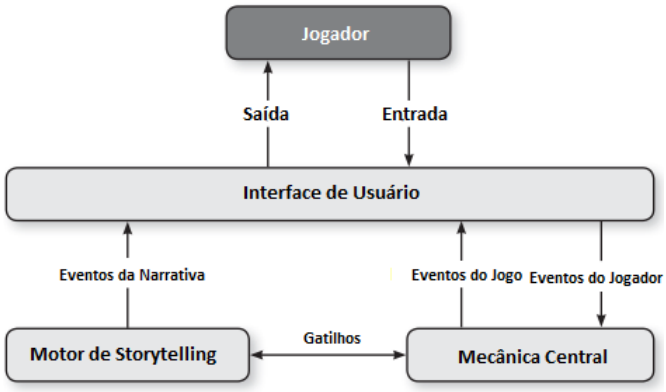

Figura 2 – Relação entre mecânica central, interface de usuário e usuário. Adaptado de (A.ERNERST, 2010).

A relação entre interface de usuário e mecânica central é preservada conforme apresentada na Figura 1, porém há o acréscimo do motor de *storytelling* que possui um canal bidirecional de comunicação com a mecânica central, cuja função é mantê-los conectados para que haja uma sincronização harmoniosa entre seus eventos.

Além disso, o motor de *storytelling* possui um canal com a interface de usuário que tem como objetivo passar as informações necessárias para a interface representar os eventos de narrativa, que podem ser de diversos tipos como descrito anteriormente nesta seção.

2.3 Tipos de Estórias em Jogos

Desde o início dos jogos digitais, os *designers* de jogos sempre estiveram intrigados com a ideia de como fazer com que o jogador pudesse influenciar o rumo da estória dos jogos.

Neste contexto, diversos tipos de estórias de jogos surgiram e, segundo (A.ERNERST, 2010), todas as estórias se encaixam em um dos seguintes tipos: linear, não-linear, linear/não-linear. Tais classificações serão descritas nas próximas seções.

2.3.1 Estória Linear

De acordo com (A.ERNERST, 2010), a estória linear em jogos digitais é similar à estória linear em qualquer outro meio: o usuário não pode mudar o enredo nem o final da estória. Entretanto, há algumas diferenças entre estórias lineares em jogos e em outras mídias.

A principal diferença diz respeito à interação do usuário com o meio que a estória se passa, isto é, em um jogo, mesmo que o usuário não possa mudar a estória, ele ainda

interage com o jogo. Em outras mídias normalmente o usuário tem uma participação apenas passiva, que é ser apenas um espectador.

Desta forma, a estória linear em jogos é considerada uma estória interativa, em que a contribuição do usuário é limitada apenas através de ações no avatar que não influenciam no fluxo de eventos da estória.

A Figura 3 ilustra a estrutura de uma estória linear em que cada nós é um etapa da estória.

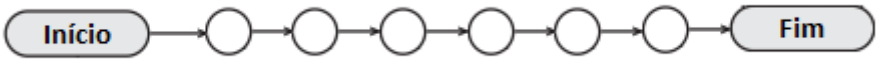

Figura 3 – Estrutura de uma estória linear. Adaptado de (A.ERNERST, 2010).

2.3.2 Estória Não-linear

A estória puramente não-linear em jogos digitais, também conhecida como estória em ramificação, permite que o usuário possa influenciar eventos e mudar a direção da estória, o que faz com que diferentes experiências sejam obtidas a cada vez que se joga do começo ao fim.

Nesse caso, a implementação do motor de *storytelling* é muito mais complexo, uma vez que é preciso que o motor de *storytelling* e a mecânica central troquem informações constantemente para saber em qual das possíveis situações o jogo se encontra e como cada elemento do jogo deve reagir.

A partir do momento em que se considera que a estória pode divergir, é preciso saber como ela pode divergir. Assim, são introduzidos os conceitos de pontos de divergência e eventos.

Pontos de divergência são pontos no fluxo de acontecimentos de um jogo em que ocorre a mudança no rumo da estória.

Os eventos são as tomadas de decisões que ocorrem dentro do jogo que direcionam o rumo da estória. Um evento pode ser classificado como: imediato, deferido e acumulativo. A seguir cada um destes tipos são explicados e ilustrados.

imediato Tipo de evento que faz com que o rumo da estória seja modificado imediatamente. A Figura 4 ilustra a forma como o evento imediato funciona.

Deferido Tipo de evento que influencia um ponto de divergência que seja posterior ao acontecimento do evento. A Figura 5 ilustra um evento deferido.

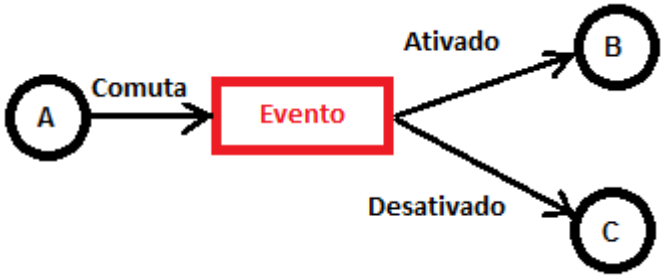

Figura 4 – Exemplo de evento imediato.

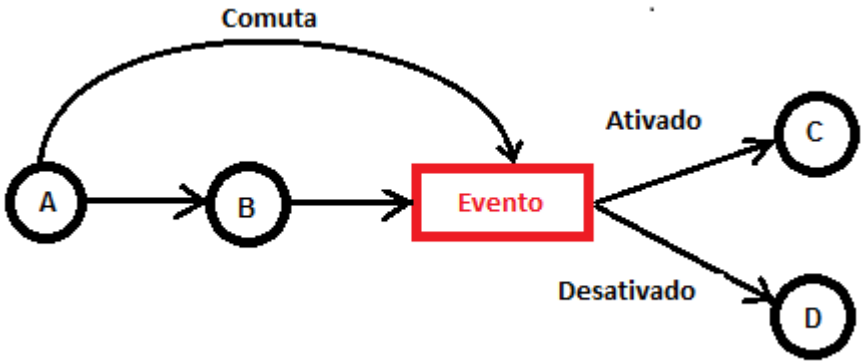

Figura 5 – Exemplo de evento deferido.

Acumulativo Tipo de evento que acumula informações, que caso satisfaçam alguma condição imposta, afetam algum ponto de divergência da estória. A Figura 6 apresenta um exemplo de evento deferido.

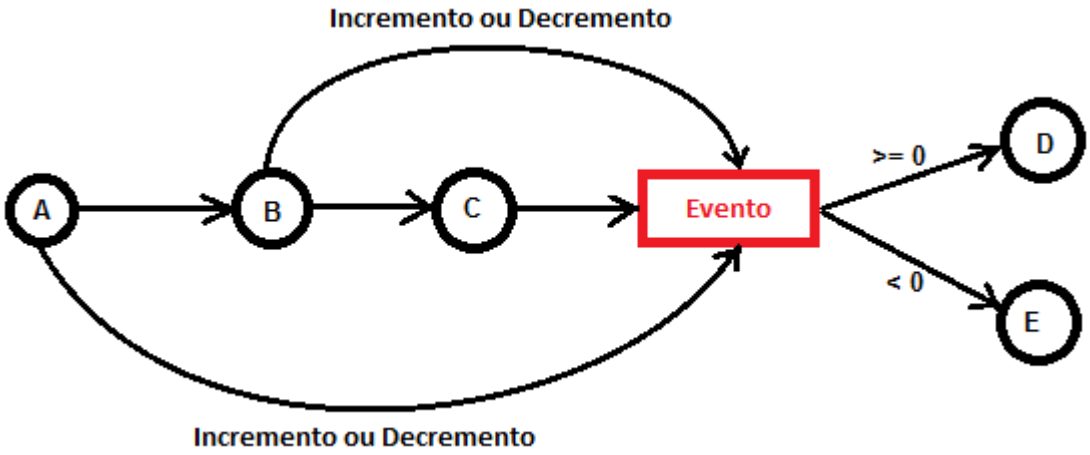

Figura 6 – Exemplo de evento acumulativo.

Com o intuito facilitar o entendimento do conceito de estória não-linear, a Figura 7 apresenta o modelo em forma de diagrama, em que cada nó representa um ponto de divergência e as setas representam os eventos que influenciam o fluxo na estória.

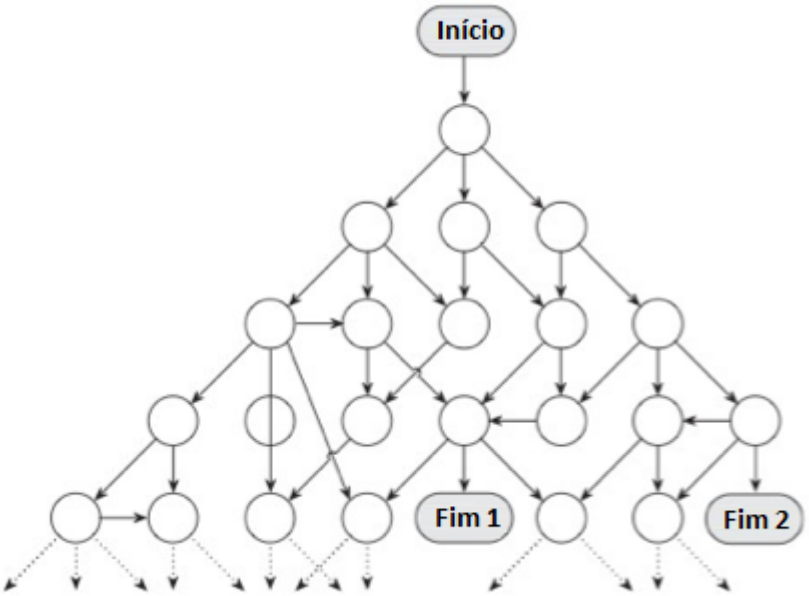

Figura 7 – Estrutura de uma estória em ramificação. Adaptado de (A.ERNERST, 2010).

2.3.3 Estória Linear/Não-linear

A estória linear/não-linear é exatamente o que o nome sugere, isto é, uma mistura entre estória linear e não-linear. Devido ao fato desse tipo de estória ser uma mistura entre os dois conceitos apresentados anteriormente, todos os conceitos apresentados na Seção 2.3.2 são aplicáveis neste tipo também.

