**UNIVERSIDADE REGIONAL DE BLUMENAU**

**CENTRO DE CIÊNCIAS EXATAS E NATURAIS**

**CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO**

# VISEDU-DRONE: MÓDULO DE INTEGRAÇÃO COM ROBOT OPERATING SYSTEM

**JOSÉ GUILHERME VANZ**

**BLUMENAU**
**2015**

**2015/1-19**

**JOSÉ GUILHERME VANZ**

# VISEDU-DRONE: MÓDULO DE INTEGRAÇÃO COM ROBOT OPERATING SYSTEM

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Dalton Solano dos Reis, M. Sc. - Orientador

**BLUMENAU**
**2015**

**2015/1-19**

# VISEDU-DRONE: MÓDULO DE INTEGRAÇÃO COM ROBOT OPERATING SYSTEM

Por

**JOSÉ GUILHERME VANZ**

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

| | |
|---|---|
| Presidente: | Prof. Dalton Solano dos Reis, M. Sc. – Orientador, FURB |
| Membro: | Prof. Aurélio Faustino Hoppe, M. Sc. – FURB |
| Membro: | Prof. Mauro Marcelo Mattos, Doutor – FURB |

Blumenau, 06 de julho de 2015

Dedico este trabalho aos meus pais, familiares e amigos que durante toda a minha vida me apoiaram em minhas conquistas

## AGRADECIMENTOS

Aos meus pais Doraci Vanz e Marina Cestari Vanz que desde criança sempre me apoiaram e incentivaram de ir atrás dos meus sonhos. Agradeço ainda por todo o esforço que realizaram para sempre proporcionarem a melhor educação possível possibilitando que conseguisse alcançar esses sonhos.

A Aline Scheren Bernardi por todo apoio, compreensão, carinho e incentivo que tem me proporcionado nos últimos meses.

Aos meus grandes amigos Deivid Sant'ana, Vander Bertoline e Cleiton Seserino Linhares formados durantes os anos de graduação. Agradeço a eles pelos momentos de companheirismo e risadas que tivemos durante todo esse tempo. Além disso, peço desculpas pelos dias em que tiveram que me aturar de mau humor devido minhas longas madrugadas de estudos.

Agradeço a todos os meus professores da graduação. Em especial ao meu orientador Dalton Solano dos Reis que auxiliou e me ajudou durante todo o desenvolvimento deste trabalho.

A Leandro Wagner artista que criou o modelo 3D utilizado pelo VisEdu-Drone.

A Lemonade que forneceu o AR.Drone Parrot para realizar testes no simulador desenvolvido.

Talk is cheap. Show me the code

Linus Torvalds

# RESUMO

Este trabalho teve como objetivo estender o VisEdu criando um simulador de *drone* integrado com o *framework* para robótica Robot Operating System (ROS). O simulador foi desenvolvido na linguagem Javascript, utilizando a biblioteca Three.js para abstrair o WebGL e facilitar a manipulação do ambiente de realidade virtual. Também foi utilizada a Tween.js para criar animações. Ambas bibliotecas já utilizadas pelo VisEdu. O simulador possibilita a execução das mesmas interações realizadas com o *drone* virtual em um AR.Drone Parrot 2.0 físico. Para controlar o *drone* físico, foi utilizado o *driver* ardrone_autonomy. O *driver* tem como objetivo efetuar a comunicação entre o ROS executado em um computador e o drone. Para possibilitar a comunicação entre o simulador executado no navegador de Internet e o *driver* existente no servidor ROS, foi utilizado um *Websocket* disponibilizado pelo `Rosbridge`. O simulador conseguiu realizar todas as interações do *drone* virtual com um *drone* físico. Porém, foi percebido nos testes de avaliação uma deficiência no sistema de navegação implementado.

Palavras-chave: Drone. Robot Operating System. WebGL. VisEdu.

# ABSTRACT

This work has the goal to extend VisEdu creating a drone simulator integrated with Robot Operating System (ROS) framework. The simulator was written in Javascript programming language and uses the Three.js library to abstracting the WebGL and to make easier the manipulation of virtual reality environment. The Tween.js was also used to create animations. Both libraries already used by VisEdu. The simulator is able to execute the same interactions done by virtual drone in a AR.Drone Parrot 2.0. The ardrone_autonomy driver is used to control the the drone. The driver is responsible to realize the communication between the computer running the ROS server and the drone. The communication between the simulator running in the browser and the driver on the ROS server was able through Websocket made available by `Rosbridge`. The simulator was able to executed all interactions with virtual drone with a real one. However, during validation tests was detected a problem in the navigation system.

Key-words: Drone. Robot Operating System. WebGL. VisEdu.

# LISTA DE FIGURAS

# LISTA DE QUADROS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

API – *Application Programming Interface*

DNS – *Domain Name System*

DOM – *Document Object Model*

FURB – Fundação Universidade Regional de Blumenau

GB – *Gigabytes*

GCG – Grupo de Pesquisa em Computação Gráfica, Processamento de Imagens e Entretenimento Digital

Ghz – *Giga Hertz*

GPU – *Graphics Processor Unit*

HD – High Definition

HTML – *Hyper Text Markup Language*

JOGL – Java OpenGL

MIT – Massachusetts Institute of Technology

OpenGL ES – Open Graphics Library Embedded System

ROS – Robot Operating System

TCP – *Transmission Control Protocol*

TCP/IP – *Transmission Control Protocol/Internet Protocol*

UAV – *Unmanned Aerial Vehicles*

UML – *Unified Modeling Language*

URI – *Uniform Resource Identifier*

WebGL – Web Graphics Library

XML-RPC – *Extensible Markup Language-Remote Procedure Calling*

# SUMÁRIO

# 1 INTRODUÇÃO

A informática tem sido uma aliada nas escolas como elemento complementar no ensino de inúmeras matérias. A sofisticação das tecnologias computacionais tem provocado o aparecimento de meios pedagógicos com base em ambientes gráficos. A realidade virtual é uma tecnologia que tem sido considerada um bom recurso no ensino e aprendizagem por permitir a interação com modelos tridimensionais em uma experiência atraente. Por terem um baixo custo, os sistemas de realidade virtual não imersivos são mais comuns no desenvolvimento de aplicações educativas. Já que as maiorias das escolas possuem equipamentos básicos de informática (CARRIJO, 2007).

Diante disto, simuladores podem ser uma boa alternativa na criação de ambientes de realidade virtual para auxiliar no processo de ensino, já que possuem um baixo custo operacional e fornecem uma experiência atraente para os estudantes. Considerando essa realidade, a Web Graphics Library (WebGL) se torna um aliado valioso. Com ela pode ser criado todo o ambiente virtual necessário para um simulador em um navegador de Internet, tornando o uso desse tipo de tecnologia ainda mais simples, pois não é necessária a instalação de softwares adicionais. Além disso, buscando tornar a experiência proporcionada pela realidade virtual ainda mais atraente, podem ser utilizados equipamentos como robôs. Os quais podem ser programados para executarem as mesmas interações que foram desenvolvidas no simulador, possibilitando desse modo uma experiência ainda mais atrativa para os estudantes.

Diante disto, foi criado um simulador de *drones* integrado com o *framework* de robótica Robot Operating System (ROS). O simulador permite a interação com um *drone* virtual movimentando-o por um espaço 3D. Também é permitido executar as mesmas instruções de movimentação do *drone* virtual em um AR.Drone Parrot 2.0 físico. O simulador foi implementado na linguagem Javascript e trata-se de uma extensão do trabalho desenvolvido por Nunes (2014). O trabalho de Nunes (2014) foi utilizado como base para implementação reutilizando a *Application Program Interface* (API) da WebGL abstraída pela biblioteca Three.js que já é utilizada. Para possibilitar a comunicação entre o simulador executado no navegador de Internet e o ROS foi utilizado o `Rosbridge`. Software existente na plataforma ROS que permitiu a comunicação com o aparelho físico através de *websocket*.

## 1.1 OBJETIVOS

Este trabalho se propõe integrar o Visualizador de Material Educacional (VisEdu) com o Robot Operating Systems (ROS) permitindo a interação com um *drone* AR.Drone Parrot 2.0.

Os objetivos específicos são:

a) criar um nova peça que representa um *drone*;
b) criar uma nova peça que habilite animações;
c) alterar peças de translação e rotação no VisEdu para que permitam a movimentação do *drone* virtual e físico;
d) criar um ambiente 3D onde o usuário poderá visualizar a execução de animações do *drone* virtual;
e) enviar sequências de instruções para movimentar um *drone* físico.

## 1.2 ESTRUTURA

Este trabalho está organizado em quatro capítulos. O primeiro capítulo contém a introdução ao tema abordado, objetivos e estrutura do trabalho.

No segundo capítulo é apresentada a fundamentação teórica necessária para a compreensão dos temas abordados durante o desenvolvimento.

O terceiro capítulo contém as etapas do desenvolvimento do trabalho. Ao final são discutidos os resultados obtidos durante todo o processo de implementação.

No quarto capítulo são expostas as conclusões do trabalho, discutindo os resultados obtidos e sugerindo extensões para trabalhos futuros.

# 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo está organizado em quatro seções. Na seção 2.1 é apresentado o conceito de *drones* e como podem ser utilizados. Na seção 2.2 são apresentadas as principais tecnologias utilizadas no desenvolvimento da aplicação: *Hyper Text Markup Language 5* (HTML5), WebGL e Javascript. A seção 2.3 explora o Robot Operating System (ROS), *framework* utilizado para efetuar a comunicação com *drone*. Na seção 2.4 é apresentado o *framework* base utilizado na implementação, o Visualizador de Material Educacional (VisEdu). Finalmente, na seção 2.5, são apresentados os trabalhos correlatos à aplicação desenvolvida.

## 2.1 DRONE

Os *Unmanned Aerial Vehicles* (UAV), também conhecidos como *drones*, são essencialmente aeronaves controladas por pilotos em terra ou por softwares em computadores ou sistemas embarcados. Isso significa que *drones* podem ser programados para realizar tarefas sem a interferência humana. Isso permite várias aplicações onde o uso de aeronaves convencionais poderia aumentar o custo e o risco para a integridade física da tripulação (UNMANNED AERIAL VEHICLE SYSTEMS ASSOCIATION, 2015).

Entre as várias aplicações dos *drones* pode-se destacar o uso para busca e salvamento, estudo da atmosfera, monitoramento de estradas, pesquisas sobre a vida selvagem e entretenimento. Também é utilizado por forças armadas em missões de reconhecimento, ataque e espionagem (LIVE SCIENCE, 2013). Algumas das principais vantagens que permite todas essas aplicações são o tempo de voo, o baixo custo e o fato de não oferecer risco a tripulação (UNMANNED AERIAL VEHICLE SYSTEMS ASSOCIATION, 2015).

Entre os *drones* comerciais voltados ao público geral existe o quadcóptero AR.Drone 2.0 desenvolvido pela empresa Parrot (figura 1). Ele possui uma câmera *High Definition* (HD) que possibilita filmagens e captura de fotos, uma série de sensores como acelerômetros, giroscópios, sensor de pressão, sensor ultrassônico, entre outros. Pode ser utilizado em voos em locais abertos ou fechados, sendo controlado por um *smartphone* (PARROT, 2014). O AR.Drone Parrot possibilita ainda que sejam escritos softwares que rodem em seu sistema embarcado possibilitando voos autônomos (EDX, 2014).

Figura 1 – AR.Drone Parrot 2.0

Fonte: Parrot (2014).

## 2.2 HTML 5, WEBGL E JAVASCRIPT

Para desenvolver o trabalho proposto são utilizadas 3 tecnologias principais, *o* HTML5, WebGL e Javascript. O HTML5 é o último padrão para o HTML, e nele são disponibilizadas novas APIs com elementos de semântica, gráficos, multimídia, entre outros (W3C SCHOOLS, 2014b). O HTML5 possui alguns objetivos em sua especificação, entre eles pode-se destacar: evitar expor os usuários da tecnologia a complexidade de *multithreading*, interagir com outras especificações e ser extensível (W3C, 2015). O principal elemento disponibilizado nesta nova especificação do HTML utilizado neste trabalho é o *canvas*. Ele consiste em um contêiner onde, em tempo de execução, podem ser desenhados gráficos em uma página *web* (W3C SCHOOLS, 2015).

A API utilizada na criação do espaço 3D através do *canvas* é o WebGL. Trata-se de uma API de renderização 3D desenvolvida para *web* (KHRONOS GROUP, 2014) compatível com qualquer *browser* e sem a necessidade de instalação de *plug-ins*. O WebGL é derivado do Open Graphics Library Embedded Systems 2.0 (OpenGL ES 2.0), e prove funcionalidade de renderização semelhante mas no contexto HTML. Tendo em mente os vários casos de uso existentes para gráficos 3D, o WebGL escolhe uma abordagem flexível que permite sua aplicação em qualquer situação. O que permite que outras bibliotecas especificas para cada necessidade sejam construídas sobre ele (KHRONOS GROUP, 2014). Algumas de suas principais vantagens são: ser uma API baseada em um padrão de renderização de gráficos 3D vastamente aceito, possuir compatibilidade com diversos navegadores de Internet, multiplataforma, ter aceleração 3D de hardware no navegador e possuir boa integração com o conteúdo HTML (KHRONOS GROUP, 2011).

Para a manipulação desse espaço 3D e de toda a aplicação é utilizado o Javascript. Que é uma linguagem de programação *script* orientada a objetos, pequena, leve e multiplataforma

(MOZILLA FOUNDATION, 2014a). Atualmente, todos os navegadores modernos existentes em diversas plataformas de *desktop*, *tablets* e *smartphones* incluem um interpretador Javascript. Além disso, é usada na maioria dos *websites* para criar, apagar, alterar e copiar elementos *Document Object Model* (*DOM)* (W3C SCHOOLS, 2014a).

## 2.3 ROBOT OPERATING SYSTEM

O Robot Operating System (ROS) é um meta sistema operacional *open source* para robôs. Ele provê recursos de um sistema operacional como abstração de hardware, controle de dispositivos em baixo nível, passagem de mensagens entre processos e gerenciamento de pacotes. Além disso, são disponibilizadas ferramentas e bibliotecas para obter, escrever, compilar e rodar código através de múltiplos computadores (OPEN SOURCE ROBOTICS FOUNDATION, 2014b).

O principal objetivo do *framework* é permitir a reutilização de código em pesquisas e desenvolvimento de softwares para robótica. Isso é possível graças ao fato de que um software desenvolvido em ROS é divido em pequenas partes, chamados de nós. Isso permite que cada nó do sistema seja desenvolvido individualmente, com baixo acoplamento e sem afetar diretamente o resto do sistema (OPEN SOURCE ROBOTICS FOUNDATION, 2014b). Na seção 2.3.1 são apresentados os principais conceitos necessários para entender o funcionamento de aplicações escritas usando o *framework*.

### 2.3.1 Conceitos

O ROS é separado em três níveis conceituais. Que são o sistema de arquivos, o grafo computacional e a comunidade (OPEN SOURCE ROBOTICS FOUNDATION, 2014a). Os níveis mais importantes para o desenvolvimento deste trabalho, o sistema de arquivo e o grafo computacional, serão mais explorados nas subseções 2.3.1.1, 2.3.1.2. Na seção 2.3.1.3 é apresentado outro conceito importante para entender o *framework*, os nomes.

O terceiro nível conceitual responde aos recursos existentes para facilitar a troca de software e conhecimento entre diferentes comunidades e usuários. Entre esses recursos podem ser citados as distribuições do ROS, que facilitam a instalação e mantêm versões compatíveis de diferentes softwares. Os repositórios onde diversas instituições podem desenvolver e disponibilizar seus próprios componentes para serem utilizados com o ROS. Há ainda a *wiki*, lista de e-mails e fórum onde usuários podem consultar documentação e trocar experiências (OPEN SOURCE ROBOTICS FOUNDATION, 2014a).

#### 2.3.1.1 Sistema de arquivos

Este nível conceitual cobre os principais recursos utilizados pelo ROS. Entre eles, podem ser citados os *packages*, *message types* e *service types*. Softwares escritos para o ROS são organizados em *packages*. Eles são a principal unidade de organização e seu objetivo é prover uma funcionalidade de uma maneira fácil de ser utilizada e distribuída. Um pacote pode conter nós ROS, dados, arquivos de configuração, software de terceiros e tudo o que for útil para a utilização do mesmo (OPEN SOURCE ROBOTICS FOUNDATION, 2014c).