A característica linear desse tipo de estória vem do fato de que existem pontos na estória que o jogador deve passar obrigatoriamente, porém o que ocorre entre esses pontos obrigatórios não necessariamente precisa ser linear, obtendo-se assim também uma natureza não-linear.

Para facilitar o entendimento é apresentada a Figura 8, onde cada nó representa um ponto de divergência, as setas representam o eventos que influenciam a divergência e os nós mais escuros representam os pontos da estória que são inevitáveis, os quais podem ser entendidos como pontos de convergência obrigatórios.

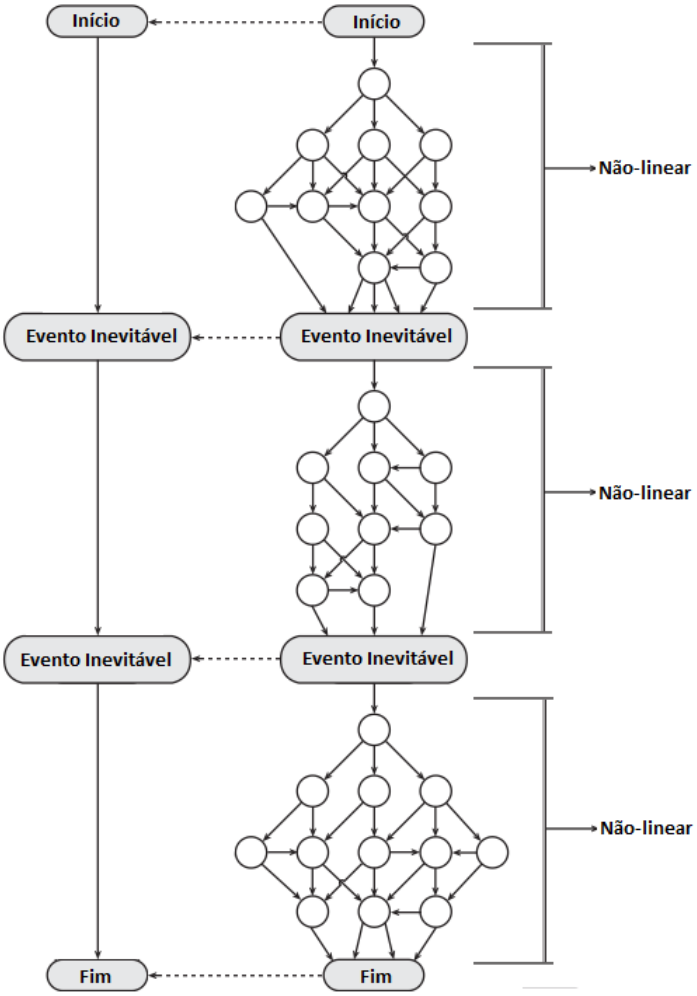

Figura 8 – Estrutura de uma estória linear/não-linear. Adaptado de (A.ERNERST, 2010).

2.4 Padrões de Gerência da Configuração

Uma das atividades mais importantes do desenvolvimento de *software* é a gerência da configuração, pois ela está atrelada diretamente a todos os aspectos que tangem o processo de desenvolvimento. Uma boa gerência da configuração pode aumentar a produtividade do desenvolvimento de um *software*, mas caso seja feita de maneira errônea, pode atrasar o projeto e criar frustração para os desenvolvedores (S.BERCZUK; B.APPLETON, 2003).

Para ajudar a guiar a gestão da configuração, (S.BERCZUK; B.APPLETON, 2003) define um conjunto de práticas conhecido como padrões de gerência da configuração. Serão abordado nesta seção somente os padrões adotados durante o desenvolvimento deste trabalho.

2.4.1 *Mainline*

Segundo (S.BERCZUK; B.APPLETON, 2003), quanto maior o número de *codelines*, mais difícil é identificar em qual estado o projeto se encontra em um momento específico. Neste contexto, a forma mais simples de se minimizar a grande quantidade de *codelines* é possuir uma *codeline* principal chamado de *mainline*.

Utilizando este padrão, a ideia é fazer com que no final do desenvolvimento todas as outras *codelines* convirjam em algum momento para a *mainline*.

Um problema deste padrão é o risco de algum desenvolvedor realize um conjunto de alterações no projeto, que levem o mesmo a parar de funcionar. Para resolver este problema, pode-se utilizar o padrão *active development line*, apresentado a seguir.

2.4.2 *Active development line*

De acordo com (S.BERCZUK; B.APPLETON, 2003), o padrão *active developement line* serve para aumentar a garantia de que a *mainline* esteja em um estado estável.

Este padrão sugere que no momento de desenvolver uma nova funcionalidade ou corrigir um defeito, uma cópia da *mainline* seja feita para uma outra *codeline*. Todas as alterações devem ser feitas nesta nova *codeline* até que o código esteja estável o suficiente, seguindo um determinado critério. Uma vez estáveis, as alterações feitas na nova *codeline* podem ser convergidas para a *mainline*.

2.4.3 *Private Workspace*

Definido por (S.BERCZUK; B.APPLETON, 2003), o padrão *private workspace* enfrenta o problema de vários desenvolvedores trabalhando junto.

O padrão sugere que os desenvolvedores necessitam ter controle sobre todo o projeto para realizar testes e criar *builds*. Para isso, cada desenvolvedor possui uma *codeline* própria, na qual ele esteja constantemente adicionando funcionalidades e realizando modificações.

2.5 *Design Patterns*

De acordo com (E.GAMMA et al., 1995), os *design patterns* são descrições de soluções para problemas recorrentes no paradigma de orientação a objetos. Ao todo, (E.GAMMA et al., 1995) define 23 *patterns*, porém neste trabalho serão apresentados apenas os dois que foram importantes para a arquitetura do projeto: *Singleton* e *Factory Method*.

2.5.1 *Singleton*

Segundo (E.GAMMA et al., 1995), a intenção deste *pattern* é garantir que exista apenas uma instância de um determinado objeto e que exista um ponto de acesso global para o mesmo. Esse padrão é útil quando é necessário exatamente uma instância para coordenar ações através do sistema como um todo.

Por este padrão prover um acesso global, é muito fácil ocorrer o acoplamento de classes que não deveriam ter relação entre si. Desta forma, é preciso evitar o uso constante e inadequado deste *pattern*.

2.5.2 *Factory Method*

De acordo com (E.GAMMA et al., 1995), a intenção deste *pattern* é definir uma interface para criação de objetos de uma classe pai e deixar com que suas subclasses decidam que classe instanciar. Ao utilizar este *pattern*, é possível definir uma arquitetura que seja customizável.

É bem comum que as interfaces do *factory methods* sejam definidos por um *framework* arquitetural e as implementações do *factory method* em subclasses feitas por desenvolvedores que fazem uso do *framework*.

3 Desenvolvimento

3.1 Escolha do Tema

A ideia inicial para o tema do trabalho surgiu durante um estágio na GEL Lab[1], empresa de desenvolvimento de jogos, onde uma das tarefas foi criar um sistema de diálogos para um projeto em específico.

Durante o desenvolvimento do sistema, percebeu-se que um sistema de diálogos envolvia muito mais do que apenas diálogos, mas devido ao prazo de entrega inflexível, um estudo mais aprofundado das diversas variáveis que poderiam afetá-lo não foi feito.

Assim que o sistema foi finalizado e integrado ao projeto, os *Designers* começaram a utilizar a ferramenta. Entretanto, diversos problemas foram surgindo relacionados à organização e escalabilidade do projeto tais como:

- Testar uma determinada parte da estória isoladamente sem a necessidade de começar o jogo desde o início;
- Encontrar o ponto certo no sistema para se adicionar um conjunto de elementos relacionados à estória sem comprometer o que já havia sido feito antes;
- Usabilidade da ferramenta que comprometia a produtividade do *Designer*.

Neste contexto, surgiu a ideia de desenvolver um sistema similar, que tratasse das questões que ficaram em aberto e que se mostraram importantes.

3.2 Levantamento Bibliográfico

Uma vez escolhido o tema e escopo do trabalho, o primeiro passo consistiu em um levantamento bibliográfico, com o intuito de tanto identificar o grau de relevância de tal tema quanto analisar o que já foi desenvolvido na área.

Os levantamentos bibliográficos acadêmicos foram realizados através de diversas revisões estruturadas nas bibliotecas digitais da *Michigan State University* e da *Rasselear Polytechnic Institute*. Além desses, diversas leituras superficiais foram feitas em livros, com o intuito de identificar o que exatamente contempla uma *storytelling engine*.

[1] *http://www.gel.msu.edu/*

Durante esta etapa, identificou-se que o sistema de diálogos era apenas uma pequena porção de um sistema maior, denominado de *storytelling engine*, que é o produto final deste trabalho.

3.3 Equipamentos Utilizados

Para o desenvolvimento do trabalho, foi utilizado um *MacBook Pro* de 15 polegadas, cuja configuração é: Processador Intel Core i7 de 2,4 GHz, memória de 4 GB 1333 MHz DDR3, placa de vídeo AMD Radeon HD 6770M com 1 GB de memória dedicada GDDR5.

Baseado nas decisões de configuração de ambiente citado na Seção 3.4, a ferramenta poderia ter sido desenvolvida tanto em *Windows* quanto em *OSX*, porém como o máquina do autor deste trabalho era um *MacBook Pro*, o sistema operacional utilizado para desenvolvimento foi o *OSX Mavericks*.

3.4 Configuração do Ambiente

Em seguida, foi feito a configuração do ambiente de trabalho. Uma vez que a *game engine* escolhida para integrar a *storytelling engine* foi a *Unity3d*, a versão instalada no sistema operacional foi a 4.3.1, que na época do início do trabalho era a versão mais recente.

Além disso, outras ferramentas foram instaladas para auxiliar o desenvolvimento. O *Git* foi a ferramenta utilizada para versionar o código durante o desenvolvimento para diminuir as chances de perda de trabalho. A versão utilizada foi a 1.9.2, que era a mais recente no início do desenvolvimento do trabalho.

Para hospedar o repositório *Git* do projeto, o serviço escolhido foi o *Bitbucket* da *Atlassian*. A escolha desse serviço foi feita porque ele permite a criação gratuita de repositórios privados. Assim, o projeto remoto foi criado e os repositórios locais configurados adequadamente.

Além do fato da principal linguagem de desenvolvimento para *Unity3D* ser C#, por ser multiplataforma e gratuita, a IDE utilizada e instalada foi a *MonoDevelop*, versão 4.0.

3.5 Fluxo de Trabalho e Boas Práticas

A fim de aumentar a garantia de que o produto do trabalho seria finalizado com êxito, no prazo estimulado, e as chances de fatores inusitados que prejudicassem o pro-

jeto diminuíssem, um fluxo de trabalho e boas práticas da engenharia de *software* foram adotados.

Vale ressaltar que, por se tratar de um desenvolvimento feito por apenas um membro, não se adotou ou adaptou um processo ou metodologia específico, mas foram utilizadas características de algumas delas.

3.5.1 Desenvolvimento Iterativo Incremental

O desenvolvimento foi feito de maneira iterativa incremental por meio de *sprints* curtas de no máximo uma semana. Assim, tanto os documentos e o código do projeto foram evoluindo ao longo do projeto. No caso do código, sempre que necessário o mesmo era refatorado e a documentação referentes ao desenvolvimento atualizada.

3.5.2 Documentação

Os documentos gerados a fim de facilitar o entendimento e manutenibilidade do projeto foram: diagramas de classe, comentários no código, casos de uso e diagramas de casos de uso.

a) Diagrama de Classe

O padrão utilizado para fazer o diagrama de classe foi o *UML* 2.0. Com o intuito de fazer o diagrama de classe ser um documento para o entendimento da arquitetura do projeto em um nível mais abstrato, apenas os principais atributos e métodos necessários para o entendimento de cada classe foram apresentados. Um exemplo desta forma é ilustrado na Figura 9.

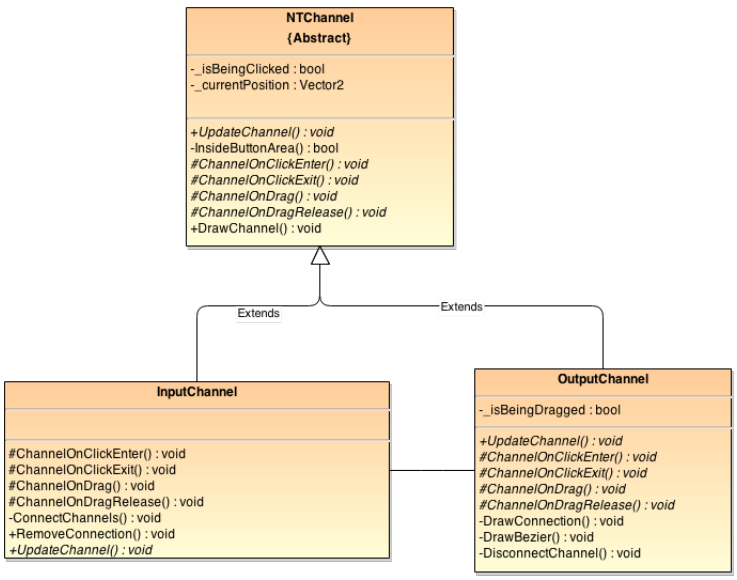

Figura 9 – Exemplo da abordagem escolhida para o diagrama de classe da arquitetura.

b) Comentários

O padrão utilizado para realizar os comentários foi o sugerido pela *Microsoft* no guia de programação para C#[2]. Este padrão faz uso de *tags* do *XML* que facilitam a geração automática de documentação do código a partir do código-fonte. As principais são:

- *<summary>*: descrição do método;
- *<returns>*: descrição do retorno ou retornos esperados do método;
- *<param name=*"nome do parametro"*>*: descrição de cada parâmetro de entrada do método.

O Código 3.1 exemplifica o uso dos cometários seguindo o padrão descrito.

Código 3.1 – Exemplo de comentário utilizando *tags* do *XML*

```
1  /// <summary>
2  /// Check if it should resize the window
3  /// </summary>
4  /// <returns>
5  /// <c>true</c>, if should resize the window,
6  /// <c>false</c> otherwise.
7  /// </returns>
8  /// <param name="currentEvent">current event</param>
9  private bool ShouldResize(Event currentEvent)
10 {
11         return (InsideResizableArea(currentEvent.mousePosition)
12                 && currentEvent.type == EventType.MouseDown);
13 }
```

c) Casos de Uso

Os casos de uso, que estão no Anexo A, são utilizados para descrever o comportamento do sistema quando utilizados pelo usuário. O padrão utilizado neste trabalho para fazê-los foi o casual, o qual segundo (A.COCKBURN, 2001), tem os seguintes campos, com suas devidas descrições:

Título Interação do usuário com o sistema. Sempre se inicia com um verbo no infinitivo;

Ator principal Principal usuário que participará do caso de uso;

Estória Um ou dois parágrafos descrevendo como a interação do usuário com o sistema ocorre. Deve descrever tanto a entrada do usuário quanto a saída do sistema.

[2] *http://msdn.microsoft.com/pt-BR/library/b2s063f7.aspx*

d) Diagramas de Caso de Uso

O padrão utilizado para fazer o diagrama de caso de uso é o *UML* 2.0, em que são representados atores se relacionado com seus respectivos casos de uso. O objetivo deste diagrama com relação ao projeto é englobar as diversas funcionalidades do sistema em um documento. Um exemplo de diagrama de caso de uso é apresentado na Figura 10.

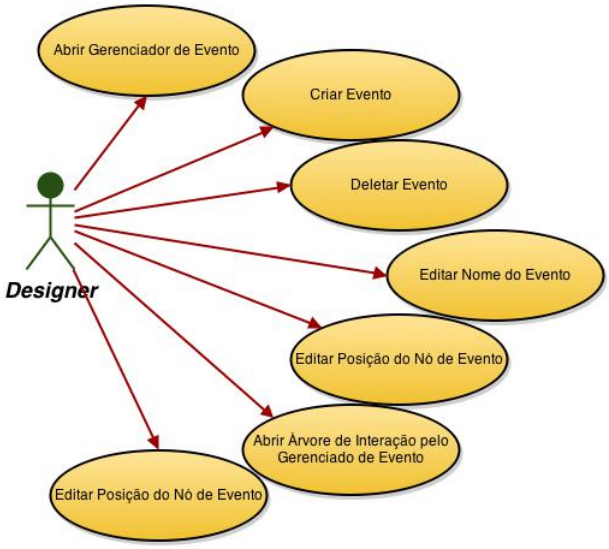

Figura 10 – Exemplo de diagrama de caso de uso.

3.5.3 Versionamento

O objetivo do *Git* como ferramenta de controle de versão foi prevenir perda de trabalho, tanto relacionado ao código fonte e *assets* do projeto quanto de outros documentos, como por exemplo: documentação do produto, trabalho escrito de conclusão de curso, etc.

Para que o versionamento fosse feito de maneira estruturada, os três padrões de gerência da configuração de software definidos na Seção 2.4 foram utilizados.

3.5.4 *Design Patterns*

Para aumentar a qualidade do código gerado e facilitar o entendimento, alguns *design patterns* foram utilizados em diferentes partes do código. São eles:

a) ***Singleton.*** O *singleton* foi utilizado para facilitar a propagação da informação de dados relacionados ao estado do *mouse* no contexto da ferramenta.

b) ***Factory Method.*** O *factory method* foi utilizado na criação de instâncias que respondem a uma interface em particular. Mais especificamente, na arquitetura desse

projeto o *factory method* foi utilizado em conjunto com a capacidade do C# de reflexão, também conhecida como introspecção, para realizar de maneira mais estruturada a persistência dos dados entre a aplicação e o *XML*.

c) **Outros *Patterns*** Além dos dois *patterns* citados, alguns outros utilizados na arquitetura do código foram *template method*, *mediator* e *chain of responsability*.

3.6 Implementação

Como foi dito na Seção 3.5.1, o desenvolvimento foi feito de maneira iterativa e incremental.

Neste contexto, a primeira etapa do desenvolvimento foi criar um editor de árvore genérico. Assim, definiu-se que as funcionalidades do editor de árvore seriam: ligar os nós, navegar pela árvore, mover os nós através de operações de arrastar e soltar (*drag and drop*) e redefinir o tamanho dos nós. A Figura 11 ilustra o diagrama de classes desta primeira etapa. Nela, a classe *NTInteractionTreeWindow* renderiza um conjunto de nós representados pela classe *NTNode*.

Os *NTNodes* possuem canais de saída e entrada que são representados respectivamente pelas composições com as classes *NTOutputChannel* e *NTInputChannel*. Através dessas duas informações para cada *NTNode*, e com o auxílio do *singleton* de *NTMouseController*, foi possível realizar a aresta de conexão entre dois *NTNodes*, conforme ilustrado na Figura 11.

Para navegar pela árvore foi implementada a classe *NTInteractionCamera*, que encapsula informações relacionados à área do grafo que deve ser renderizada e que possui a funcionalidade de gerar um pequeno mapa de navegação. As informações da *NTInteractionCamera* são passadas para os nós através da *NTInteractionTreeWindow*.

Métodos de *NTNode* foram criados para mover e redefinir o tamanho de cada um dos nós. Esses método são chamados pelo método `Draw()`, o qual é chamado a cada *frame* da *NTInteractionTreeWindow*.

Não menos importante, grande parte dos comportamentos de *NTOutputChannel* e *NTInputChannel* foram herdadas da classe abstrata *NTChannel*. Além disso, vários métodos virtuais de *callback* relacionados à interação do *designer* com um objeto herdado de *NTChannel* são definidos e sobrescritos por suas classes filhas.

Uma vez que as funcionalidades básicas de um editor de árvore foram criadas, a segunda etapa de implementação foi adaptar o código de forma que fosse possível a criação de novos tipos de nós de uma maneira extensível.