Os *message types* são arquivos de extensão `msg`, que descrevem a estrutura dos dados enviadas nas mensagens publicadas nos tópicos dentro do ROS. Mensagens, tópicos e serviços serão melhores descritos na seção 2.3.1.2. Cada mensagem pode conter campos de tipos primitivos, *arrays* de tipos primitivos e outras estruturas de dados aninhadas (OPEN SOURCE ROBOTICS FOUNDATION, 2012a). Da mesma forma, similar aos arquivos que descrevem os dados para as mensagens, os arquivos de extensão `srv` descrevem *services types.* Esses são arquivos que, utilizando os mesmos tipos de dados existentes nos *messages types*, descrevem os campos que são enviados e recebidos de um serviço ROS (OPEN SOURCE ROBOTICS FOUNDATION, 2012b).

#### 2.3.1.2 Grafo computacional

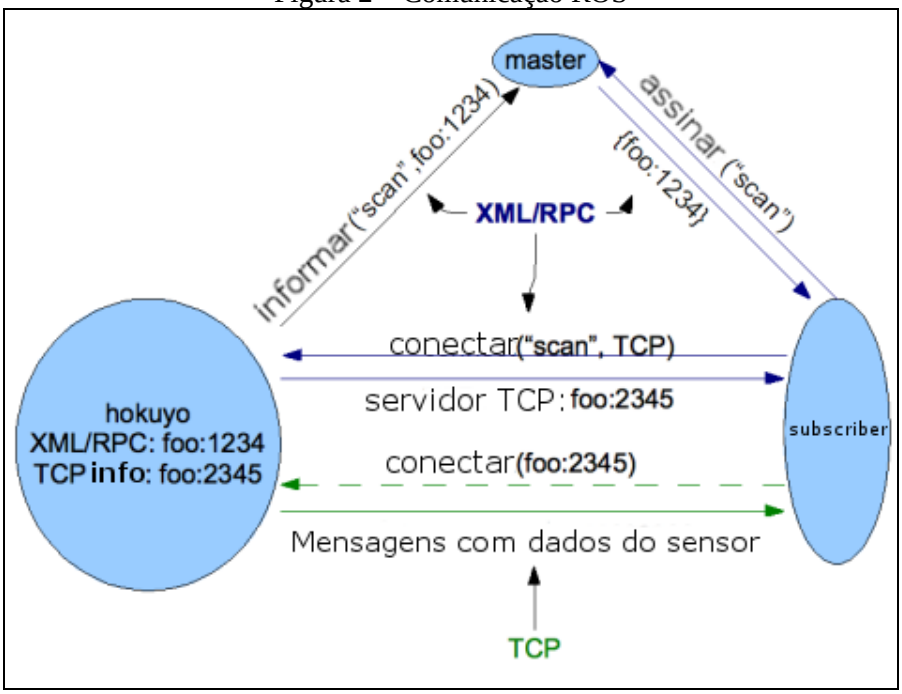
Um grafo no ROS é uma rede *peer-to-peer* de processos trabalhando juntos para processar dados. Os principais elementos nesse grafo são os nós, o mestre, servidor de parâmetros, mensagens, serviços, tópicos e *bags* (OPEN SOURCE ROBOTICS FOUNDATION, 2014a). Cada nó do grafo é um processo que realiza algum tipo de processamento. Por exemplo, um nó pode ser responsável por controlar os motores, outro planejar o caminho executado pelo robô ou controlar um sensor (OPEN SOURCE ROBOTICS FOUNDATION, 2014a). Para que cada nó consiga encontrar os demais nós, trocar mensagens e chamar serviços existe o mestre. Ele é similar a um servidor de *Domain Name System* (DNS), onde é disponibilizado o registro e busca de nomes para todo o grafo. Esse recurso é o que permite a comunicação entre nós (OPEN SOURCE ROBOTICS FOUNDATION, 2014a). É no mestre que são registrados os tópicos e todas as informações de *publishers* e *subscribers*, bem como o nó que provê algum tipo de serviço. Além disso, é ele que também fornece o servidor de parâmetros, onde dados podem ser armazenados em um único local e organizados em uma estrutura de chaves e valores (OPEN SOURCE ROBOTICS FOUNDATION, 2012c).

A comunicação entre os nós do grafo ocorre através de trocas de mensagens. Essas mensagens são basicamente estruturas de dados enviadas de um nó do grafo para outro. A troca de mensagens pode ocorrer de duas formas: tópicos e serviços (OPEN SOURCE ROBOTICS FOUNDATION, 2014a).

Tópicos são meio de comunicação unidirecionais onde os nós podem trocar mensagens (OPEN SOURCE ROBOTICS FOUNDATION, 2014e). Nós registrados como *publishers* podem publicar mensagens no tópico que são consumidas por outros nós registrados como *subscribers*. Quando um nó se registra com a intenção de consumir mensagens de um tópico, conexões diretas com os *publishers* são estabelecidas (OPEN SOURCE ROBOTICS FOUNDATION, 2014a). Para que essa comunicação ocorra é necessário que cada nó possua um servidor Extensible Markup Language-Remote Procedure Calling (XML-RPC). É através desse servidor que um nó consegue estabelecer a comunicação inicial com o mestre e com os demais nós do grafo (OPEN SOURCE ROBOTICS FOUNDATION, 2014d).

Quando um nó pretende se tornar *publisher* de um tópico, antes de mais nada, é necessário que o mestre seja avisado da sua intenção. Para isso, é enviada uma mensagem via XML-RPC ao mestre contendo informações sobre qual tópico se deseja publicar mensagens e a *Uniform Resource Identifier* (URI) de conexão do servidor XML-RPC existente no mesmo. Essa informação é armazenada em uma tabela onde estão registrados todos os *publisher*es. Quando outro nó desejar se registrar como *subscriber* o mestre retorna os registros com as URIs de conexão para todos os *publisheres* do tópico solicitado. O *subscriber* então envia uma mensagem também em XML-RPC para todos os *publishers* requisitando uma conexão. Como resposta, é enviado uma nova URI de conexão para o servidor *Transmission Control Protocol* (TCP) que estiver rodando no *publisher*. Esse servidor é por onde as mensagens publicadas no tópico serão enviadas (OPEN SOURCE ROBOTICS FOUNDATION, 2014d). O fluxo descrito pode ser visto na figura 2.

Figura 2 – Comunicação ROS




Fonte: adaptado de Open Source Robotics Foundation (2014).

Outro fator importante que permite que o ROS seja bastante flexível é que *subcribers* e *publishers* são independentes, o que permite que todos os nós possam ser iniciados, parados e reiniciados sem afetar o resto do grafo (OPEN SOURCE ROBOTICS FOUNDATION, 2014a).

Outra forma de realizar a troca de dados entre nós são os serviços. Serviços são semelhantes às chamadas de métodos remotos (OPEN SOURCE ROBOTICS FOUNDATION, 2012b), de modo que, quando um cliente invoca determinado serviço uma mensagem de requisição é enviada ao nó que o provê. Após realizar o processamento necessário, outra mensagem de resposta é enviada ao nó que fez a requisição original (OPEN SOURCE ROBOTICS FOUNDATION, 2014d).

Inicialmente, como na comunicação por tópicos, o nó que desejar ser provedor de algum tipo de serviço deve se registrar no mestre. Uma vez registrado, quando um nó procurar por um serviço o mestre irá retornar os dados necessários para estabelecer uma conexão *Transmission Control Protocol/Internet Protocol* (TCP/IP) com o nó provedor do serviço. Depois que a conexão com o nó for estabelecida é efetuada a troca do *header* da conexão, uma estrutura onde estão contidas informações úteis sobre a conexão, como tipo de dados e roteamento (OPEN SOURCE ROBOTICS FOUNDATION, 2011). Depois de concluída a troca do *header*, o cliente poderá mandar uma mensagem de requisição, que depois de processada, será respondida com outra mensagem com o resultado da requisição (OPEN SOURCE ROBOTICS FOUNDATION, 2014d).

#### 2.3.1.3 Nomes

Outro conceito importante para a construção de sistemas complexos em ROS são os nomes. Eles são separados em dois grupos principais, os nomes para os recursos do grafo e nomes para recursos de pacotes. No primeiro grupo pode-se encontrar os nomes para recursos como nós, parâmetros, tópicos e serviços, sendo que seus nomes são divididos em quatro tipos: nomes base; nomes relativos; nomes globais e privados (OPEN SOURCE ROBOTICS FOUNDATION, 2014a).

Nomes de recursos são encapsulados em *namespaces*, os quais podem ser compartilhados com outros recursos. De modo geral, nomes são resolvidos relativamente ao *namespace* atual do nó. Assim, se um nó possui o nome `/foo/no1` ele está no *namespace* `foo`. Quando buscado pelo nome `no2`, será resolvido para `/foo/no2`. Nomes base são como subclasses de nomes relativos e possuem as mesmas regras de resolução. São geralmente utilizados para inicializar nomes de nós (OPEN SOURCE ROBOTICS FOUNDATION, 2014a).

Quando utilizado o símbolo tilde (~) na frente de um nome significa que se trata de um nome privado. Quando usada essa nomenclatura o nome de um nó se torna um *namespace*. Por exemplo, o nó de nome `node1` existente no *namespace* `/bar/` possui um *namespace* privado `/bar/node1`, de modo que, quando referenciado o recurso `~resource` o nome é resolvido para `/bar/node1/resource`. Enquanto nomes privados são relativos ao *namespace* do próprio nó, nomes globais são considerados totalmente resolvidos. Portanto, não são relativos a nenhum *namespace*. São nomes que começam com / e sua utilização deve ser evitada para não causar problemas de compatibilidade (OPEN SOURCE ROBOTICS FOUNDATION, 2014a). No quadro 1 é apresentado alguns exemplos de como são resolvidos os tipos de nomes existentes (OPEN SOURCE ROBOTICS FOUNDATION, 2014a).

Quadro 1 – Resolução de nomes em ROS

| nó | relativo | global | privado |
|---|---|---|---|
| /node1 | bar -> /bar | /bar -> /bar | ~bar -> /node1/bar |
| /wg/node2 | bar -> /wg/bar | /bar -> /bar | ~bar -> /wg/node2/bar |
| /wg/node3 | foo/bar -> /wg/foo/bar | /foo/bar -> /foo/bar | ~foo/bar -> /wg/node3/foo/bar |

Fonte: Adaptado de Open Source Robotics Foundation (2014b).

Existem duas regras para que nomes de recursos em um grafo sejam válidos:

a) primeiro caractere dever ser alfanumérico, começar com barras (/) ou com tilde (~);
b) os demais caracteres devem ser alfanuméricos, *underscore* (_) ou barras (/).

A única exceção a essas regras são os nomes bases, onde não pode existir barras ou tildes (OPEN SOURCE ROBOTICS FOUNDATION, 2014a).

Enquanto os nomes para recursos no grafo são utilizados para nomear nós, tópicos e serviços, os nomes para recursos de pacote são utilizados para simplificar o acesso a arquivos e tipos de dados em disco, no nível conceitual de sistema de arquivos existente no ROS. O nome possui uma estrutura que é formada pelo pacote no qual o recurso é encontrado seguido pelo nome do recurso. É assumido que os pacotes possuem uma estrutura de diretórios pré-definida onde os recursos podem ser encontrados. Dessa forma, o recurso `std_msgs/String` é uma forma reduzida para o caminho completo em disco até o arquivo, de extensão `msg`, que define a mensagem `String` (OPEN SOURCE ROBOTICS FOUNDATION, 2014a).

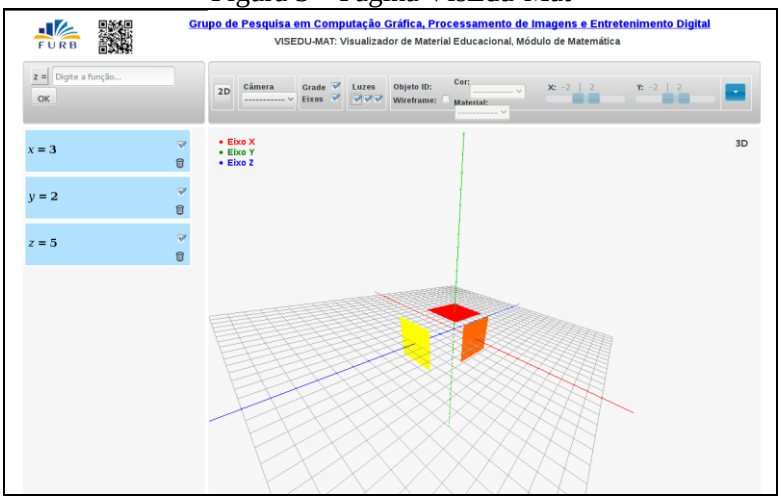
## 2.4 VISEDU – VISUALIZADOR DE MATERIAL EDUCACIONAL

O VisEdu, é um projeto que faz parte de linha de pesquisa do Grupo de Pesquisa em Computação Gráfica, Processamento de Imagens e Entretenimento Digital (GCG) da Fundação Universidade Regional de Blumenau (FURB). Esse projeto tem como objetivo desenvolver tecnologias para facilitar a disponibilização de material educacional interdisciplinar auxiliando educadores no ensino de seus objetos de estudo. O projeto utiliza objetos de aprendizagem para facilitar a decomposição em módulos menores e potencialmente reutilizáveis (GRUPO DE PESQUISA EM COMPUTAÇÃO GRÁFICA, PROCESSAMENTO DE IMAGENS E ENTRETERIMENTO DIGITAL, 2014). Cada módulo desenvolvido pode utilizar diversos tipos de tecnologias disponíveis no mercado, em sua maioria tecnologias *web*.

Entre os módulos já desenvolvidos, existe o VisEdu-Mat que consiste em uma aplicação *web* que tem como principal objetivo a criação, validação e exibição de funções matemáticas, permitindo a visualização de funções em um ambiente tridimensional (3D), onde as funções podem ter as variáveis `x`, `y` e `z`. Possibilita também a visualização em um ambiente bidimensional (2D) sendo que as funções podem possuir somente as variáveis `x` e `y` (KRAUSS, 2013, p. 7). Na página *web* da aplicação (figura 3), o usuário tem a possibilidade de informar os valores para as variáveis `x`, `y` e `z` da função que deseja visualizar. Conforme os valores informados pelo usuário para cada variável e o tipo de visualização desejada, podendo ser 2D ou 3D, a aplicação após validar a função matemática informada pelo usuário, mostra em uma área gráfica a representação gráfica correspondente. Além disso, existem controles adicionais como posicionamento da câmera, sendo possível selecionar uma das câmeras fornecidas pela aplicação e com a utilização do mouse interagir com ela, aproximando e

movimentando-se pela cena. Existem ainda configurações para habilitar e desabilitar luzes, exibir os eixos e grade entre eixos na área gráfica onde são exibidas as funções.