Primeiramente, a classe *NTNode* passou a ser uma classe abstrata composta de

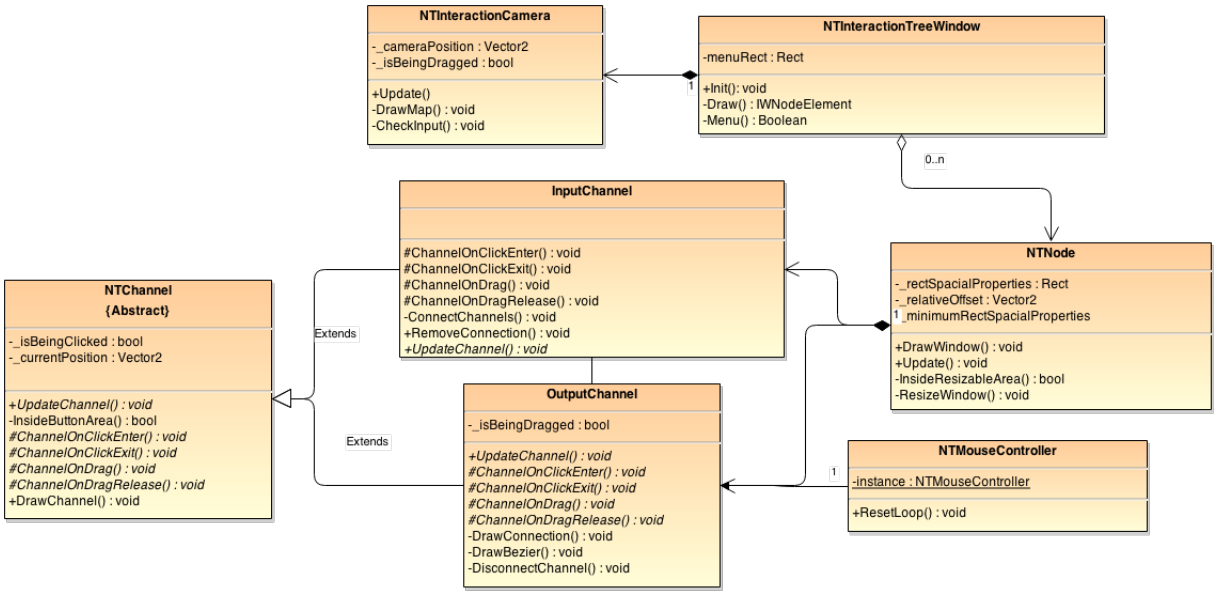

Figura 11 – Diagrama de classes da etapa de desenvolvimento do editor de árvore.

NTComponents, como pode ser visto na Figura 12. Desta forma, para estender os tipos de nó basta criar uma nova classe de nó que herde de *NTNode* e criar um componente que herde de *NTComponent* e sobrescreva o método de `DrawContent()`.

Além disso, uma refatoração foi feita para que o *NTOutputChannel* se tornasse uma composição de *NTComponent* ao invés de *NTNode*. Desta forma, um nó tem um ponto de entrada único e vários pontos de saídas pertencentes a cada um dos componentes que pertencem àquele nó.

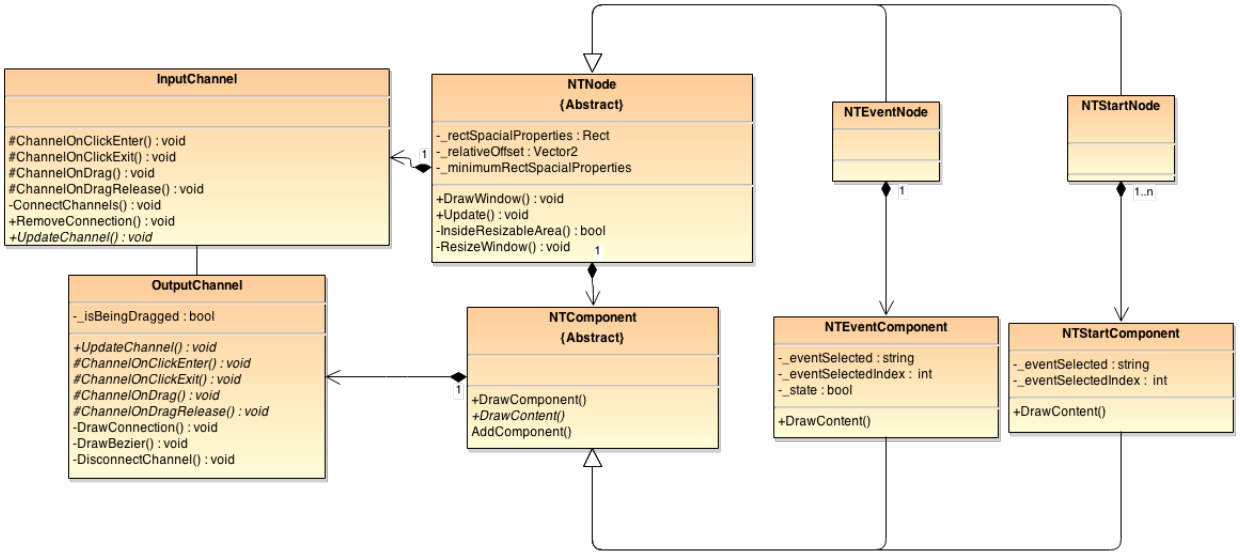

Figura 12 – Diagrama de classes da etapa de refatoração para suportar diferente tipos de nós.

Feito isso, a terceira etapa foi realizar a persistência das informações da árvore em um documento de *XML*. A persistência consistiu em salvar e carregar informações

referentes aos nós. Assim, criou-se o método `Save()` na classe *NTInteractionTreeWindow* que invoca o método `XmlPersistDataSave()` de todos nós do editor. Por sua vez, cada nó invoca o método `XmlPersistDataSave()` de todos os *NTComponents* que possui.

A operação de carregar acontece de forma semelhante à operação de salvar, mas no momento de carregar os nós o *NTInteractionTreeWidnow* faz uso do método estático `CreateEmptyNode()` da classe *NTNodeFactory* para criar um novo nó a partir do nome da classe, conforme mostra o diagrama da Figura 13. Isto foi possível porque, no momento de salvar a informação de tipo de nó que herda de *NTNode*, foi utilizada reflexão, de modo que a informação do tipo de nó é o nome da própria classe. Um processo semelhante ocorre no momento de carregar os *NTComponents*, porém fazendo uso da *NTComponentFactory*. Vale ressaltar que esses foram os pontos da implementação que se utilizou o *factory method* em conjunto com a reflexão, citados na Seção 3.5.4.

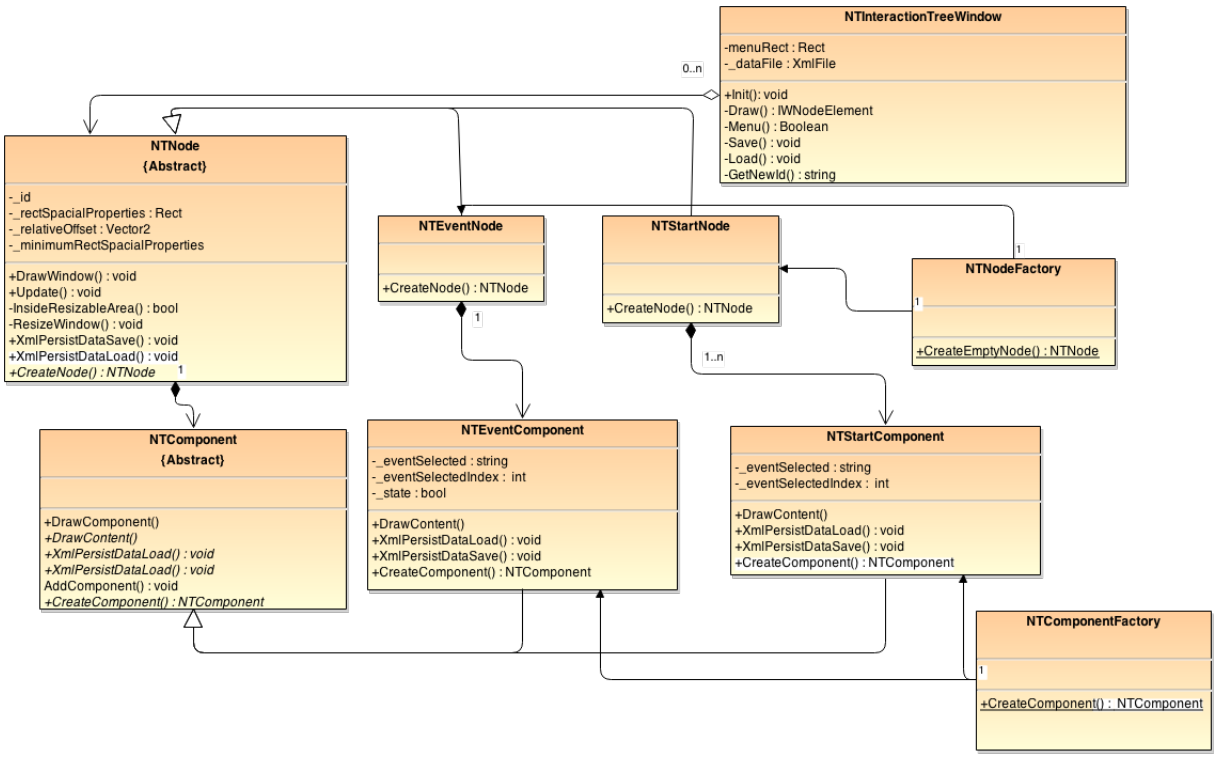

Figura 13 – Trecho do diagrama de classes da persistência.

A quinta e última etapa do desenvolvimento da ferramenta foi criar o gerenciador de eventos. A principal tarefa deste gerenciador seria criar eventos e ter um *feedback* gráfico de como os eventos estão interagindo entre si na forma de um grafo.

Para isso, foi criado a classe *NTLocalEventsWindow* composta de um conjunto de *NTLocalEventsNode* e um *NTLocalGraph*, como mostra a Figura 14.

No momento que uma instância da classe *NTLocalEventsWindow* é criada, os eventos são carregados através do método `Load()`. Após carregados, invoca-se o método `BuildGraph()`, cuja função é instanciar um *NTLocalGraph* e realizar uma tra-

vessia em todas as árvores de interações já criadas por meio da chamada recursiva de `PreorderTraversalCheck()`. Uma vez que o grafo esteja completo, ele é renderizado método `Draw()` da instância de *NTLocalEventsWindow*.

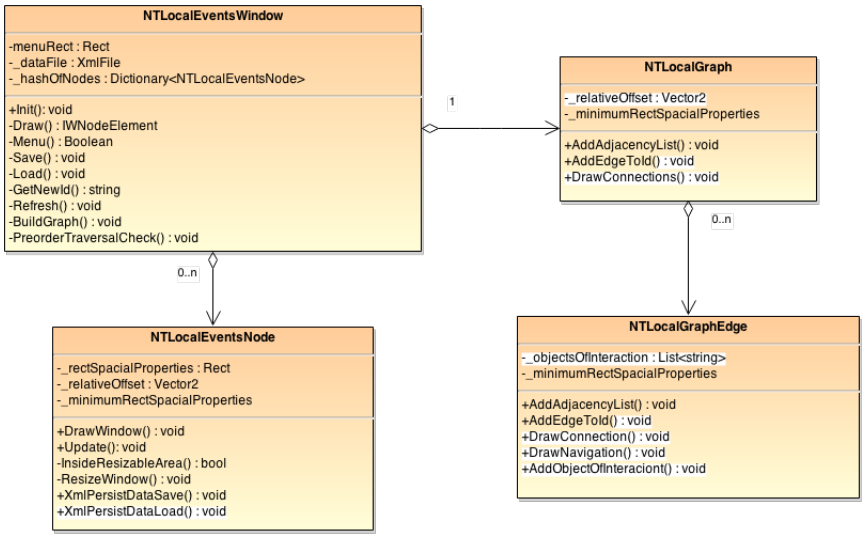

Figura 14 – Diagrama de classes do editor de eventos.

4 Resultados

4.1 NTeraction

O resultado deste trabalho foi o desenvolvimento de um *plugin* de criação e gerenciamento de estórias totalmente integrada na *game engine Unity3D*.

Este *plugin* é capaz de criar *interaction objects*, editar *interaction trees*, editar nós de interação e editar os *events*. Nas próximas subseções estas funcionalidades serão detalhadas.

4.2 Criação de *Interaction Object*

O padrão de arquitetura do *Unity3D* é o ECS, no qual todo objeto em uma cena é considerado uma entidade composta por diversos componentes. Tais componentes podem realizar desde funcionalidades primárias da *engine* como renderização e colisão até *scripts* customizados por programadores para lidar com o comportamento das entidades. Assim sendo, o padrão de entidade e componente foi seguido para fazer com que um objeto desempenhe a função de um *interaction object*.

Para adicionar o componente de *interaction object* é preciso dar um *drag and drop* do *script NTInteractionObject* em cima do objeto alvo, como mostra a Figura 15.

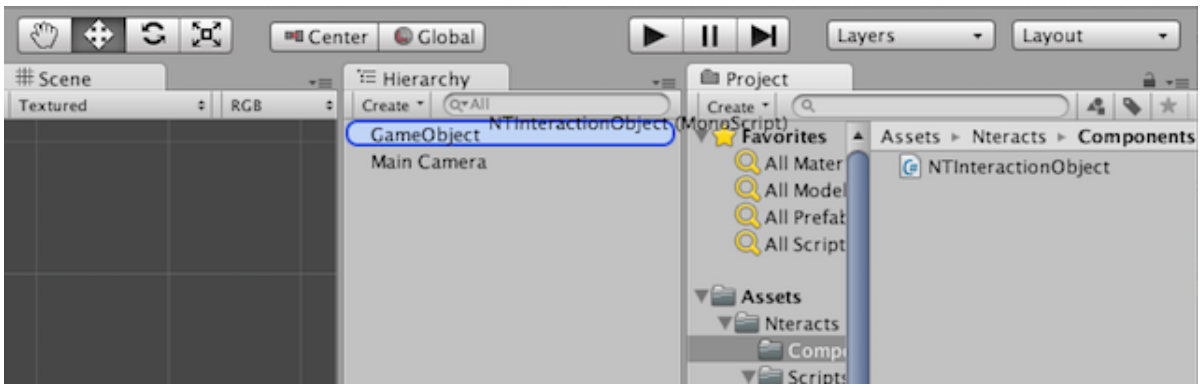

Figura 15 – Adição do componente de *interaction object* em um objeto.

Uma vez criado o componente, ao inspecionar o objeto, é possível ver o campo *Interactable Character/Object* e o botão *Edit Interaction*, como mostra a Figura 16.

O campo *Interactable Character/Object* é o identificador único de um objeto de interação. Caso um mesmo conjunto de interações possa ser compartilhado entre vários objetos, o *Designer* pode reutilizar o identificador em vários objetos.

O botão *Edit Interaction* é utilizado para abrir o editor de *interaction tree* para o identificador especificado no campo *Interactable Character/Object*.

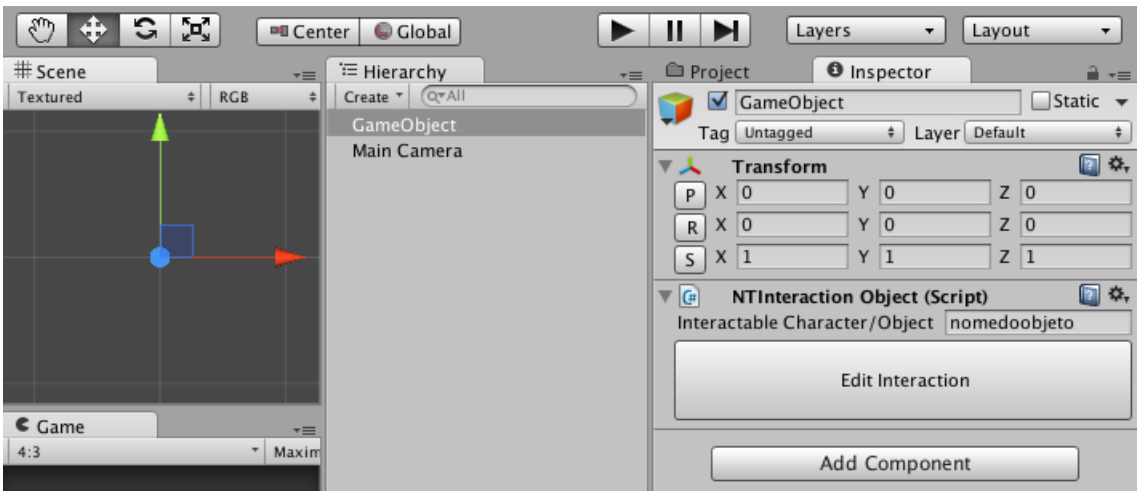

Figura 16 – Adição do componente de *interaction object* em um objeto.

4.3 Edição de *Interaction Trees*

Como explicado na Seção 4.2, ao clicar no botão *Edit Interaction* o editor de *interaction tree* é aberto. Caso ainda não exista, o editor abre uma árvore com um nó criado, conforme mostra a Figura 17.

Neste editor é possível criar outros nós ao clicar nos botões que estão abaixo da *label Node Types*. A Figura 18 mostra o editor com outros dois nós de manipulação de *events* criados.

É possível conectar os nós e assim criar várias sequências de interações citadas na Seção 2.1.1. Para isso, é preciso pressionar o botão esquerdo do *mouse* em um canal de saída e em seguida soltar no canal de entrada de outro nó.

Para indicar que a funcionalidade está em execução, enquanto o botão esquerdo do *mouse* estiver sendo pressionado, uma curva de conexão é renderizada entre a posição do *mouse* e o canal de saída do nó, conforme mostra a Figura 19. Quando a conexão é efetuada com sucesso, uma curva passa a ser renderizada entre o canal de saída e canal de entrada, como ilustra a Figura 20.

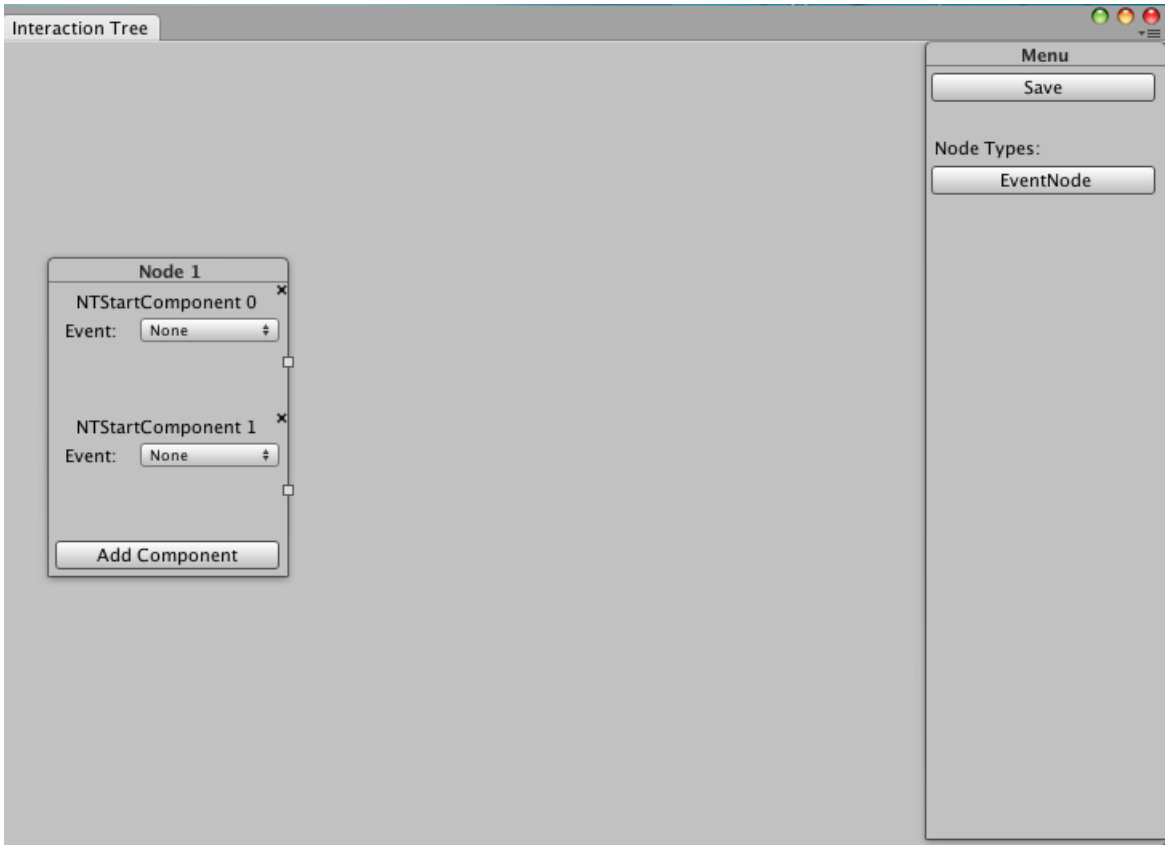

Figura 17 – Editor de *Interaction Object* aberto pela primeira vez.