Figura 3 – Página VisEdu-Mat

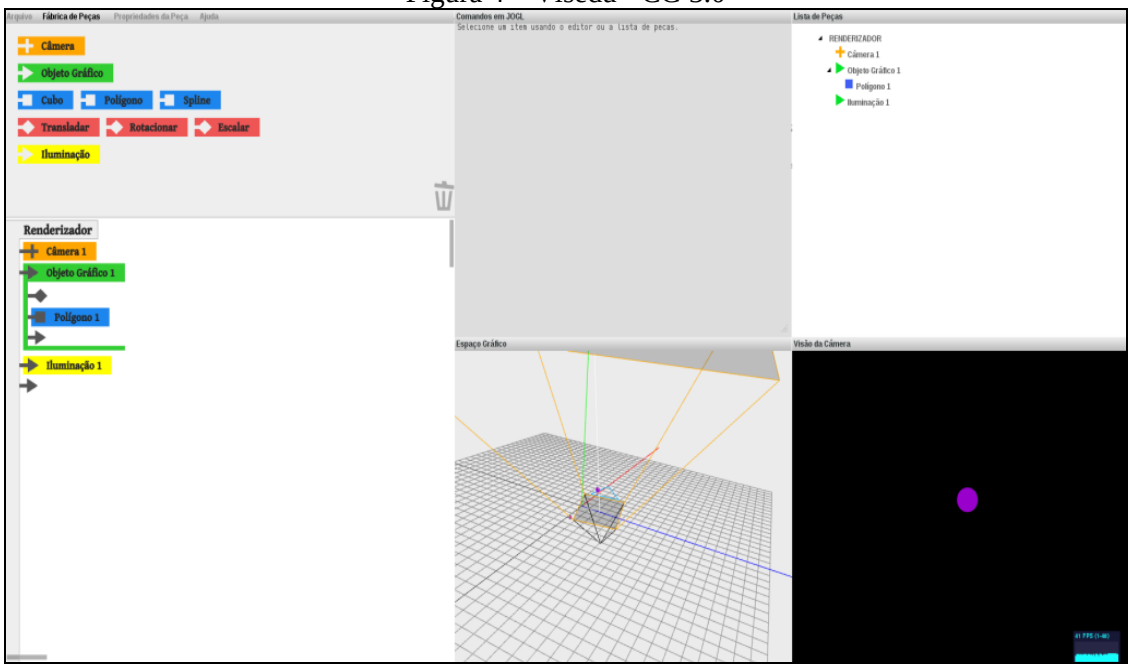



Fonte: Krauss (2013).

Um segundo módulo já implementado é o VisEdu-CG 3.0 que é um trabalho que dá continuidade ao VisEdu-CG (MONTIBELLER, 2014). O trabalho tem o intuito de disponibilizar uma aplicação *web* para auxiliar o ensino de computação gráfica, possuindo alguns conceitos básicos sobre o assunto. Nela é disponibilizada uma funcionalidade de encaixar peças, onde cada peça disponível trabalha com um conceito diferente existente na computação gráfica como câmera, objeto gráfico e transformações geométricas. Conforme as peças vão sendo encaixadas, é possível visualizar o resultado gráfico e o código fonte em Java utilizando a API Java OpenGL (JOGL) necessário para gerar o resultado correspondente. Que permite o usuário ter uma melhor compreensão da ordem lógica dos comandos necessários para o desenvolvimento da cena exibida (MONTIBELLER, 2014, p. 7).

Utilizando como base o trabalho de Montibeller (2014), o VisEdu-CG 3.0 (NUNES, 2014) adiciona novas funcionalidades na aplicação que exploram outros conceitos de computação gráfica 2D e 3D. Como pode ser visto, na figura 4, foram incluídos conceitos como iluminação da cena, novos objetos gráficos (polígono e *spline*), bem como a seleção de objetos gráficos a partir da área gráfica da aplicação. Além disso, foram disponibilizadas as mesmas funcionalidades gráficas em 3D em um espaço 2D (NUNES, 2014, p. 7). No anexo A pode ser encontrada uma explicação mais detalhada sobre cada componente do VisEdu-CG 3.0.

Figura 4 – Visedu - CG 3.0




Fonte: Nunes (2014).

## 2.5 TRABALHOS CORRELATOS

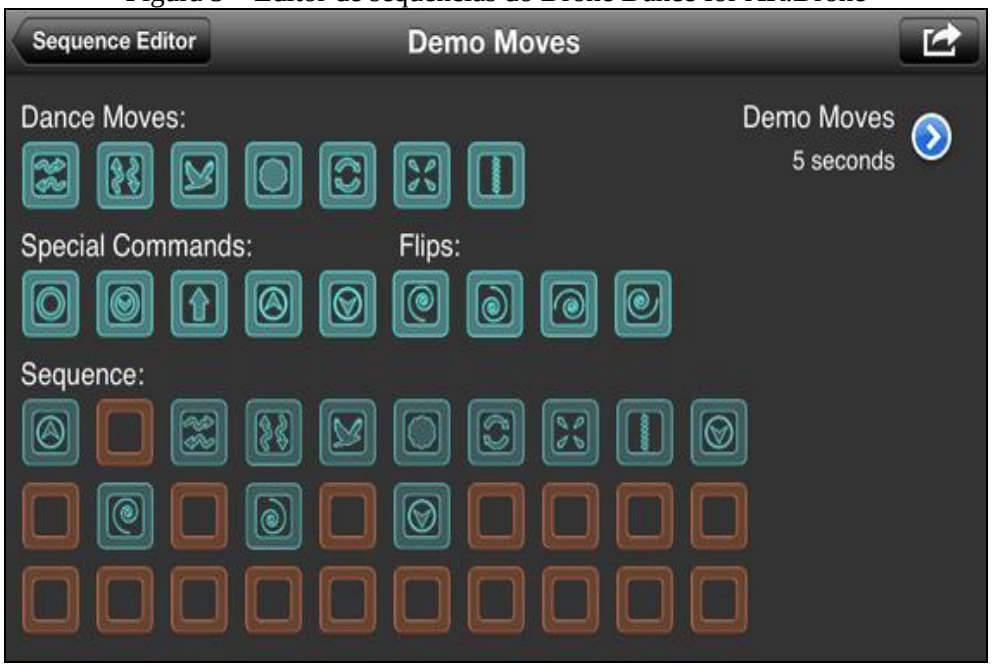
Entre os trabalhos pesquisados foram selecionados alguns que tenham alguma relação com as tecnologias utilizadas ou pela linha de pesquisa do trabalho desenvolvido. Os selecionados foram: Drone Dance for AR.Drone (KAMMERER, 2013); Scratch (LIFELONG KINDERGARTEN GROUP, 2014); e RC Drone – Quadcopter (POLYESTERGAMES PTY. LTD., 2013).

### 2.5.1 Drone Dance for AR.Drone

O Drone Dance for AR.Drone consiste em um aplicativo desenvolvido por Kammerer (2013) voltado para dispositivos com sistema operacional iOS na versão 4.3 ou superior. O aplicativo permite que o usuário crie sequências de movimentos que são executados por um AR.Drone Parrot conectado com o dispositivo através de uma rede *wi-fi*. Como pode ser visto na figura 5, o aplicativo disponibiliza um editor para a criação das sequências de movimentos. Nesse editor existe uma série de *slots* onde o usuário informa a sequência de movimentos desejado. Para isso, basta arrastar um dos movimentos pré-definidos existentes em categorias como movimentos de dança, comandos especiais ou *flips* para dentro de um *slot* disponível. Uma vez criada, a sequência de movimentos pode ser executada por um ou dois *drones*. Para a execução simultânea entre dois *drones*, é necessária a utilização de dois dispositivos pareados entre si via Bluetooth, onde cada um deles está conectado a um dos *drones* pela rede *wi-fi* (KAMMERER, 2013). Para uma melhor precisão nos movimentos executados é

aconselhado que seja colocado um marcador abaixo do *drone*. Quando identificado esse marcador, o *drone* tenta manter sua posição sobre este marcador (KAMMERER, 2013).

Figura 5 – Editor de sequências do Drone Dance for AR.Drone

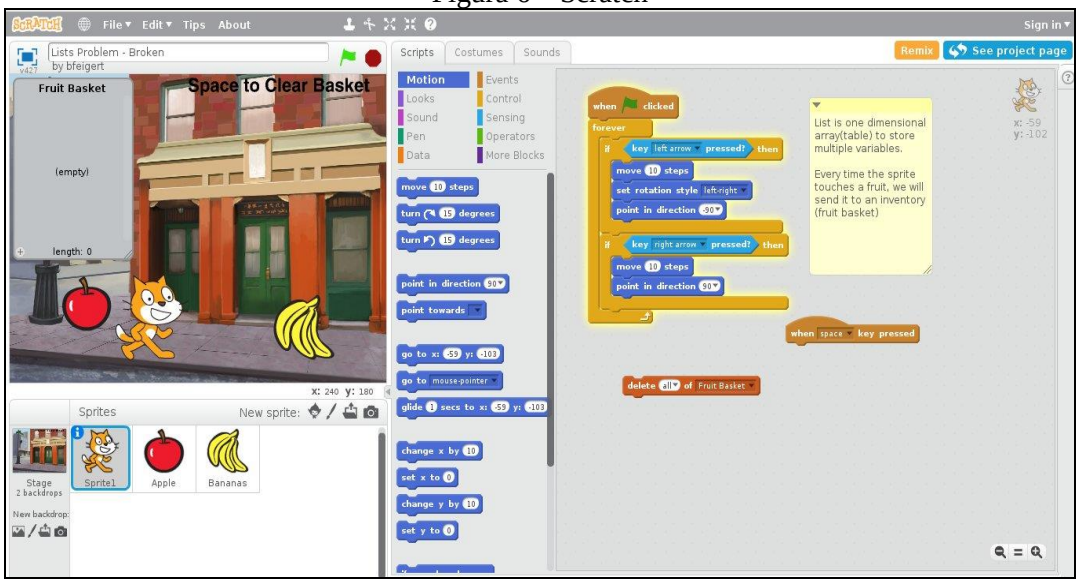



Fonte: Kammerer (2013).

### 2.5.2 Scratch

Scratch é uma aplicação desenvolvida pelo Lifelong Kindergarten Group do Massachusetts Institute of Technology (MIT) que tem como objetivo ajudar principalmente o público mais jovem a ter criatividade, pensar sistematicamente e trabalhar colaborativamente. Essa ferramenta permite a criação de animações, jogos, músicas e histórias interativas através de um ambiente de programação visual, ou seja, não é necessário que o usuário escreva nenhuma linha de código para criar seus programas. Ao invés disso como pode ser visto na figura 6, o Stratch faz analogia a um jogo de encaixe de peças, onde o usuário pode criar toda a lógica de execução arrastando peças e acoplando elas entre si. Cada peça é classificada em categorias onde são separadas de acordo com o tipo de ação que executam dentro da lógica criada. Podem ser utilizadas peças que tem funções de controle de fluxo e repetição, enquanto outras representam eventos, sons, operações matemáticas, variáveis, entre outros. A aplicação permite ainda que ao finalizar seus projetos os usuários possam compartilhá-los na Internet. Permitindo que outros usuários possam utilizar e visualizar como foi feito (LIFELONG KINDERGARTEN GROUP, 2014).

Figura 6 – Scratch




Fonte: Lifelong Kindergarten Group (2014).

Um grande número de educadores, pesquisadores e pais vem utilizando esta ferramenta de forma formal ou informal em seus ambientes de aprendizagem. Devido ao crescente número de usuário do Scratch, foi lançado em 2009 o ScratchEd. Consiste em uma comunidade *on-line* onde educadores que utilizam a ferramenta podem compartilhar histórias, trocar recursos e tirar dúvidas (LIFELONG KINDERGARTEN GROUP, 2015).

### 2.5.3 RC Drone – Quadcopter

O RC Drone - Quadcopter é um jogo disponível para iPhone, iPad e iPod que estejam rodando o iOS nas versões 6.0 ou superior lançado em outubro de 2013 e desenvolvido pela Polyestergames Pty. LTD. Ele consiste em simulador de quadcóptero em um ambiente virtual 3D, como pode ser visto na figura 7. Nele o usuário tem o controle sobre a potência dos motores, rotação e direção do *drone* virtual, podendo movimentá-lo por todo o cenário através de toques na tela do dispositivo utilizado. Possui também várias câmeras com diferentes ângulos de visualização e detecção de colisão entre o *drone* operado pelo usuário com os objetos existente na cena como árvores, pedras e construções.

Figura 7 – Imagem do aplicativo RC Drone - Quadcopter


Fonte: Polyestergames Pty. LTD (2014).

# 3 DESENVOLVIMENTO

Neste capítulo é descrito o processo de desenvolvimento do VisEdu-Drone. Na seção 3.1 serão descritos os principais requisitos da aplicação desenvolvida, a seção 3.2 apresenta a especificação e a última seção apresenta detalhes de implementação.

## 3.1 REQUISITOS PRINCIPAIS DO PROBLEMA

O simulador deverá:

a) fornecer uma interface gráfica de interação com o usuário (Requisito Funcional – RF);
b) fornecer componentes de movimentação, rotação e altitude do *drone* virtual (RF);
c) fornecer um visualizador do ambiente 3D (RF);
d) permitir a execução dos componentes de movimentação no ambiente virtual (RF);
e) permitir o envio de comandos que possam ser executados em um *drone* Parrot AR.Drone 2.0 físico (RF);
f) implementar o simulador utilizando Javascript, HTML5 e WebGL (Requisito Não Funcional – RNF);
g) utilizar o VisEdu como *framework* base para a implementação (RNF);
h) utilizar o *framework* ROS para realizar a comunicação com o AR.Drone Parrot (RNF).

## 3.2 ESPECIFICAÇÃO

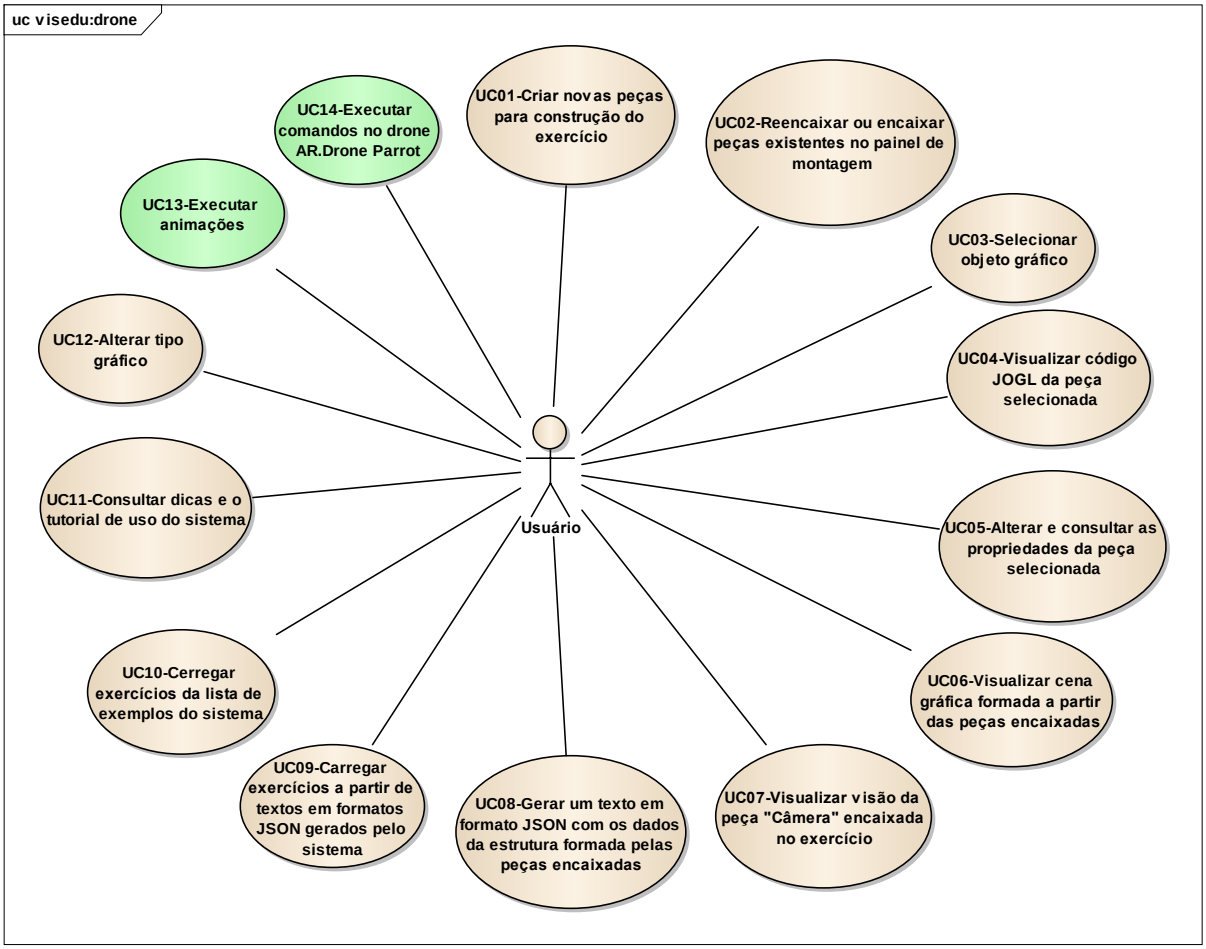
Para especificar a aplicação proposta foram utilizados diagramas *Unified Modeling Language* (UML). A seção 3.2.1 descreve os novos casos de uso e os que sofreram alterações em relação ao trabalho de Nunes (2014). Na seção 3.2.2 pode ser visto o diagrama de classes do trabalho desenvolvido. Os diagramas apresentados nesta seção foram desenvolvidos utilizando-se a ferramenta *Enterprise Architect* (EA) na versão 9.0.908.

### 3.2.1 Casos de uso

Nesta seção serão descritos os casos de uso da aplicação. A maioria dos casos de uso dos trabalhos de Montibeller (2014) e Nunes (2014) foi mantida, sendo necessário alterar somente os casos de uso UC01, UC02, UC03, UC04, UC05 e UC07 para ficarem conforme as alterações realizadas na aplicação. Além das alterações, foi removido o caso de uso UC12 existente no trabalho de Montibeller (2014) pelo fato de que o painel `Lista de peças` que especificava foi removido. Para contemplar as extensões realizadas (figura 8), foram incluídos

os casos de uso **UC13** e **UC14**. O detalhamento dos casos de uso alterados e dos adicionados neste trabalho pode ser encontrado no apêndice A.

Figura 8 – Casos de uso

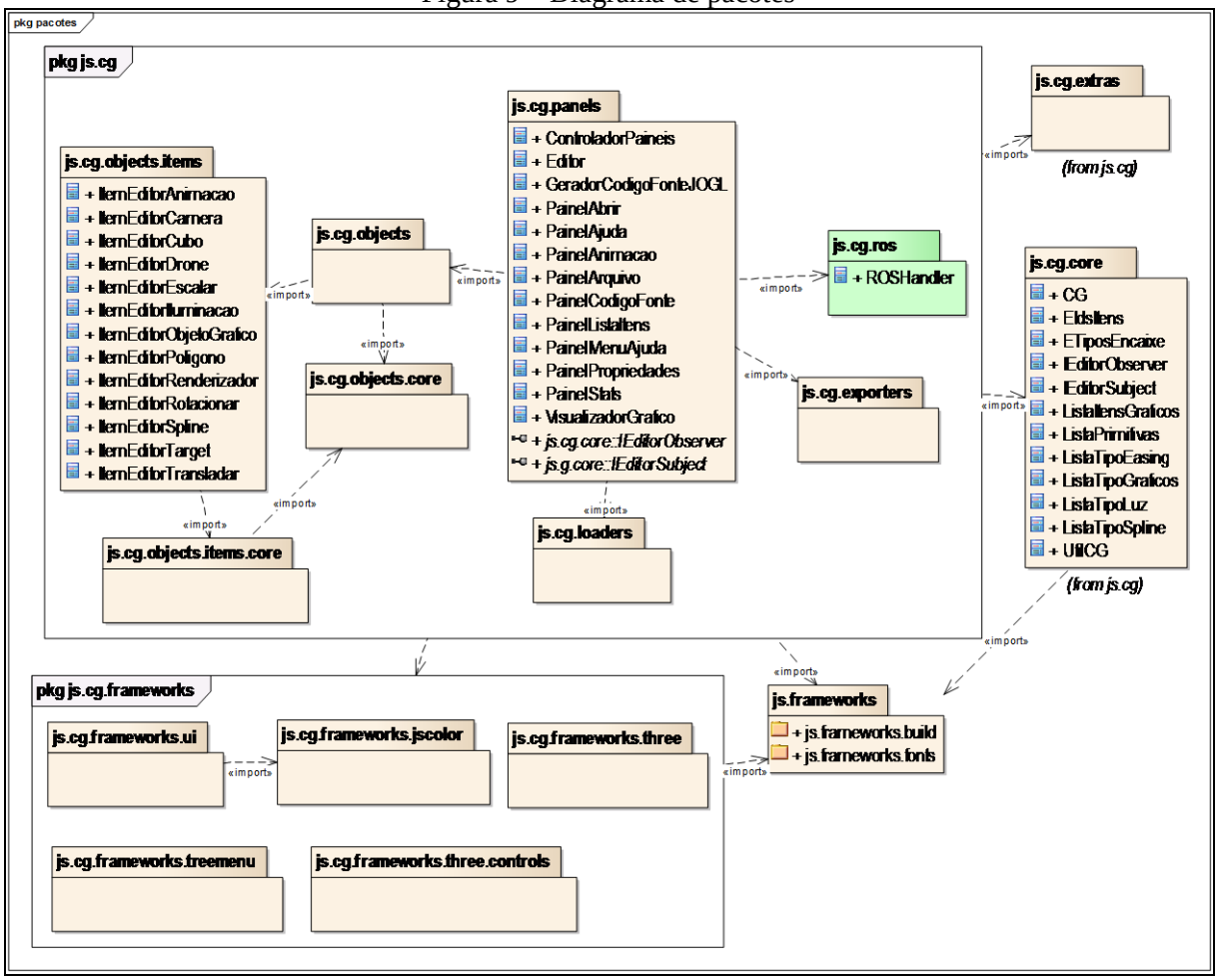



Fonte: estendido de Nunes (2014, p.32).

### 3.2.2 Diagrama de classes

Nesta seção são descritas todas as classes que foram incluídas e alteradas durante a implementação deste trabalho. Toda a estrutura de classes do trabalho de Nunes (2014) foi mantida. Na figura 9 pode ser visto o diagrama de pacotes e as dependências entre eles.

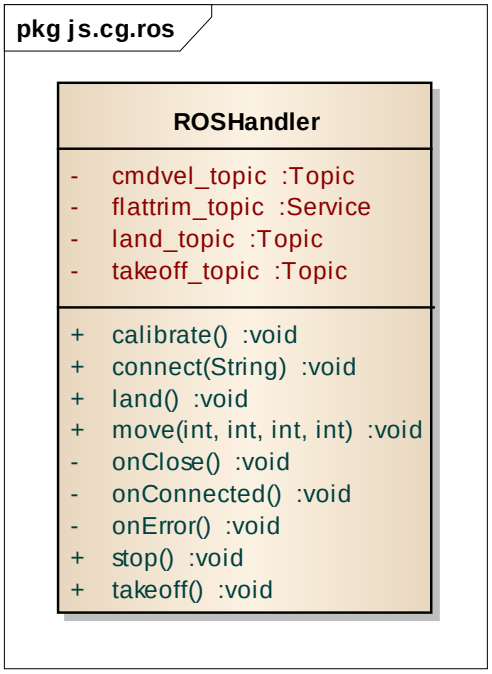
Figura 9 – Diagrama de pacotes




Fonte: estendido de Nunes (2014, p.33).

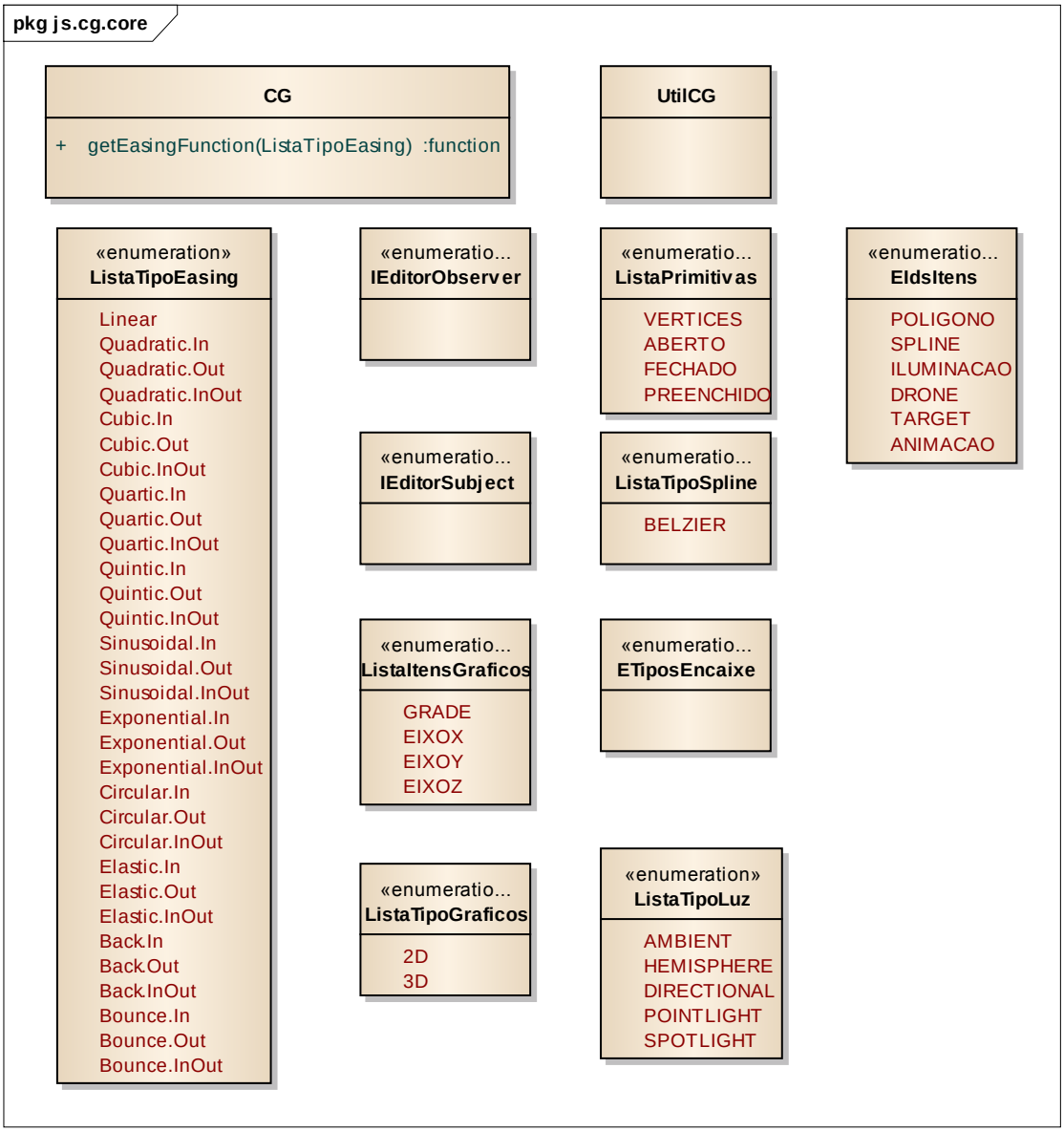
Para viabilizar a realização deste trabalho foi necessário adicionar o pacote `js.cg.ros`. Todos os demais já existiam no trabalho de Nunes (2014). O pacote incluso, que pode ser visualizado na figura 10, contém a classe `ROSHandler`, a qual é utilizada pelo painel `Animação` e é responsável por realizar a comunicação com o servidor ROS. Além disso, também fornece os métodos necessários para enviar comandos que controlam o voo do *drone* físico.

Figura 10 – diagrama do pacote `js.cg.ros`




Além da nova classe adicionada foi necessário alterar classes já presentes na aplicação. Algumas dessas alterações foram feitas em classes pertencentes ao pacote `js.cg.core`. Segundo Nunes (2014, p.34), neste pacote são encontradas as classes que dão suporte as demais classes de toda a aplicação. Nunes (2014, p.34) também diz que na classe `CG` podem ser encontrados funções utilitárias, cores utilizadas na `Fábrica de peças`, lista de exemplos, entre outros. Como pode ser observado na figura 11, nesta classe foram adicionados a enumeração `ListaTipoEasing` que armazena as possíveis funções de *easing* usadas nas animações, além de uma função utilitária chamada `getEasingFunction` que retorna à função da biblioteca Tween correspondente ao item da enumeração escolhido. A função retornada é utilizada na execução das animações. Além disso, foram incluídos novos itens nas enumerações `colors`, `msgs` existentes dentro da classe `CG` e na `EIdsItens`. Esses novos itens são utilizados na `Fábrica de peças` e em mensagens exibidas para o usuário.

Figura 11 – Diagrama do pacote `js.cg.core`

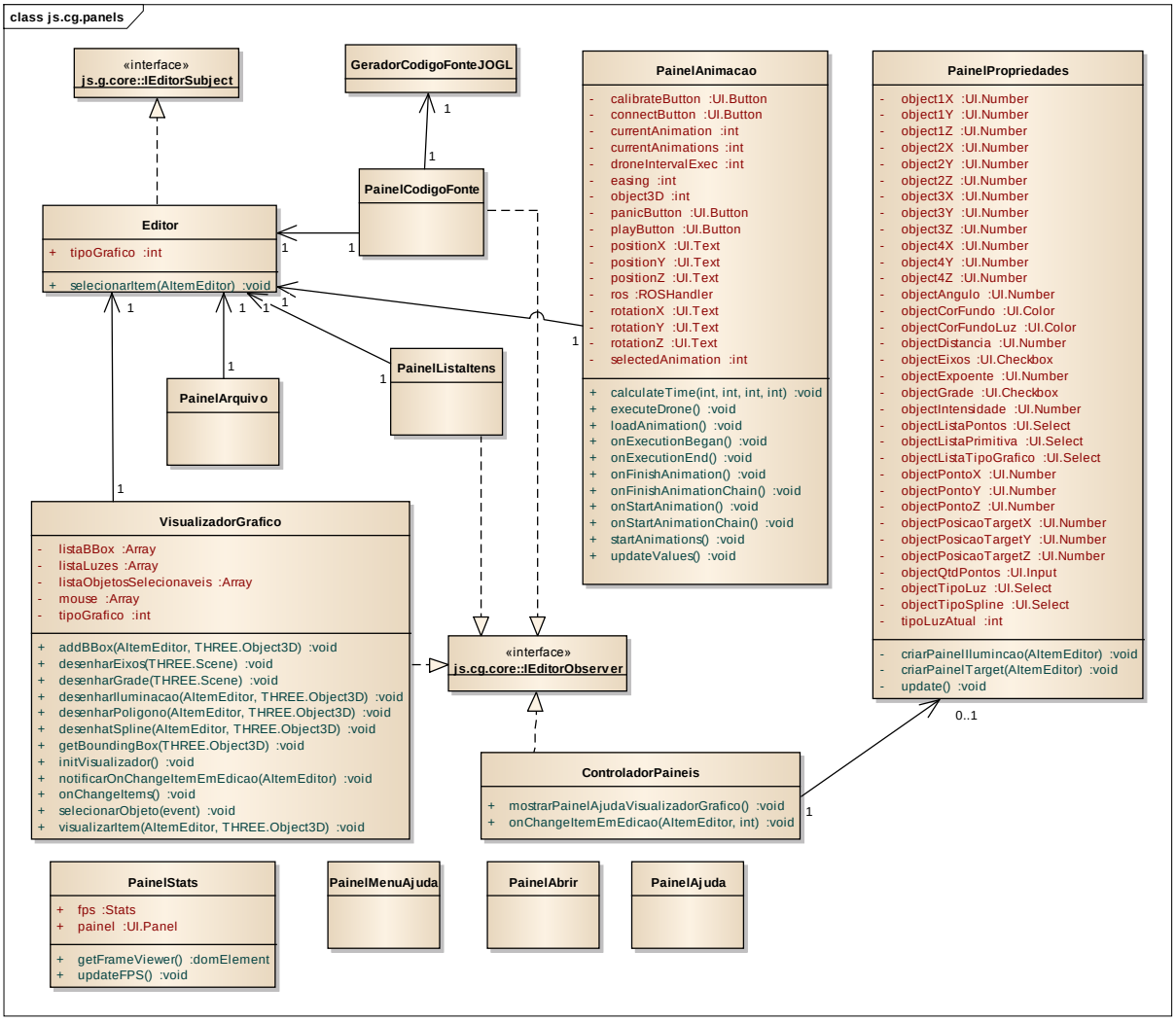



Fonte: estendido de Nunes (2014, p.34).