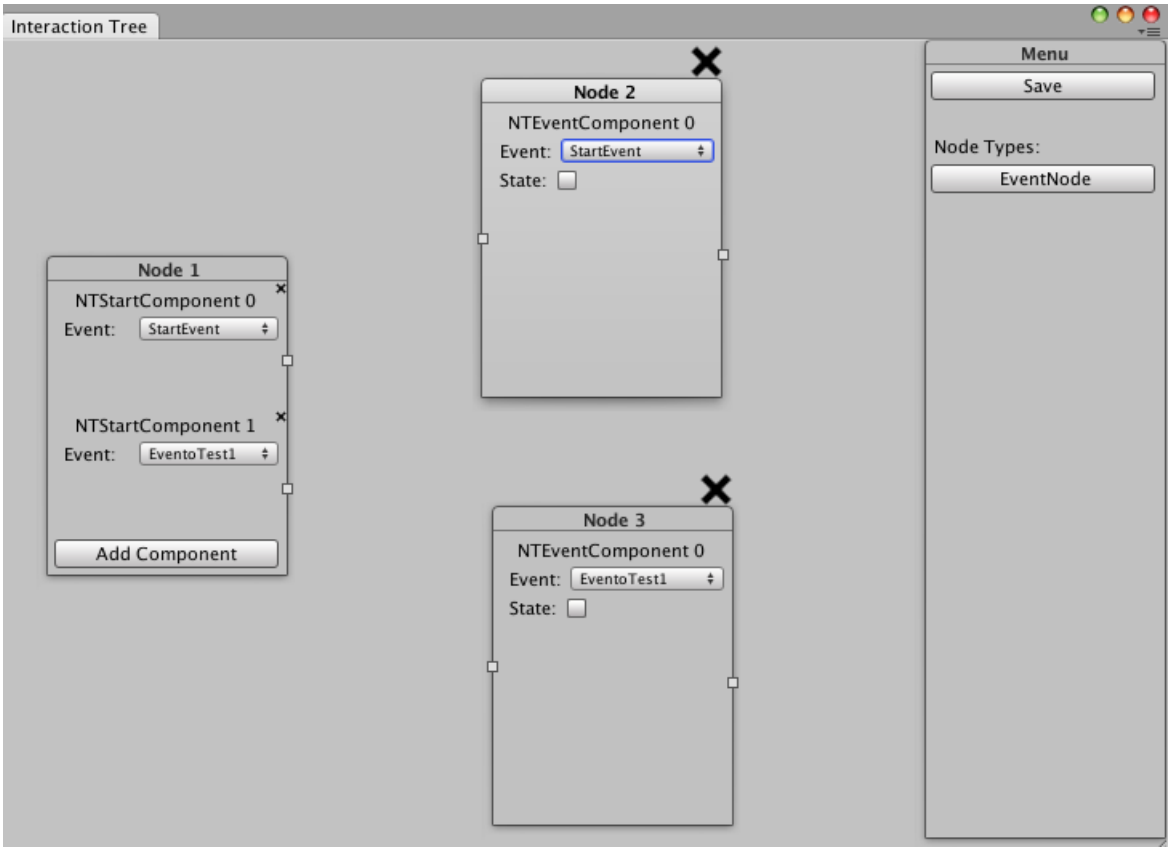

Figura 18 – Editor de *Interaction Object* com nós criados

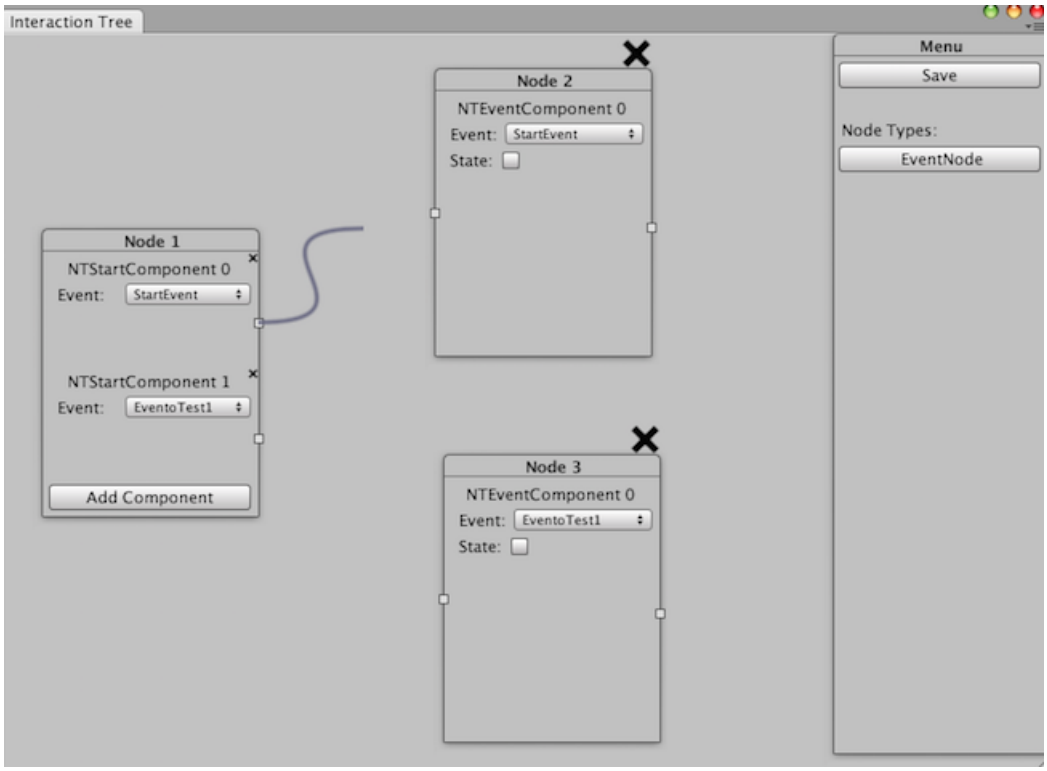

Figura 19 – Editor de *Interaction Object* com a funcionalidade de conexão em andamento.

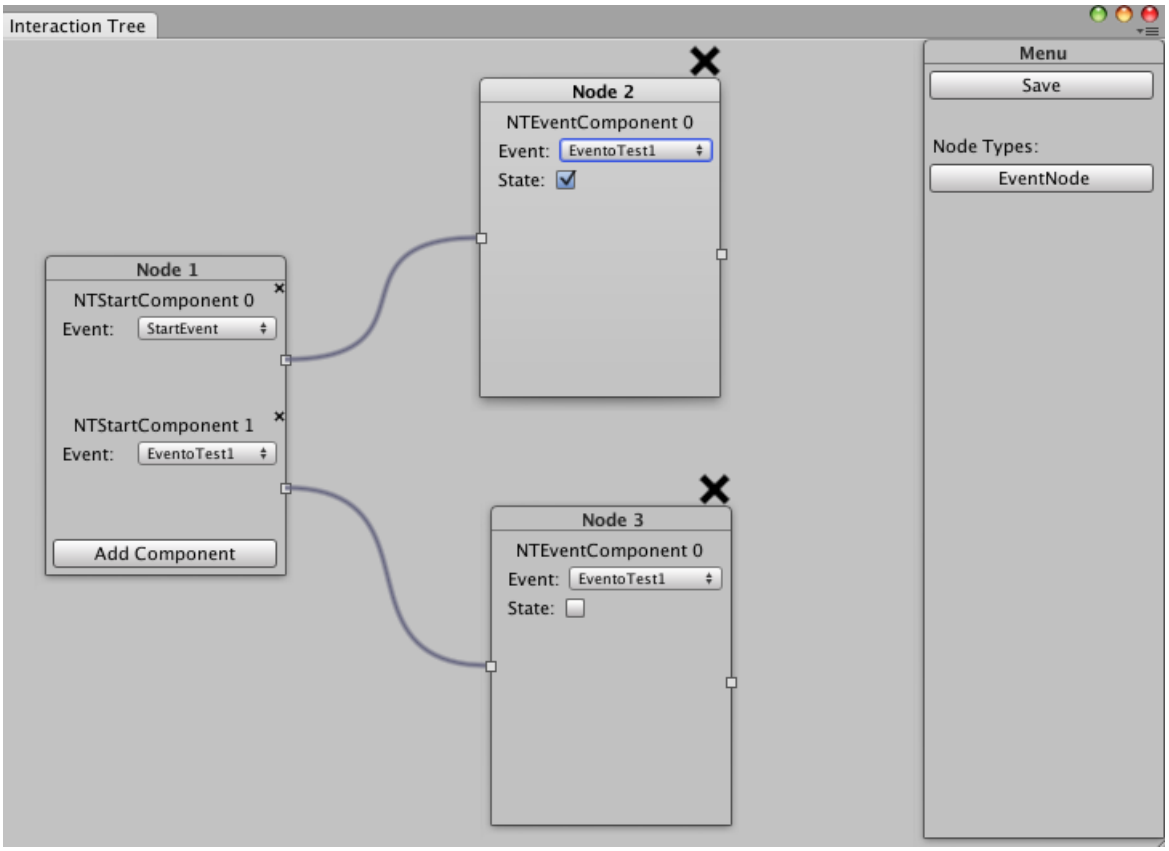

Figura 20 – Editor de *Interaction Object* com a funcionalidade de conexão efetuada com sucesso.

4.4 Edição de Nós de Interação

Como é possível perceber, a edição do *interaction trees* é baseada no conceito de nós, que são chamados de nós de interação. O produto deste trabalho vem com dois nós implementados, que são: nó de início e nó de manipulação de *events*.

O nó de início é um nó único para cada *interaction tree*, isto é, existe apenas um nó de início para cada *interaction tree*. Ele é o único nó que não possui um canal de entrada e pode ser entendido com a raiz ou ponto de partida da *interaction tree*. Além disso, ele é formado por componentes que possuem uma *droplist* com todos os *events* cadastrados no jogo. A Figura 21 representa um nó com dois componentes. Vale ressaltar que não existe um limite máximo para o número de componentes nesse tipo de nó. Caso haja a necessidade de adicionar mais componentes, basta clicar no botão *Add Component*.

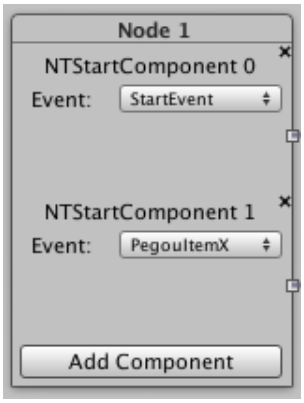

Figura 21 – Representação gráfica do nó de início.