Dentro do pacote `js.cg.objects` é encontrada a `Fábrica` e o sub pacote `js.cg.objects.items`. Neste sub pacote é onde ficam todas as classes que controlam os itens utilizados na `Fábrica de peças` (NUNES, 2014, p. 35). Dentro dele foram inclusas as classes `ItemEditorDrone`, `ItemEditorTarget` e `ItemEditorAnimacao`. Elas correspondem às novas peças criadas para contemplar os novos recursos criados por este trabalho. O `ItemEditorDrone` é a classe que possui os atributos do item `Drone` existente no editor. O `ItemEditorTarget` é a classe onde são encontrados os atributos do item `Destino` existente na `Fábrica de peças`, enquanto os atributos do item `Animação`, que são necessários para parametrizar a execução das animações, são armazenados em `ItemEditorAnimacao`.

Os parâmetros existentes dentro da classe `ItemEditorAnimacao` são utilizados na nova classe `PainelAnimacao` adicionada dentro do pacote `js.cg.panels`. O objetivo dessa classe é controlar as animações e a execução em um *drone* físico e corresponde ao painel `Animação`. Como pode ser observado na figura 12, possui funções necessárias à realização das funcionalidades esperadas, bem como funções de *callback* que são chamadas no início de uma animação, durante sua execução e no seu término.

Figura 12 – Diagrama do pacote `js.cg.panels`




Fonte: adaptado de Nunes (2014, p.37).

Dentro do pacote `js.cg.frameworks`, são encontradas as principais bibliotecas utilizadas pela aplicação (NUNES, 2014, p.34). Como observado na figura 13, o trabalho desenvolvido atualizou a biblioteca Tween (TWEEN, 2014) e adicionou as seguintes bibliotecas no sub pacote `js.framework.build`:

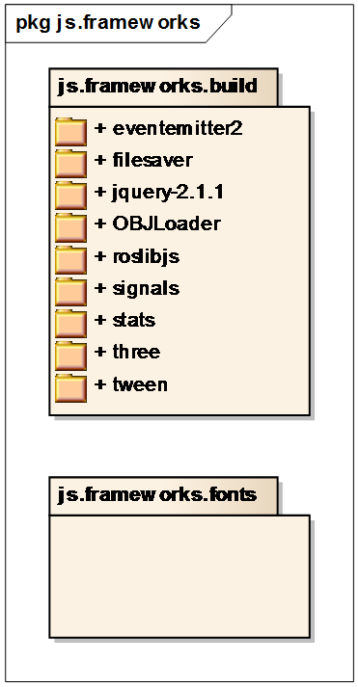
a) `OBJLoader`: biblioteca Javascript utilizada para carregar arquivos `.obj`;

b) `roslibjs`: biblioteca Javascript utilizada para conectar e enviar comandos ao servidor ROS;

c) `eventemitter2`: biblioteca Javascript necessária para utilizar a `roslibjs`;

d) `filesaver`: biblioteca Javascript utilizada para salvar arquivos na máquina local.

Figura 13 – Diagrama do pacote `js.frameworks`




## 3.3 IMPLEMENTAÇÃO

Nesta seção serão abordadas as técnicas e ferramentas utilizadas durante o processo de desenvolvimento deste trabalho, além de apresentar quais foram as modificações necessárias no trabalho de Nunes (2014) para que a extensão se tornasse funcional. Durante toda esta seção sempre que o texto se referir ao AR.Drone Parrot 2.0 físico será utilizada a expressão "*drone* físico"

### 3.3.1 Técnicas e ferramentas utilizadas

Por se tratar de uma aplicação para ser utilizada em navegadores de Internet, o trabalho de Nunes (2014) e o trabalho desenvolvido foram implementados na linguagem de programação Javascript, HTML5 e WebGL. Durante o processo de desenvolvimento o editor de texto Vi improved (VIM) na versão 7.4.475 foi utilizado para a criação e edição dos arquivos de código fonte. Para facilitar o desenvolvimento, foi adicionado ao VIM um plug-in para navegação entre diretórios. O NERD Tree em sua versão 4.2.0. Foram também utilizados no processo de implementação os navegadores de Internet Mozilla Firefox na versão 37.0.2 e Google Chrome na versão 42.0.2311.152. Para a realização dos testes com o *drone* físico, foi utilizado o *drone* AR.Drone Parrot 2.0 Elite Edition.

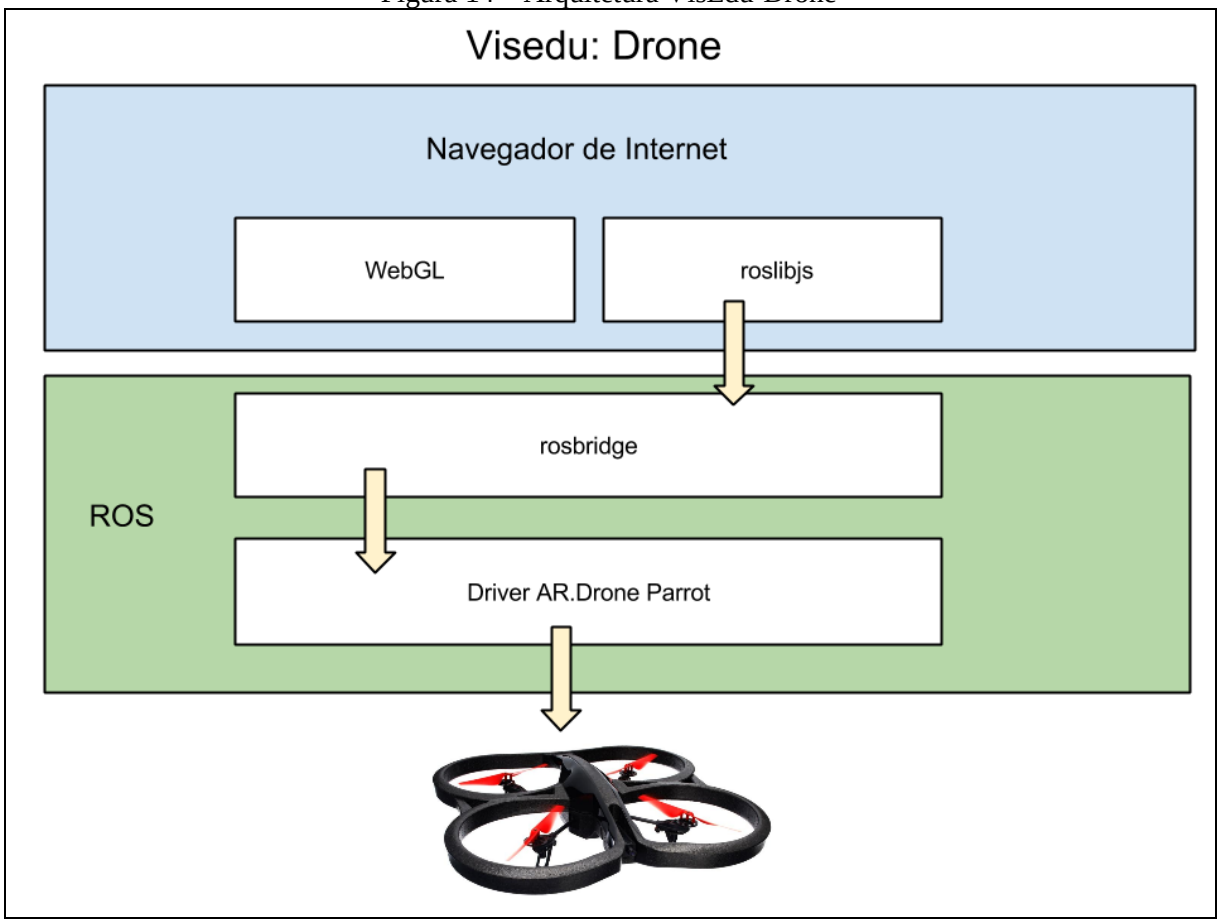
Como sistema de versionamento dos arquivos de código fonte foi utilizado o Git na versão 2.1.0, sendo utilizado o repositório pertencente ao GCG hospedado no BitBucket. Todo o desenvolvimento deste trabalho foi feito em dois computadores: um laptop com sistema operacional Linux, na distribuição Fedora 21 64 *bits* em um notebook Samsung com um processador Intel Core i7 2.40 *Giga Hertz* (GHz) com 4 núcleos, 8 *Gigabytes* (GB) de memória RAM e com uma *Graphics Processor Unit (*GPU) NVIDIA GF108M. E, um computador com sistema operacional Windows 7 64 *bits* com um processador Intel Core i7 3.20 *Giga Hertz* (GHz) com 4 núcleos, 4 *Gigabytes* (GB) de memória RAM e com uma *Graphics Processor Unit (*GPU) NVIDIA GTX480.

### 3.3.2 Arquitetura

Esta seção descreve a arquitetura da aplicação desenvolvida. Na figura 14 pode ser observado que o VisEdu-Drone é divido em duas camadas. A primeira camada, destacada em azul, refere-se ao que é executado no navegador do usuário. Ela consiste na interface gráfica onde o usuário interage com o editor criando e visualizando a cena, executa as animações e envia comandos para o *drone* físico. A segunda camada, por sua vez, é onde ocorre a comunicação com o AR.Drone Parrot.

A camada de comunicação destacada em verde corresponde as aplicações que rodam dentro do ecossistema do ROS. Ela é subdividida em dois pacotes principais, o `rosbridge` e o *driver* para AR.Drone Parrot. Pelo fato de se tratar de um *framework* para robótica genérico o ROS por si só não consegue se comunicar com o *drone*. Por esse motivo, é necessário que se instale o pacote com o *driver* que realiza essa comunicação. Uma vez instalado, a comunicação com o *drone* se torna viável. Porém, aplicações que se utilizam do ROS são em sua grande maioria escritas nas linguagens de programação Python ou C/C++. Como o *framework* não possui suporte nativo a aplicações escritas em Javascript, foi utilizado o `rosbridge`. Ele consiste em um software que permite a comunicação com o servidor ROS a partir de *websocket*, de forma que a biblioteca Javascript `roslibjs`, que também é desenvolvida pelos desenvolvedores da `rosbridge`, consiga conectar e enviar comandos ao servidor ROS.

Figura 14 – Arquitetura VisEdu-Drone




### 3.3.3 VISEDU-DRONE

Nesta seção será descrita a implementação do Visedu-Drone apresentando quais foram as alterações realizadas no trabalho de Nunes (2014).

#### 3.3.3.1 Fábrica de peças

Para que fosse possível a utilização da extensão desenvolvida neste trabalho foi necessário incluir novas peças na `Fábrica de peças`. Foram adicionados três novos itens: o item `Drone`, `Destino` e `Animação`. Cada qual estende a classe base conforme o tipo de item. Sendo assim, as classes `ItemEditorDrone` e `ItemEditorTarget`, que correspondem respectivamente ao item `Drone` e `Destino`, estendem da classe `AitemEditorEncaixaQuadrado`. Enquanto a classe `ItemEditorAnimacao`, responsável pela peça `Animação`, estende a classe `AitemEditorEncaixeSeta`. O quadro 2 apresenta o código fonte da classe `ItemEditorDrone` que é responsável por controlar a peça *drone*.

Quadro 2 – Código fonte da classe `ItemEditorDrone`

```
1  /**
2  * Classe do item drone da fábrica de peças
3  */
4  function ItemEditorDrone() {
5  AItemEditorEncaixeQuadrado.call( this );
6  var scope = this;
7  //propriedades
8  scope.valorXYZ = undefined;
9  scope.id =  EIdsItens.DRONE;
10 scope.propriedadeCor = undefined;
11 scope.object3D = undefined;
12 scope.texture = new THREE.Texture();
13 createObject3D(); //load the obj file;
14
15 function createObject3D() {
16     if( scope.object3D == undefined ){
17         CG.ImageLoader.load( 'resources/drone.jpg', function ( image )
18 {
19                 scope.texture.image = image;
20                 scope.texture.needsUpdate = true;
21         } );
22         CG.OBJLoader.load("resources/Drone_1.obj",
23             function ( model ) {
24                 model.traverse( function ( child ) {
25                     if ( child instanceof THREE.Mesh ) {
26                         child.material.map = scope.texture;
27                     }
28                 } );
29                 console.log("Modelo 3D carregado com sucesso");
30                 scope.object3D = model;
31                 scope.object3D.item = scope;
32             },
33             function( progress ){ //função executada durante o
34 carregamento do .obj
35                 console.log(progress);
36             },
37             function( error ){ //função executada em caso de erro ao
38 carregar o .obj
39                 alert("Erro ao carregar modelo 3D");
40                 console.log("Erro ao carregar modelo 3D");
41             });
42     }
43 }
44 }
45 ItemEditorDrone.prototype = Object.create(
46 AItemEditorEncaixeQuadrado.prototype );
47 .prototype );
```

As linhas entre 6 e 13 são as propriedades do item `Drone`. A função definida a partir da linha 15, que é chamada na linha 13, utiliza a classe `ImageLoader` existente na biblioteca Three.js para carregar a imagem da textura do arquivo `drone.jpg` existente dentro do diretório `resources` (linhas 17 até 21). Depois de carregada a textura é utilizada a classe `OBJLoader`, que também fornecida pela biblioteca Three.js, para carregar o arquivo `Drone_1.obj`, existente dentro do diretório `resources`, que contém o modelo 3D de um *drone* que é utilizado quando renderizada a cena (linhas 22 até 41). Quando carregado o

modelo, é aplicada a textura previamente carregada (linhas 24 até 27). Já no quadro 3 pode ser visto o código fonte da classe `ItemEditorTarget`

Quadro 3 – Código fonte da classe `ItemEditorTarget`

```
1  function ItemEditorTarget() {
2      AItemEditorEncaixeQuadrado.call( this );
3      var scope = this;
4      //eventos
5      //evento será executado quando um filho for adicionado ou removido
6  ou alguma propriedade for alterada
7      scope.onChange = function () {
8          scope.object3D = createObject3D();
9          scope.object3D.item = scope;
10     };
11     //evento será executado quando um filho for adicionado
12     scope.onAddFilho = function ( item ) {};
13     //evento será executado quando um filho for removido
14     scope.onRemoveFilho = function ( item ) {};
15     //evento será executado quando for removido ou inserido algum filho
16 em um dos filhos do objeto, ou filho dos filhos e assim por diante
17     scope.onChangeFilhos = function ( filho ) {};
18     //evento será executado quando o nome do item for alterado
19     scope.afterChangeNome = function ( nomeAntigo ) {};
20     //propriedades
21     scope.id =  EIdsItens.TARGET
22     scope.valorXYZ.set( 100, 100, 100 );
23     scope.propriedadeCor.setHex( 0xFFFFFF );
24     scope.object3D = createObject3D();
25
26     function createObject3D() {
27         var geometria = new THREE.PlaneGeometry( 10, 10);
28         var material  = new THREE.MeshPhongMaterial({ color:
29 scope.propriedadeCor.getHex(), ambient: scope.propriedadeCor.getHex(),
30 overdraw: true });
31         var target = new THREE.Mesh( geometria, material);
32         if( scope.object3D != undefined ){
33             target.position = scope.object3D.position;
34         }
35         target.rotateX(Util.math.converteGrausParaRadianos(-90));
36         return target;
37     }
38 }
39 ItemEditorTarget.prototype = Object.create(
40 AItemEditorEncaixeQuadrado.prototype );
```

Da linha 7 até 19 estão os métodos de *callbacks* para situações quando ocorre alguma alteração nas propriedades da peça. Das linhas 21 a 24 são as propriedades do item `Destino` que podem ser alteradas no painel `Propriedades da peça`. Na linha 24 pode ser observado o atributo `object3D` que possui o objeto 3D, que tem forma de um plano nos eixos `x` e `z`, utilizado quando esse item for renderizado na cena no painel `Espaço Gráfico`. Esse atributo é inicializado toda vez que um novo item `Destino` for criado com o valor retornado pela função `createObject3D` que está definida na linha 26.

O último item adicionado na aplicação foi o `ItemEditorAnimacao`. A classe referente a essa peça pode ser vista no quadro 4.