O nó de manipulação de *events* é um nó que possui apenas um componente, conforme apresentado na Figura 22. Nesse componente existe uma *droplist* para selecionar o *event* que se deseja realizar a manipulação e um *checkbox* que indica pra qual estado o *event* vai ser modificado. No caso da Figura 22, o *event* "PegouItemX" está ativado.

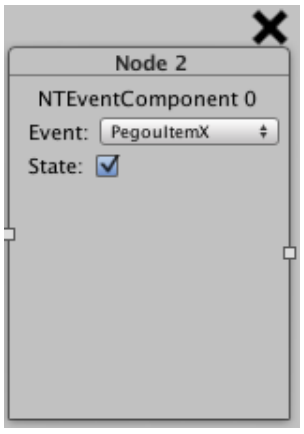

Figura 22 – Representação gráfica do nó de manipulação de evento.

4.5 Editor de *Events*

Ao longo deste capítulo diversas vezes foi citado o uso de *events* nos componentes dos nós. Para que seja possível utilizá-los, antes é preciso cadastrá-los. Para isso, existe o editor de *events*, o qual está localizado na aba *NTeraction>Local Events Editor* (como é mostrado na Figura 23).

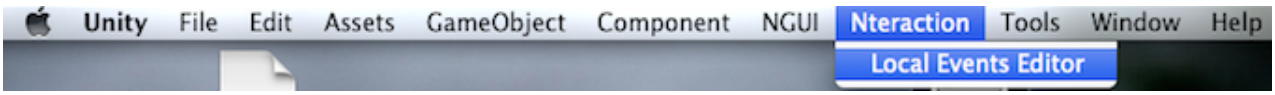

Figura 23 – Aba para abrir o editor de *events*.

Uma vez com o editor aberto, é possível verificar os *events* cadastrados. No caso da Figura 24, existem dois *events*. A cor do *event* significa o seu atual estado no editor do *Unity3D*. A cor for vermelha indica que o *event* está desativado. Enquanto que a cor verde sinaliza que o estado do *event* está ativado. Desta forma, é possível, em tempo de execução do jogo, verificar as mudanças dos *event*.

Uma das principais funcionalidades do editor de *events* é o cadastro de novos *events*, o que pode ser feito através do botão *Create Event*. Além disso, é possível verificar as relações entre *events* por meio do grafo direcionado que é gerado. No caso da Figura 24, o *event* "*StartEvent*" aponta para o *event* "PegouItemX". Isto significa que o *event* "PegouItemX" é manipulado quando o *event* "*StartEvent*" é ativado por meio de um ou mais *interaction objects*.

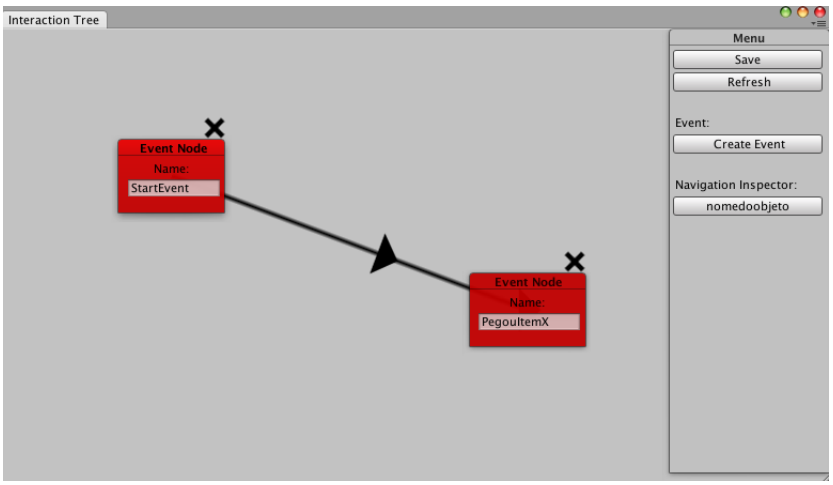

Figura 24 – Editor de *events* aberto.

É possível verificar os *interaction objects* que estão afetando as relações entre os *events*. Para isso, é preciso clicar na seta de navegação entre dois *events*, o que faz com que no menu à direita no campo *Navigation Inspector* sejam listados todos os *interaction objects* que afetam aquela relação. No caso da Figura 24, a única aresta de navegação está

selecionada, e o *interaction object* criado na Figura 15 foi listado. Caso um *interaction object* seja selecionado, o seu editor de *interaction tree* é aberto.

É importante ressaltar que um *event* só pode ser deletado quando não existe nenhuma aresta saindo ou entrando nele. Isto é feito para garantir que não haverá inconsistências nos módulos do produto por uma ação não intencional do *Designer*.

4.6 Estendendo o sistema

Cada jogo possui características próprias quando se trata da forma de contar uma estória. Desta forma, o sistema foi feito de forma a facilitar que novos nós sejam criados por outros programadores a medida que os *Designers* tenham necessidade.

Para que a extensão seja feita, é preciso estender em dois níveis: do componente de nó e do nó.

4.6.1 Estendendo componente de nó

A extensão dos componentes é feita seguindo o *template* apresentando no Código 4.1, no qual é preciso sobrescrever os métodos `DrawContent()`, `XmlPersistDataSave()`, `XmlPersistDataLoad()`, `LoadEventCallBack()`.

No método `DrawContent()` deve ser inserido o código que renderiza os campos que fazem parte do componente. Para exemplificar, se for um componente que representa um diálogo, certamente seria necessário um *text field* para obter a frase que se deseja imprimir para o usuário.

No método `XmlPersistDataSave()` deve ser inserido o código que vai salvar os dados que estão sendo apresentados para o usuário no método `DrawContent()`.

No método `XmlPersistDataLoad()` deve ser inserido o código que faz a leitura dos dados no *XML* e seta as informações que serão disponibilizadas na `DrawContent()`.

O método `LoadEventCallback()` é utilizado somente se dentro do componente é necessário utilizar todos os *events* cadastrados no editor de *events*. Um exemplo são as *droplist* dos nós de início e manipulação de *events*.

Além disso, é preciso criar o construtor fazendo uso do construtor da classe pai *NTComponent*.

Código 4.1 – Template de extensão de um novo componente de nó

```
1 using UnityEngine;
2 using UnityEditor;
3 using System.Collections;
4 using System.Xml;
```

```
5
6  public class NewComponent : NTComponent
7  {
8          public NewComponent(NTNode ownerNode)
9                  : base(ownerNode)
10         {
11                 /*Initialize fields when it's constructed for
12                  *the first time, which means when it's not
13                  *load the data from the XML*/
14         }
15
16         public override void DrawContent (int componentIndex)
17         {
18                 /*Draw the content that should be rendered
19                  *in the component*/
20         }
21
22         public override XmlElement XmlPersistDataSave
23         (XmlFile xmlFile, XmlElement componentRootNode)
24         {
25                 //Save the data on their respective tags
26         }
27
28         public override void XmlPersistDataLoad
29         (XmlElement componentXmlElement)
30         {
31                 //Load data to their respective fields
32         }
33
34         public override void LoadEventCallBack()
35         {
36                 //Callback if you need to deal with the events
37         }
38 }
```

4.6.2 Estendendo um nó

Para estender um nó é preciso criar uma classe que herde de *NTNode* e que o construtor utilize o construtor da classe pai. Dentro deste construtor é necessário adicionar o componente criado para este nó. Um exemplo é apresentado no Código 4.2.

Existem algumas *flags* que pode ser ativadas ou desativadas para aumentar as

funcionalidades do nó. São elas:

- *_isDynamic*: permite ao nó possuir mais de um componente e que componentes possam ser adicionados dinamicamente pelo *Designer*;
- *_closeButton*: caso seja setado para *null*, não é possível deletar o nó;
- *_inputChannel*: caso seja setado para *null*, o nó não possuirá um canal de entrada.

Além disso, é preciso sobrescrever os métodos `Activate()`, `IsReady()` e `GetNextNode()`. Estes métodos são os pontos de comunicação entre o motor de *storytelling* e a mecânica central do jogo.

No método `Activate()` deve ser inserido o código que irá ativar alguma funcionalidade do sistema. Um exemplo pode ser iniciar e popular uma caixa de diálogo.

No método `IsReady()` deve ser inserido o código que indica para a entidade que realiza a travessia de nós em tempo de execução do jogo que já é permitido passar para um próximo nó.

No método `GetNextNode()` deve ser inserido o código que retorna o próximo nó. Além disso, é preciso sobrescrever os métodos `Activate()`, `IsReady()` e `GetNextNode()`. Estes métodos são os pontos de comunicação entre o motor de *storytelling* e a mecânica central do jogo.

Código 4.2 – *Template* de extensão de um novo nó

```
1  using UnityEngine;
2  using System.Collections;
3  using System.Collections.Generic;
4
5
6  public class NewNode : NTNode {
7
8          public NewNode(Rect rectSpacialProperties, string id)
9                  : base(rectSpacialProperties, id, false)
10         {
11                 //if you want to make the node have more
12                 //than one component
13                 //_isDynamic = true;
14
15                 //if you want make impossible to delete this node
16                 //_closeButton = null;
17
18                 //if you don't want to have an input channel for the node
19                 //_inputChannel = null;
```

```
20                  _components.Add(new NewComponent(this));
21          }
22
23          public override void Activate ()
24          {
25                  /*What happens when this
26                  *node is activated in the game*/
27          }
28
29          public override bool IsReady()
30          {
31                  /*boolean that represents that it the traverser
32                   *can move to another node*/
33          }
34
35          public override NTNode GetNextNode()
36          {
37                  /*returns the next node
38                  *that should be activated*/
39          }
40
41  }
```

5 Conclusão

No contexto de desenvolvimento de jogos que fazem forte uso de estórias como parte do entretenimento, o presente trabalho propôs e desenvolveu um motor de *storytelling* cuja arquitetura permite a paralelização das *pipelines* de criação de funcionalidades (feita por programadores) e de geração de conteúdos da estória (feita por *designers*).

Para isso, uma vez definidos as diversas formas de interações que farão parte de como a estória será contada, o programador pode implementar as interfaces dos nós, sem necessariamente implementar as funcionalidades em si. Assim, sem que as funcionalidades estejam totalmente ou praticamente prontas, é possível gerar o conteúdo que fará parte da estória, o que pode ser um interessante fluxo de trabalho para se aplicar em empresas que não dispõem de grandes equipes ou que estejam com fluxos de trabalho morosos ou ineficazes..

Em trabalhos futuros pode-se se trabalhar diversos aspectos que tangem o desenvolvimento de jogos e que não foram abordados ou aprofundados neste trabalho, como:

- Comparativos com métricas bem definidas com relação à eficácia do uso e não uso do *plugin* em ambientes reais de desenvolvimento de jogos;
- Análise das informações obtidas nas árvores de *interaction tree* e do grafo de eventos direcionado que podem auxiliar na tomada de decisão de um *game designer*. Um exemplo disso seria analisar a probabilidade de se chegar em certo evento partindo de um evento em específico.
- Estudos relacionados a um processo de desenvolvimento de jogos que inclua o uso desta ferramenta, uma vez que ela pode auxiliar na paralelização de atividades de diversos membros.

Referências

A.COCKBURN. *Writting Effective Use Cases*. 1. ed. Ann Arbor, Michigan: Addison-Wesley, 2001. Citado na página 36.

A.ERNERST. *Fundamentals of game design*. 2. ed. Berkeley, California: New Riders, 2010. Citado 7 vezes nas páginas 9, 23, 24, 25, 26, 28 e 29.

C.OWEN. Simdialog: A visual game dialog editor. The computing research repository. Nova Iorque, 2008. Citado na página 18.

E.GAMMA et al. *Design Patterns:*: Elements of reusable object-oriented software. Boston, Massachusetts: Addison-Wesley, 1995. Citado 2 vezes nas páginas 30 e 31.

L.SHELDON. *Character Development and Story*. 1. ed. Boston, Massachusetts: Thomson Course, 2004. Citado na página 17.

M.BUCKLAND. *Programming Game AI by Example*. [S.l.]: Wordware, 2007. Citado na página 17.

M.CARBONARO. Interactive story writing in the classroom: Using computer games. International digital games research conference. Vancouver, 2005. Citado na página 17.

M.MCLAUGHLIN; M.KATCHABAW. A reusable scripting engine for automating cinematics and cut-scenes in video games. Proceedings of the 2007 conference on future play. Toronto, 2007. Citado na página 18.

S.BERCZUK; B.APPLETON. *Software Configuration Management Patterns:*: Effective teamwork, practical integration. Boston, Massachusetts: Addison-Wesley, 2003. Citado 2 vezes nas páginas 29 e 30.

S.KIM; S.MOON; S.HAN. Programming the story:: Interactive storytelling system. Informatica: An international Journal of Computing and Informatics. Ljubljana, 2011. Citado na página 23.

Anexos

ANEXO A – Casos de Uso

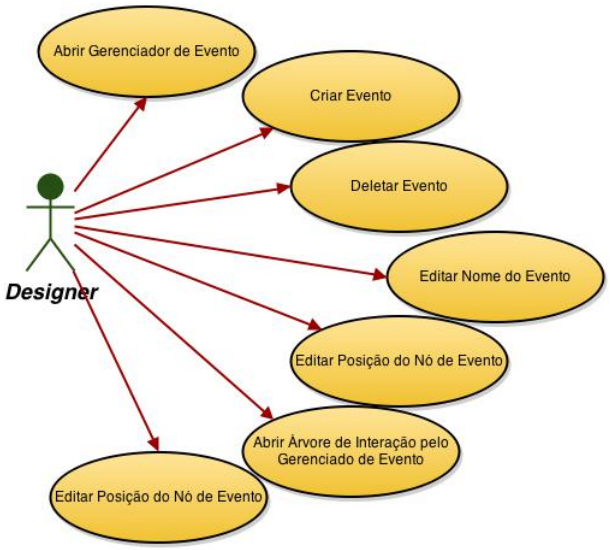

Figura 25 – Diagrama de casos de uso do módulo editor de eventos

Título	Abrir Gerenciador de Eventos
Ator Principal	*Designer*
Estória	O *designer* clica na aba do sistema e escolhe a opção "*Open Event Manager*". Em seguida o sistema carrega todos os eventos, conectando todos aqueles entre si baseado nas árvores de interação.

Tabela 1 – Caso de uso de abrir gerenciador de eventos

Título	Criar Evento
Ator Principal	*Designer*
Estória	Assim que o *designer* abre o gerenciador de eventos, ele clica na opção "*Create Event*". Em seguida um nó de evento é criado na posição zero relativo. Para que o evento seja persistido pelo sistema, o *designer* precisa clicar no botão "*Save*".

Tabela 2 – Caso de uso de criar evento

Título	Deletar Evento
Ator Principal	*Designer*
Estória	O *designer* clica no ícone de deletar do evento alvo. Caso o evento possua arestas com outros eventos, a operação não é efetivada. Caso o evento não possua arestas, a operação é efetuada. Para que a operação seja persistida pelo sistema, o *designer* precisa clicar no botão "*Save*".

Tabela 3 – Caso de uso de deletar evento

Título	Editar Nome do Evento
Ator Principal	*Designer*
Estória	Assim que o *designer* abre o gerenciador de eventos, ele edita o campo "*Name*" do nó de evento. Para que o novo nome do evento seja persistido pelo sistema, o *designer* precisa clicar no botão "*Save*".

Tabela 4 – Caso de uso de editar nome do evento

Título	Editar Posição do Nó de Evento
Ator Principal	*Designer*
Estória	Assim que o *designer* abre o gerenciador de eventos, ele clica em um nó e arrasta para a posição desejada. Para que a posição do evento seja persistido pelo sistema, o *designer* precisa clicar no botão "*Save*".

Tabela 5 – Caso de uso de editar posição do nó de evento

Título	Abrir Árvore de Interação pelo Gerenciador de Evento
Ator Principal	*Designer*
Estória	Com o gerenciador de eventos aberto, o *designer* clica em uma aresta de ligação entre nós de eventos. O sistema carrega todos os objetos de interação relacionados àquela aresta disponibilizando-os. Em seguida o *designer* seleciona um dos objetos de interação e um editor de árvore de interação relacionado ao objeto de interação escolhido é apresentado.

Tabela 6 – Caso de uso de abrir árvore de interaçã pelo gerenciador de evento

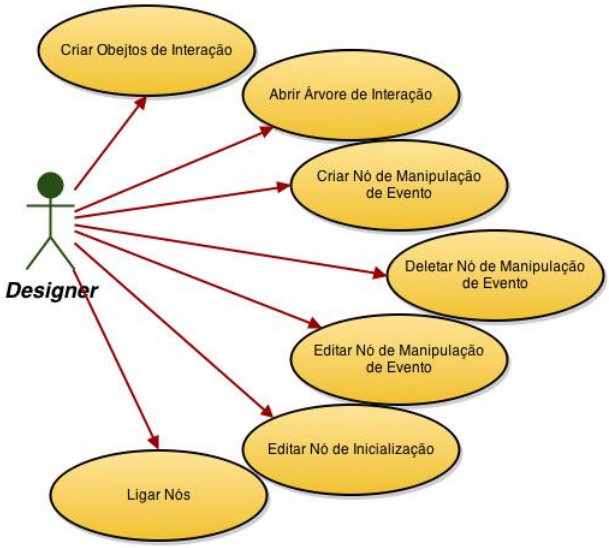

Figura 26 – Diagram de casos de uso do módulo árvore de interação

Título	Criar Objeto de Interação
Ator Principal	*Designer*
Estória	O *designer* realiza a operação de *drag and drop* do script "*NTInteractionObject*" em cima do objeto da cena que se deseja virar um objeto de interação. Em seguida, o *designer* preenche o campo "*Interactable Charater/Object*" do compoenent "*NTInteractionObject*".

Tabela 7 – Caso de uso de criar objeto de interação

Título	Abrir Árvore de Interação
Ator Principal	*Designer*
Estória	O *designer* clica no botão "*Edit Interaction*" no componente "*NTInteractionObject*". Em seguida, o sistema carrega a árvore de interação daquele objeto de interação e apresenta para o *designer*.

Tabela 8 – Caso de uso de abrir árvore de interação

Título	Criar Nó de Manipulação de Evento
Ator Principal	*Designer*
Estória	O *designer* clica no botão "*EventNode*". Em seguida um nó de manipulaço de evento é criado na posição zero relativo. Para que o nó adicionado seja persistido pelo sistema, o *designer* precisa clicar no botão "*Save*".

Tabela 9 – Caso de uso de criar nó de manipulação de evento

Título	Deletar Nó de Manipulação de Evento
Ator Principal	*Designer*
Estória	O *designer* clica no ícone de deletar do nó de manipulação de evento alvo. Para que a operação seja persistida pelo sistema, o *designer* precisa clicar no botão "*Save*".

Tabela 10 – Caso de uso de deletar nó de manipulação de evento

Título	Editar Nó de Manipulação de Evento
Ator Principal	*Designer*
Estória	O *designer* escolhe o nó de manipulação de evento e seleciona o evento que se deseja manipular na *droplist* "*Event*". Além disso, o *designer* escolhe se quer que o evento seja manipulado para ligado ou desligado através do *checkbox* "*State*". Para que os campos editados persistam no sistema, o *designer* precisa clicar no botão "*Save*".

Tabela 11 – Caso de uso de editar nó de manipulação de evento

Título	Editar Nó de Inicialização
Ator Principal	*Designer*
Estória	O *designer* seleciona o nó de início da árvore de interação. Caso o *designer* queira criar mais uma opção de início baseado em evento, ele clica no botão "*Add Component*". Caso o *designer* queira deletar uma opção de início baseado em evento, ele clica no ícone de deletar referente àquela opção. Caso o *designer* queira mudar o evento relacionado à opção de início baseado em evento, ele seleciona o *droplist* "*Event*" da opção desejada. Para que as edições persistam, o *designer* precisa clicar no botão "*Save*".

Tabela 12 – Caso de uso de editar nó de inicialização

Título	Ligar Nós
Ator Principal	*Designer*
Estória	O *designer* pressiona com clique esquerdo sobre o canal de saída do nó e arrasta e solta sobre o canal de entrada do nó que se deseja ligar o nó. Para que a relação entre os nós sejam persistidos pelo sistema, o *designer* precisa clicar no botão "*Save*".

Tabela 13 – Caso de uso de ligar nós