Quadro 4 – Código fonte da classe `ItemEditorAnimacao`

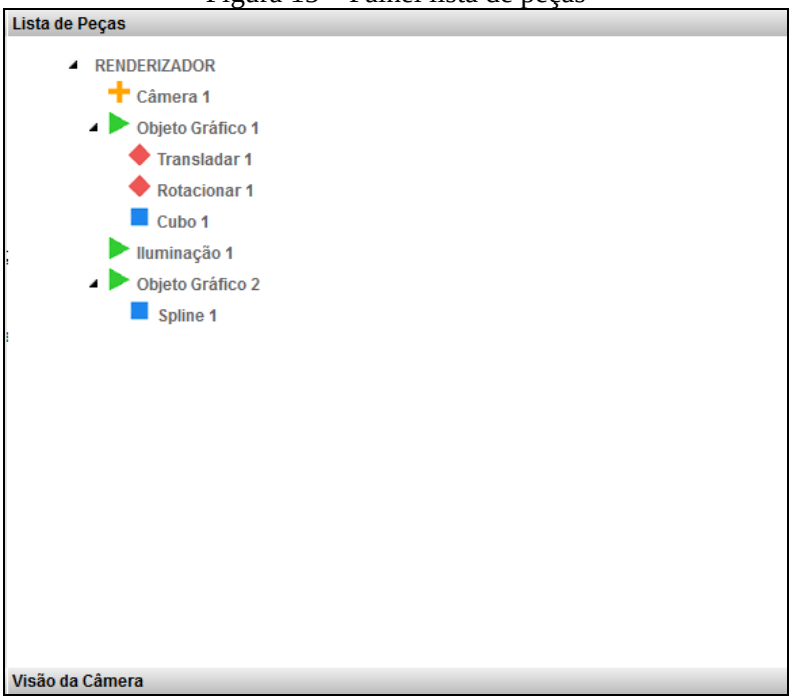
```
1   /**
2    * Item responsável pela animação do objeto gráfico
3    */
4   ItemEditorAnimacao = function() {
5       AItemEditorEncaixeSeta.call( this );
6       var scope = this;
7       //evento será executado quando um filho for adicionado ou removido
8   ou alguma propriedade for alterada
9       scope.onChange = function () {};
10      //evento será executado quando um filho for adicionado
11      scope.onAddFilho = function ( item ) {};
12      //evento será executado quando um filho for removido
13      scope.onRemoveFilho = function ( item ) {};
14      //evento será executado quando for removido ou inserido algum filho
15  em um dos filhos do objeto, ou filho dos filhos e assim por diante
16      scope.onChangeFilhos = function ( filho ) {};
17      //evento será executado quando o nome do item for alterado
18      scope.afterChangeNome = function ( nomeAntigo ) {};
19      //propriedades
20      scope.id =  EIdsItens.ANIMACAO
21      scope.corHex = CG.colors.corPecaIluminacao;
22      scope.gerarMeshsPecaSuperior();
23      /** Função utilizada nas animações*/
24      scope.easing = CG.listaTiposEasing['Linear'];
25  };
26  ItemEditorAnimacao.prototype = Object.create(
27  AItemEditorEncaixeSeta.prototype );
```

A classe `ItemEditorAnimacao`, como pode ser visto em outros itens, possui os métodos de *callback* de eventos lançados pelo editor nas linhas 9 até 18. Nas linhas 20 a 24 são definidas as propriedades da peça `Animação`. Não foi implementado suporte à geração de código fonte para os novos itens adicionados pelo fato que não se tratava do foco principal da extensão criada.

#### 3.3.3.2 Painel Animação

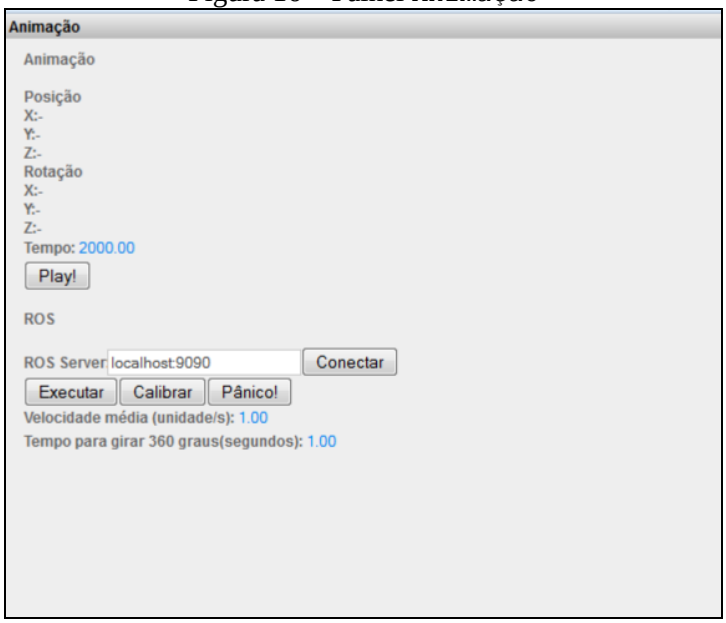
No trabalho de Montibeller (2014) existe o painel `Lista de peças`, que pode ser visto na figura 15. Segundo Montibeller (2014, p.28), esse painel exibe uma lista com as peças colocadas no exercício criado no VisEdu-CG, facilitando sua visualização e seleção.

Figura 15 – Painel lista de peças




Durante a implementação deste trabalho notou-se que as informações exibidas no painel de `Lista de peças` são as mesmas que podem ser visualizadas no editor da cena. Por esse motivo, optou-se em remover esse painel da aplicação. Em seu lugar foi colocado o painel de `Animação`, que pode ser visto na figura 16.

Figura 16 – Painel `Animação`




A classe que controla esse painel é uma subclasse de `UI.Panel` e é criada dentro do `PainelAnimacao`. Quando esta classe é instanciada são ajustadas as configurações de todos os componentes de tela e os objetos necessários para sua execução são instanciados.

Os dois recursos que esse painel fornece são a possibilidade iniciar a execução das animações e a interação com o *drone* físico. A implementação dos dois recursos será detalhada nas seções 3.3.3.2.1 e 3.3.3.2.2, respectivamente. As duas funcionalidades precisam que os itens animados (possuam o item `Animação`) existentes no editor sejam carregados. Isso é feito através do método `loadAnimation()`, que pode ser visto no quadro 5. Ele é executado sempre que os métodos para executar animações e interagir com o *drone* físico são chamados

Quadro 5 – Implementação do método `loadAnimation()`

```
1   function PainelAnimacao( editor ) {
2       ...
310     function loadAnimation(){
311         currentAnimation = 0;
312         selectedAnimation = undefined;
313         currentAnimatios = [];
314         easing = [];
315         object3D = [];
316         var item = editor.painelMontagem;
317         if (item.filhos.length > 0){
318             //percorre os filhos procurando um objeto grafico que tenha
319 animação
320             var totalAnimations = 0;
321             for (var i = 0; i < item.filhos.length; i++) {
322                 var filho = item.filhos[i];
323                 //verifica se eh um filho animado \o/
324                 if (filho.id == EIdsItens.OBJETOGRAFICO &&
325 filho.isAnimated ){
326                     currentAnimatios[totalAnimations] = [];
327                     //verifica se o item esta selecionado no editor
328                     if ( filho === editor.getItemSelecionado() ){
329                         selectedAnimation = totalAnimations;
330                     }
331                     //pega todas as animações para esse filho
332                     for( var q = 0; q < filho.filhos.length; q++ ){
333                         var animation = null;
334                         if( filho.filhos[q].tipoEncaixe ==
335 ETiposEncaixe.DIAMANTE ){
336
337 currentAnimatios[totalAnimations].push(filho.filhos[q]);
338                         } else if( filho.filhos[q].id ==
339 EIdsItens.ANIMACAO ){
340                             easing[totalAnimations] =
341 filho.filhos[q].easing;
342                         } else if( filho.filhos[q].tipoEncaixe ==
343 ETiposEncaixe.QUADRADO ){
344                             object3D[totalAnimations] =
345 filho.filhos[q].object3D;
346                         }
347                     }
348                     totalAnimations++;
349                 }
350             }
351         }
352     }
353     ...
365 }
366 PainelAnimacao.prototype = Object.create( UI.Panel.prototype );
```

Inicialmente, o método limpa todos os atributos do painel que armazenam qualquer tipo de dado utilizado nas animações (linhas 311 até 315). Após isso, nas linhas 316 e 317 é recuperada a referência do painel de montagem existente no editor e verificado se possui itens. Se o painel possuir filhos, é efetuada uma busca entre todos eles para encontrar aqueles que forem do tipo objeto gráfico e que foram animados (linhas 321 até 324). Uma vez encontrado um filho com essas características o algoritmo verifica se esse filho está atualmente selecionado. Se estiver, o índice das animações deste filho no campo `currentAnimations` é guardado no atributo `selectedAnimation` (linhas 326 a 330). Isso é necessário, pois os valores de posicionamento e rotação exibidos durante a animação no painel são do item que estiver selecionado no editor.

Por fim, como pode ser visto da linha 331 até 347, são percorridos todos os filhos do item animado. Quando o atributo `tipoEncaixe` de um filho for `EtipoEncaixe.DIAMENTE` for encontrado (linha 334), significa que um item que corresponde a algum tipo de transformação (`Translação`, `Rotação` ou `Escalar`) foi encontrado. Esse tipo de item define qual animação será executada. Sua referência é armazenada na lista de animações do objeto gráfico que está sendo carregado (linha 337). Nas linhas 338 a 340, quando encontrado o filho com o tipo `EtipoEncaixe.ANIMACAO`, é armazenado a propriedade `easing` do item no atributo `easing` do painel de animação. Essa propriedade se trata da função de interpolação selecionada pelo usuário e que deve ser utilizada para gerar os valores de posicionamento e rotação enquanto executada a animação. O filho de tipo `EtipoEncaixe.QUADRADO` define qual é o objeto 3D da biblioteca Three.js deve ser animado. Quando encontrado (linhas 442 até 445) a referência para o objeto é mantida no atributo `object3D` do painel de animações.

#### 3.3.3.2.1 Animação

No quadro 6 pode-se visualizar o código fonte do método `startAnimations()`. Esse é o método responsável em instanciar e iniciar as animações. Para isso, é utilizada a biblioteca Tween.js. Ela permite que valores de propriedade sejam alterados sem interrupções (TWEEN, 2014). Ela é utilizada para gerar os valores de posicionamento e rotação dos objetos 3D existentes na cena durante as animações. Para isso basta informar qual propriedade do objeto deve ser animada, qual o valor final deve ter quando a animação terminar e quanto tempo deve levar. A biblioteca é responsável em gerar os valores do ponto inicial ao final da animação (TWEEN, 2014). Os valores intermediários são gerados por funções de *easing* disponibilizadas pela própria biblioteca, que são configuradas pelo próprio usuário nas propriedades de peça `Animação` no editor da cena.

Quadro 6 – Implementação do método `startAnimations()`

```
120     function startAnimations() {
121         loadAnimation();
122         //se não tem animação, cai fora
123         if( currentAnimatios.length == 0 ){ return; }
124         //pega todas as animações para esse filho
125         for( var q = 0; q < currentAnimatios.length; q++ ){
126             var animation = undefined;
127             var animationChain = [];
128             for( var s = 0; s < currentAnimatios[q].length; s++ ){
129                 var animationItem = currentAnimatios[q][s];
130                 if( animationItem.id == EIdsItens.TRANSLADAR ){
131                     animation = new TWEEN.Tween(object3D[q].position)
132                     .to({x: (animationItem.valorXYZ.x >= 0 ? "+" : "-")
133 + Math.abs(animationItem.valorXYZ.x),
134                         y: (animationItem.valorXYZ.y >= 0 ? "+" : "-")
135 + Math.abs(animationItem.valorXYZ.y),
136                         z: (animationItem.valorXYZ.z >= 0 ? "+" : "-")
137 + Math.abs(animationItem.valorXYZ.z)}, time.getValue())
138                     .easing(CG.getEasingFunction(easing[q]))
139                     if( selectedAnimation != undefined && q ==
140 selectedAnimation ){
141                         setAnimationCallbacks(animation, q, s);
142                     }
143                 } else if( animationItem.id == EIdsItens.ROTACIONAR ){
144                     animation = new TWEEN.Tween(object3D[q].rotation)
145                     .to({x: (animationItem.valorXYZ.x >= 0 ? "+" : "-")
146 +
147 Util.math.converteGrausParaRadianos(Math.abs(animationItem.valorXYZ.x))
148 ,
149                         y: (animationItem.valorXYZ.y >= 0 ? "+" : "-")
150 +
151 Util.math.converteGrausParaRadianos(Math.abs(animationItem.valorXYZ.y))
152 ,
153                         z: (animationItem.valorXYZ.z >= 0 ? "+" : "-")
154 +
155 Util.math.converteGrausParaRadianos(Math.abs(animationItem.valorXYZ.z))
156 }, time.getValue())
157                     .easing(CG.getEasingFunction(easing[q]));
158                     if( selectedAnimation != undefined && q ==
159 selectedAnimation ){
160                         setAnimationCallbacks(animation, q, s);
161                     }
162                 }
163                 animationChain.push(animation);
164             }
165             //monta o chain com todas as animações para esse obj
166 gráfico
167             for( var ai = animationChain.length-1; ai > 0; ai-- ){
168                 animationChain[ai-1].chain(animationChain[ai]);
169             }
170             //iniciar!
171             if( animationChain[0] != undefined ){
172                 animationChain[0].start();
173             }
174         }
175     }
```

A primeira instrução que o método executa é a chamada ao método `loadAnimations()` (linha 121). Após o carregamento dos objetos gráficos animados, é feita

uma interação sobre todos eles em busca dos itens filhos que possuírem identificação `EidsItens.TRANSLADAR` e `EIdsItens.ROTACIONAR` (linhas 130 e 143). Quando encontrado algum tipo desses filhos é instanciado um objeto `Tween` (linhas 131 e 144). Quando criado esse objeto é passado como parâmetro do construtor qual objeto 3D da cena deve ser animado. Nas linhas 132 até 137 e 145 até 156 pode ser visto que, depois de criado o objeto é chamado o método `to()` passando dois parâmetros: o primeiro trata-se de quais são as propriedades que devem ser atualizadas e qual o valor máximo que deve ser atingido ao final da execução; o segundo parâmetro é o tempo que a execução desta animação deve durar. Nas linhas 138 e 157 é configurada a função de *easing* utilizada. Essa função irá gerar os valores que serão somados à propriedade do objeto 3D durante a execução da animação. Se as animações que estiverem sendo criadas correspondem ao item selecionado no editor é necessário que sejam configurados alguns métodos de *callback* que serão chamados durante a animação. Para isso, é invocado o método `setAnimationCallbacks()` que pode ser visto no quadro 7. No final (linhas 167 a 169) as animações que foram criadas e configuradas são colocadas na sequência de animações do objeto gráfico a qual pertencem. São ligadas entre si para que ao termino de uma a seguinte comece automaticamente. Por último a execução é iniciada (linha 171, 172 e 173).

Quadro 7 – Implementação do método `setAnimationCallbacks()`

```
1   function PainelAnimacao( editor ) {
2       ...
166     /** Configura os métodos de callbacks das animações */
167     function setAnimationCallbacks(animation, animationChain,
168 animationStep) {
169         animation.onUpdate(updateValues);
170         if( animationStep == currentAnimatios[animationChain].length-1
171 ){
172 
173 animation.onComplete(onFinishAnimationChain).onStop(onFinishAnimationC
174 hain);
175         } else {
176 
177 animation.onComplete(onFinishAnimation).onStop(onFinishAnimation);
178         }
179         if( animationStep == 0 ){
180             animation.onStart(onStartAnimationChain);
181         } else {
182             animation.onStart(onStartAnimation);
183         }
184     }
185     ...
366 }
367 PainelAnimacao.prototype = Object.create( UI.Panel.prototype );
```

Como pode ser observado na implementação do método `setAnimationCallbacks()` na linha 169 é configurado o método de *callback* `updateValues()`. Esse método é chamado

cada vez que os valores das propriedades do objeto 3D animado são alterados. Entre as linhas 170 até 183 são configurados os métodos de *callback*s chamados quando a sequência de animações começa e termina e quando uma animação individual é iniciada e terminada. A implementação desses métodos pode ser vista no quadro 9.

No quadro 8 pode ser visualizado o método que atualiza os valores dos componentes de tela com os valores do objeto 3D animado (linhas 186 até 197).

Quadro 8 – Implementação do método `updateValues()`

```
1   function PainelAnimacao( editor ) {
2       ...
182     /**
183      * Função para atualizar os valores do object3D durante a animação
184      */
185     function updateValues() {
186         positionX.setValue(object3D[selectedAnimation].position.x);
187         positionY.setValue(object3D[selectedAnimation].position.y);
188         positionZ.setValue(object3D[selectedAnimation].position.z);
189 
190 rotationX.setValue(Util.math.converteRadianosParaGraus(object3D[select
191 edAnimation].rotation.x));
192 
193 rotationY.setValue(Util.math.converteRadianosParaGraus(object3D[select
194 edAnimation].rotation.y));
195 
196 rotationZ.setValue(Util.math.converteRadianosParaGraus(object3D[select
197 edAnimation].rotation.z));
198     }
199     ...
366 }
367 PainelAnimacao.prototype = Object.create( UI.Panel.prototype );
```

Os métodos `onStartAnimation()` e `onFinishAnimation()` (quadro 9 entre as linhas 198 e 214) são chamados antes de uma animação começar e depois que a terminou, respectivamente. No método `onStartAnimation()` é alterada a cor do item no editor para o qual a animação estiver sendo iniciada e no método `onFinishAnimation()` a cor do item restaurada, permitindo assim que o usuário visualize qual passo da animação está sendo executado. Os métodos `onStartAnimationChain()` e `onFinishAnimationChain()`, nas linhas 219 até 230, são chamados quando é iniciada e terminada uma sequência de animações. Esses métodos além de realizarem as mesmas coisas que os métodos de início e fim de animação também são utilizados para bloquear e desbloquear os botões que iniciam a execução de animações e da interação com o *drone* físico Dessa forma evita-se que animações ocorram uma sobre as outras e que duas interações com o *drone* causem acidentes. Os botões são desabilitados no método `onExecutionBegan()` e habilitados novamente em `onExecutionEnd()`. Esses mesmos métodos para habilitar e desabilitar os botões também são chamados quando interagido com o *drone* físico.

Quadro 9 – Implementação dos métodos de callback das animações

```
1   function PainelAnimacao( editor ) {
2       ...
195     /**
196      * Função volta para a cor default do item no editor
197      */
198     function onFinishAnimation(){
199
200 currentAnimatios[selectedAnimation][currentAnimation].setMeshsColor(
201 CG.colors.corPecasDiamante );
202         currentAnimation++;
203     }
204
205     /**
206      * Função muda a cor do item de animação no editor para indicar a
207 execução do mesmo
208      */
209     function onStartAnimation(){
210
211 currentAnimatios[selectedAnimation][currentAnimation].setMeshsColor(
212 CG.colors.corCurrentAnimation );
213     }
214
215     /**
216      * Função executada no inicio do animation chain
217      */
218     function onStartAnimationChain(){
219         onExecutionBegan();
220         onStartAnimation();
221     }
222
223     /**
224      * Função executada no final do animation chain
225      */
226     function onFinishAnimationChain(){
227         onExecutionEnd();
228         onFinishAnimation();
229     }
230     ...
231 }
366 PainelAnimacao.prototype = Object.create( UI.Panel.prototype );
367
```

3.3.3.2.2 Execução com drone físico

Nesta seção será descrita a rotina utilizada para interagir com o *drone* físico. Para isso, é necessário compreender alguns métodos auxiliares que foram criados para essa tarefa. Os métodos auxiliares são: o `loadAnimations()` que é descrito no início da seção 3.3.3.2 e o método `calculateTime()` o qual pode ser visto no quadro 10.

Tendo em vista que o *drone* não possui uma forma de voo onde é passada a distância que se deseja que ele percorra foi necessário criar uma rotina onde é calculado o tempo necessário que um *drone* deve se movimentar para chegar até o ponto desejado. Para que isso fosse possível, foram criadas duas configurações que precisam ser informadas para que o

cálculo funcione de forma correta. Essas configurações podem ser encontradas no painel de animação e são:

a) `Velocidade média`: tempo em segundos que o *drone* leva para percorrer uma unidade de distância definida pelo usuário;

b) `Tempo para completar 360°`: tempo em segundos que o *drone* levar para realizar uma rotação completa de 360º em seu próprio eixo.

Quadro 10 – Implementação do método `calculateTime()`

```
1   function PainelAnimacao( editor ) {
2       ...
284     /** Função que calcula o tempo em milisegundos necessário para
285 executar a movimentação
286      * informada pelo usuário
287      */
288     function calculateTime(x, y, z, rotation){
289         var seconds = 0
290         if( x != undefined && x > 0 ){
291             seconds = distanceAverage.getValue() * x;
292         }
293         if( seconds == 0 && y != undefined && y > 0 ){
294             seconds = distanceAverage.getValue() * y;
295         }
296         if( seconds == 0 && z != undefined && z > 0 ){
297             seconds = distanceAverage.getValue() * z;
298         }
299         if( seconds > 0 ){
300             return seconds * 1000;
301         }
302         if( rotation != undefined && rotation > 0 ){
303             return  ((rotation * rotationAverage.getValue()) / 360) *
304 1000;
305         }
306         return 0;
307     }
308     ...
309 }
366 PainelAnimacao.prototype = Object.create( UI.Panel.prototype );
367
```

Quando chamado o método `calculateTime()` são passados como parâmetros a quantidade de unidade de distâncias que se deseja percorrer nos eixos `x`,`y`,`z` e a rotação desejada. Como pode ser visto nas linhas 289 até 302, o valor passado como parâmetro para os eixos `x`, `y` e `z` são multiplicados pelo valor da média de distância percorrida pelo *drone* em um segundo. O resultado dessa multiplicação informa o tempo em segundos necessário para percorrer a distância desejada. Se o valor calculado for maior que zero é realizada a multiplicação por mil, para transformar segundos em milissegundos, e o produto dessa multiplicação é retornado. Caso o valor de segundos não seja superior a zero, significa que não foram passados os valores de movimentação para os eixos `x`, `y` e `z`. Neste caso, é verificado se o valor do parâmetro de rotação é maior que zero. Se a verificação indicar um

valor superior a zero (linhas 303 a 306), o tempo para efetuar a rotação precisa ser calculado e retornado. Para fazer esse cálculo o algoritmo multiplica o valor de rotação passado como parâmetro pelo valor do tempo em segundos necessários para o *drone* realizar uma volta completa (360º) em seu próprio eixo. O produto da multiplicação é dividido por 360. Assim, se obtém o tempo em segundos necessários para que o *drone* rotacione a quantidade de graus especificada no item de rotação. Uma vez obtido este valor é feita outra multiplicação para o valor retornado seja em milissegundos. Se o valor do parâmetro de rotação também for inferior ou igual a zero, o método retorna zero (linha 307).

No quadro 11 pode ser visto a implementação do método que interage com o AR.Drone Parrot. O início do método consiste em carregar as animações que estão no editor (linha 231), verificar se não ocorreu algum problema no carregamento (linha 233), desabilitar os botões que iniciam as animações e interação com o *drone* físico (linha 235) e inicializar todas as variáveis necessárias para a execução da rotina (linhas 236 e 237).

Quadro 11 – Implementação do método `executeDrone()`

```
1   function PainelAnimacao( editor ) {
2       ...
226     /**
227      * Função que executa as animações expecificadas no editor no
228 drone real
229      */
230     function executeDrone(){
231         loadAnimation();
232         //se não tem animação, cai fora
233         if( currentAnimatios.length == 0 || selectedAnimation ==
234 undefined ){ return; }
235         onExecutionBegan()
236         var currentDroneStep = -1;
237         var droneParado = true;
238         var interval = setInterval(executeStepDrone, 10);
239         function executeStepDrone() {
240             ...
241         }
242     }
```

A variável `currentDroneStep` (linha 236) indica qual item da sequência de animações deve ser utilizado para movimentar o *drone* na próxima interação executada pela função aninhada `executeStepDrone()`. Na linha 237 está definida a variável `droneParado`. Ela também é utilizada pela função aninhada para que a execução do próximo item na sequência de animações seja executada somente quando o *drone* estiver parado. Para que todas as animações a serem executadas ocorram de uma forma continua é chamado o método `setInterval()` (linha 238). Essa função faz com que a função aninhada `executeStepDrone()` seja executada a cada dez milissegundos. O código de `executeStepDrone()` pode ser visto no quadro 12.

Quadro 12 – Implementação do método `executeStepDrone()`

```
237     function executeStepDrone() {
238         if( droneParado && currentDroneStep == -1){
239             ros.takeoff();
240             droneParado = false;
241             setTimeout(function(){
242                 ros.stop();
243                 console.log('decolou -' + new Date());
244                 droneParado = true;
245                 currentDroneStep += 1;
246             }, 8000);
247         } else if( droneParado && currentDroneStep <
248 currentAnimatios[selectedAnimation].length){
249             var valorX =
250 currentAnimatios[selectedAnimation][currentDroneStep].valorXYZ.x > 0 ? 0,5: -
251 0.5;
252             var valorY =
253 currentAnimatios[selectedAnimation][currentDroneStep].valorXYZ.y > 0 ? 0,5: -
254 0.5;
255             var valorZ =
256 currentAnimatios[selectedAnimation][currentDroneStep].valorXYZ.z > 0 ? 0,5: -
257 0.5;
258             var wait = 0;
259             if( currentAnimatios[selectedAnimation][currentDroneStep].id ==
260 EIdsItens.TRANSLADAR ){
261                 wait =
262 calculateTime(currentAnimatios[selectedAnimation][currentDroneStep].valorXYZ.x
263 ,
264 currentAnimatios[selectedAnimation][currentDroneStep].valorXYZ.y,
265
266 currentAnimatios[selectedAnimation][currentDroneStep].valorXYZ.z, 0.0);
267                 console.log('transladou - ' + new Date());
268                 ros.move(valorX,valorY,valorZ, 0.0);
269             } else if(
270 currentAnimatios[selectedAnimation][currentDroneStep].id ==
271 EIdsItens.ROTACIONAR ){
272                 wait = calculateTime(0,0,0,
273 currentAnimatios[selectedAnimation][currentDroneStep].valorXYZ.y);
274                 console.log('rotacionou - ' + new Date());
275                 ros.move(0,0,0, valorY);
276             }
277             droneParado = false;
278             setTimeout(function(){
279                 ros.stop();
280                 console.log('parou - ' + new Date());
281                 setTimeout(function(){
282                     currentDroneStep += 1;
283                     droneParado = true;
284                 }, 2000);
285             }, wait);
286         } else if( droneParado && currentDroneStep ==
287 currentAnimatios[selectedAnimation].length){
288             droneParado = false;
289             setTimeout(function(){
290                 ros.land();
291                 console.log('posou - ' + new Date());
292                 clearInterval(interval);
293                   onExecutionEnd();
294               }, 1000);
295          }
296     }
```

É a função `executeStepDrone()` que envia todos os comandos para o AR.Drone Parrot através do servidor ROS. Todos os comandos são enviados por meio do objeto de

`ROSHandler` existente na classe `PainelAnimação`. A classe `ROSHandler` é descrita mais detalhadamente na seção 3.3.3.3.

A cada interação de `executeStepDrone()` é executado um item da sequência de animações com o *drone* físico. O item a ser executado é indicado pela variável `currentStepDrone`. Se o valor encontrado nessa variável for -1 indica que a sequência de animações ainda não foi iniciada e o *drone* ainda está no chão (linha 238). Por isso, é chamado o método `ros.takeoff()` (linha 239) que faz com que o *drone* decole. Para que a próxima interação só ocorra após o *drone* já estar no ar e estabilizado é configurada a *flag* `droneParado` para falso (linha 240). Também é configurado um *timeout* para que após oito segundos a partir do início da decolagem a variável `droneParado` seja alterada para o valor verdadeiro (linhas 241 a 245). Uma vez com o *drone* no ar a execução dos itens da sequência de animações do objeto gráfico selecionado pode ser executada. Isso ocorre nas linhas 246 até 289.

A primeira coisa que o algoritmo faz é inicializar as variáveis `wait`, `valorX`, `valorY` e `valorZ` (linhas 248 até 256). A variável `wait` guarda o tempo que o *drone* vai ter que ficar em movimento para completar o passo da animação atual. As demais armazenam os valores que foram configurados para os eixos x, y e z no item da animação que vai ser executada. Como o *drone* é controlado por aceleração e tempo os valores armazenados são normalizados. Sendo atribuído o valor 0,5 se encontrado valor maior que zero, caso contrário, é atribuído -0.5. Depois que as variáveis necessárias tiverem sido inicializadas, o algoritmo verifica qual o tipo de movimentação deve ser efetuada (linhas 255 e 265), calcula o tempo necessário para executar a movimentação utilizando os valores configurados para cada eixo no item corrente (linhas 257 e 268) e envia o comando para o *drone* através do método `ros.move()` (linhas 264 e 279). Depois que o comando de movimentação é enviado é necessário que as próximas interações de `executeStepDrone()` aconteçam somente depois que o *drone* estiver parado após percorrer a distância esperada ou executado a rotação desejada. Para isso, a *flag* `droneParado` é novamente configurada para falso. Para que a execução não fique presa, é criado novamente um *timeout*, de tal forma que quando o tempo esperado seja alcançado é enviado um comando para o *drone* parar e depois de dois segundos o `currentDroneStep` é incrementado e a *flag* `droneParado` é configurada novamente para verdadeiro, liberando o fluxo de interações de `executeStepDrone()` é liberado novamente.

Se o valor da variável `currentDroneStep` for igual a quantidade de itens da sequência de animações significa que o *drone* executou todas as animações e deve ser pousado. Isso é

feito nas linhas 280 até 288. Nessas linhas o valor da *flag* `droneParado` é alterado para falso, fazendo com que as próximas interações de `executeStepDrone()` sejam ignoradas. Além disso, é chamado o método `ros.land()` que instrui o *drone* pousar (linha 284). A execução das interações da função aninhada `executeStepDrone()` são desabilitadas permanentemente (linha 286) e os botões de para iniciar a execução das animação e interação com o *drone* físico são habilitados novamente (linha 287).

#### 3.3.3.3 ROS

Nesta seção é descrito detalhadamente a implementação da classe `ROSHandler`, a qual tem como objetivo encapsular todos os comportamentos necessários com o servidor ROS utilizado para a comunicação com o *drone*. No quadro 13 pode ser observada a implementação do construtor de `ROSHandler` e seus atributos.

Quadro 13 – Implementação de `ROSHandler`

```
1  /**
2   * Objeto responsável em interagir com a librosjs
3   */
4  var ROSHandler = function() {
5
6      this.ros = new ROSLIB.Ros();
7      this.ros.on('connection', ROSHandler.prototype.onConnected);
8      this.ros.on('error', ROSHandler.prototype.onError);
9      this.ros.on('close', ROSHandler.prototype.onClose);
10     /**ros topic utilizado para decolar o AR.Drone*/
11     this.takeoff_topic = new ROSLIB.Topic({
12             ros : this.ros,
13             name : '/ardrone/takeoff',
14             messageType : 'std_msgs/Empty'
15     });
16     /** ros topic utilizado para pousar o drone*/
17     this.land_topic = new ROSLIB.Topic({
18             ros : this.ros,
19             name : '/ardrone/land',
20             messageType : 'std_msgs/Empty'
21     });
22     /** ros topic utilizado para movimentar o drone*/
23     this.cmdvel_topic = new ROSLIB.Topic({
24             ros : this.ros,
25             name : '/cmd_vel',
26             messageType : 'geometry_msgs/Twist'
27     });
28     /** ros service utilizado para calibrar os sensores do drone*/
29     this.flattrim_service = new ROSLIB.Service({
30             ros : this.ros,
31             name : '/ardrone/flattrim',
32             serviceType : 'std_msgs/Empty'
33     });
34 }
```

Na linha 6 é inicializado o atributo `ros`. Esse objeto é utilizado na comunicação com o ROS através da `roslibjs`. Essa biblioteca envia os comandos por meio do *websocket*

disponibilizado pelo `rosbridge`. Nas linhas 7, 8 e 9 são configurados alguns métodos de *callback* que são chamados pelo `ROSLIB.Ros` quando o status da conexão muda para `connection`, `error` ou `close`. Na linha 11 é inicializado o atributo `takeoff_topic`. Quando publicada uma mensagem nesse tópico o *drone* decola. O campo `land_topic`, que é inicializado na linha 17, é o tópico que quando publicada alguma mensagem faz com que o *drone* pouse. Na linha 23 é inicializado o tópico `cmdvel_topic`. Mensagens publicadas nele fazem com que o *drone* se movimente. Na linha 29 é inicializado o atributo `flattrim_service` no qual devem ser enviadas as mensagens para o serviço que efetua a calibragem em todos os sensores do *drone*. Os métodos de *callback* chamados pela `ROSLIB.Ros` podem ser vistos no quadro 14. A função deles é exibir uma mensagem para o usuário e escrever mensagens de *log* no *console* do navegador de Internet.

Quadro 14 – Implementação dos *callbacks* da `ROSLIB.Ros`

```
39  ROSHandler.prototype.onConnected = function (){
40      alert("Conexão efetuada com sucesso")
41      console.log('Connected to websocket server.');
42  }
43
44  ROSHandler.prototype.onError = function(){
45      alert("Erro de conexão com o ROS")
46      console.log('Error connecting to websocket server: ', error);
47  }
48
49  ROSHandler.prototype.onClose = function(){
50      console.log('Connection to websocket server closed');
51  }
```

`ROSHandler` também disponibiliza alguns métodos que são utilizados pela aplicação, especialmente na classe `PainelAnimacao`, para enviar comandos ao ROS. As implementações desses métodos podem ser vistas no quadro 15.

Para suprir as necessidades do VisEdu-Drone foi necessário a criação de seis métodos na classe `ROSHandler`. Entre eles está o método `connect(url)`, que pode ser visto na linha 52. Ele tem como objetivo estabelecer a conexão com o `rosbridge` através de um *websocket*. Para isso, primeiramente é feita a validação do parâmetro recebido (linha 53). É validado se a URL passada por parâmetro foi definida corretamente e se não é uma *string* vazia. Depois é verificado se o objeto de `ROSLIB.Ros` já não possui uma conexão estabelecida (linha 54). Se sim, a conexão é fechada (linha 55). Após isso, é utilizado o método `connect()` da biblioteca `roslibjs` instanciada no atributo `ros` para estabelecer a conexão com as aplicações rodando sobre o *framework* ROS (linha 57). O parâmetro passado para iniciar a conexão é formatado acrescentando-se o prefixo ws:// na URL passada no parâmetro no método `connect(url)` da classe `ROSHandler`.

Quadro 15 – Implementação dos métodos da classe `ROSHandler`

```
52  ROSHandler.prototype.connect = function ( url ) {
53      if( url != undefined && url.length > 0){
54          if( this.ros.isConnected ){
55              this.ros.close();
56          }
57          this.ros.connect("ws://" + url);
58      }
59  }
60
61  ROSHandler.prototype.takeoff = function(){
62      console.log('takeoff');
63      var takeoff = new ROSLIB.Message();
64      this.takeoff_topic.publish(takeoff);
65  };
66
67  ROSHandler.prototype.land = function() {
68      console.log('land');
69      var land = new ROSLIB.Message();
70      this.land_topic.publish(land);
71  };
72
73  ROSHandler.prototype.stop = function(){
74      this.move(0.0, 0.0, 0.0, 0.0);
75  }
76
77  ROSHandler.prototype.move = function(x,y,z,rotation) {
78      var twist = new ROSLIB.Message({
79          linear: {
80              x : x,
81              y : y,
82              z : z,
83
84          },
85          angular: {
86              x : 0.0,
87              y : 0.0,
88              z : rotation,
89          }
90      });
91      this.cmdvel_topic.publish(twist);
92  };
93
94  ROSHandler.prototype.calibrate = function(){
95      var request = new ROSLIB.ServiceRequest({});
96      this.flattrim_service.callService(request, function(response){
97          console.log(response);
98      });
99  }
100
```

Os métodos `takeoff` e `land` que podem ser vistos no quadro 15 entre as linhas 61 e 67 são utilizados, respectivamente, para decolar e pousar o *drone*. Para que isso ocorra é necessário que seja publicada uma mensagem vazia nos tópicos disponibilizados pelos atributos `takeoff_topic`, responsável pela decolagem e `land_topic`, responsável pelo pouso. Essa mensagem é criada nas linhas 63 e 69 e publicada utilizando o método `publish()` disponível através da classe `ROSLIB.Topic`. A publicação dessas mensagens pode

ser vista nas linhas 64 e 70. Para realizar a movimentação do *drone*, foi criado o método `move()`, cuja a implementação pode ser observada nas linhas 77 até 92.

Na linha 78 pode ser vista a criação da mensagem que deverá ser publicada no tópico `cmdvel_topic`. Na criação dessa mensagem é passado como parâmetro um mapa contendo dois pares de chave e valor. O primeiro deles possui a chave `linear` e o como valor outro mapa sinalizando em quais eixos o *drone* deve se movimentar quando a mensagem for publicada (linha 79). O segundo valor possui a chave `angular` e o valor atribuído a ela também é um mapa. Esse mapa, por sua vez, contém os valores de quais eixos devem ser rotacionados quando publicada a mensagem (linha 85). Todos os valores atribuídos a esses mapas são passados como parâmetro do método `move()`. Na linha 91 a mensagem criada é publicada no tópico correspondente a movimentação do *drone*. O método `stop()` que está definido na linha 73, consiste em uma chamada ao método `move()` passando todos os parâmetros com valor zero. Esse método foi criado somente para facilitar a implementação e aumentar legibilidade do código.

O método `calibrate()` que está definido na linha 94 foi criado para chamar um serviço disponibilizado pelo *driver* do *drone* físico para ROS em que os sensores do *drone* são calibrados. Para que esse serviço execute é necessário enviar uma mensagem de requisição vazia para o mesmo. A criação desta pode ser observada na linha 95. Na linha seguinte, o serviço, disponibilizado através do atributo `flattrim_service` é invocado pelo método `callService()`, passando como parâmetro a mensagem que está definida na variável `request` e uma função que será executada quando a resposta do serviço for recebida. Essa função nada mais faz do que escrever no *log* do navegador de Internet utilizado para rodar a aplicação.

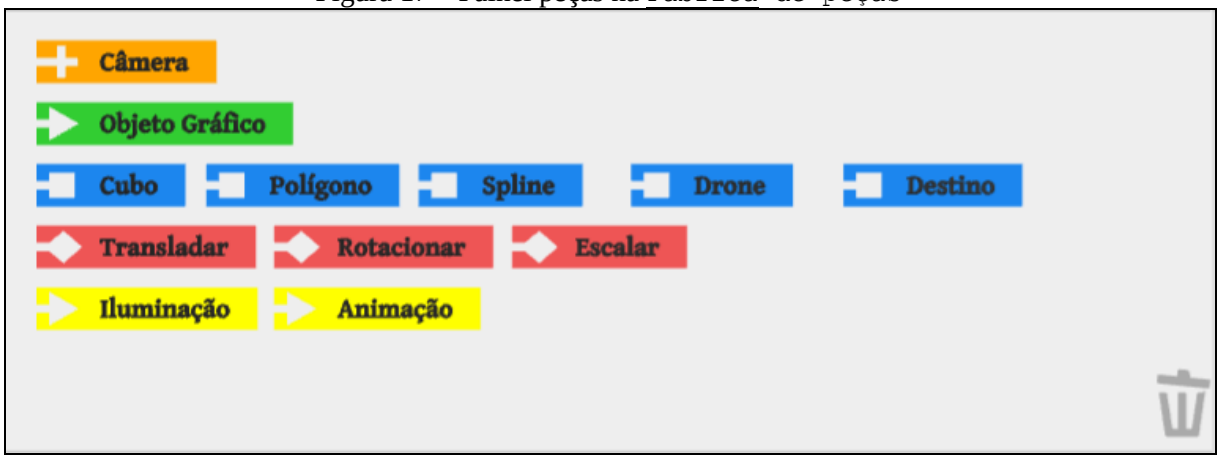
### 3.3.4 Operacionalidade da implementação

Nesta seção será apresentado o VisEdu-Drone demonstrando as novas funcionalidades desenvolvidas baseadas no trabalho de Nunes (2014).

#### 3.3.4.1 Painel `Fábrica de peças`

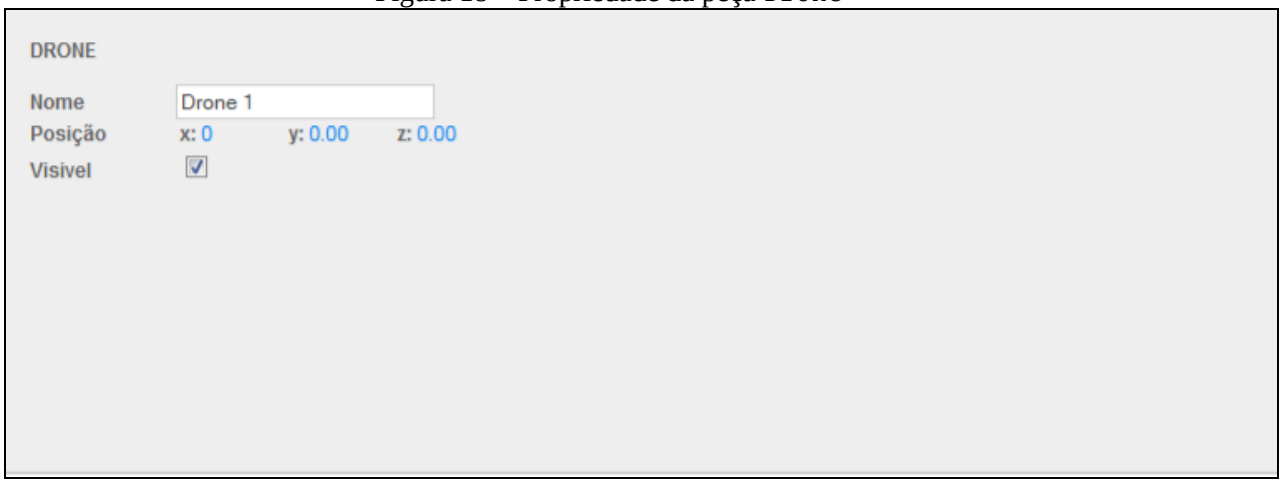
No painel `Fábrica de peças` criado por Montibeller (2014) e alterado por Nunes (2014) foram acrescentadas três novas peças. Foram adicionados os itens `Drone`, `Destino` e `Animação`. A figura 17 mostra as novas peças adicionadas.

Figura 17 – Painel peças na `Fábrica de peças`



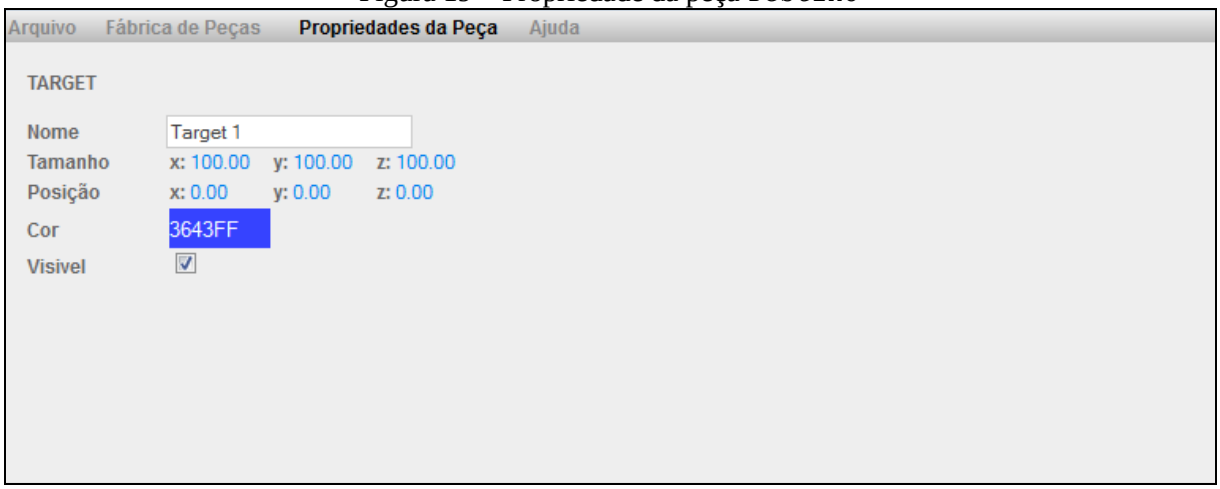

A peça `Drone` representa uma forma geométrica de um *drone* que deve ser renderizado na cena. Por possuir um encaixe quadrado só pode ser atribuído em um item do tipo Objeto Gráfico. Suas propriedades podem ser visualizadas na figura 18. As propriedades que podem ser configuradas para um item `Drone` são: o seu nome, a posição dentro da cena e se está visível.

Figura 18 – Propriedade da peça `Drone`



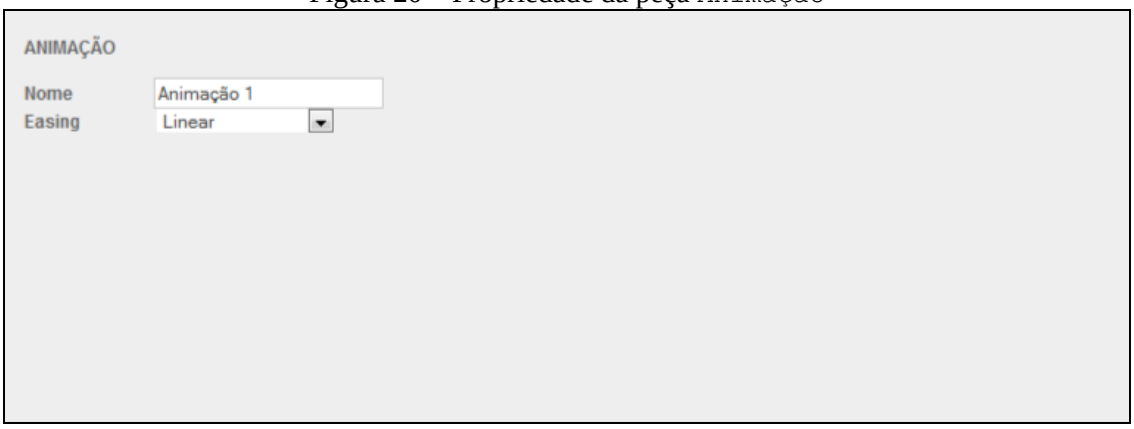

O novo item `Destino` representa o destino que o *drone* deve chegar ao terminar uma animação se os valores informados no exercício estiverem corretos. Quando um item desse é adicionado à cena um plano é renderizado no `Espaço Gráfico`. Quando essa peça é selecionada o painel de propriedade da peça (figura 19) é exibido. As propriedades são muito similares ao item `Drone`, tendo propriedades para nome, cor, posicionamento e visibilidade dentro da cena. Porém, contém uma propriedade a mais referente ao tamanho. Essa propriedade informa qual o tamanho do plano que será renderizado no `Espaço Gráfico`.

Figura 19 – Propriedade da peça `Destino`




O item `Animação` faz com que o `Objeto Gráfico` ao qual está associado seja animado. Sendo assim, não é permitido que mais de um item `Animação` seja atribuído a um mesmo pai. Quando uma peça dessas é adicionada como um item filho de um `Objeto Gráfico` o comportamento dos itens de tipo de encaixe diamante muda. Os itens passam a atuar como animações e não mais modificando a matriz de transformação do `Objeto Gráfico` na cena. Pelo fato de que o trabalho desenvolvido tem como maior interesse a integração com um *drone* real, o item `Escalar` é ignorado quando o objeto gráfico for animado, pois esse tipo de transformação não é viável de ser executado quando interagido com o *drone* físico. Como pode ser observado na figura 20 o item `Animação` possui somente duas propriedades: o nome da peça e a função de *easing* que deve ser utilizada quando executada a animação. Essa função permite que as animações tenham diferentes efeitos como uma movimentação elástica, aceleração exponencial, entre outros.

Figura 20 – Propriedade da peça `Animação`




### 3.3.4.2 Painel `Animação`

Todos os painéis desenvolvidos por Montibeller (2014) e Nunes (2014) foram mantidos, exceto o painel de `Lista de peças` que foi substituído pelo painel `Animação`. Esse

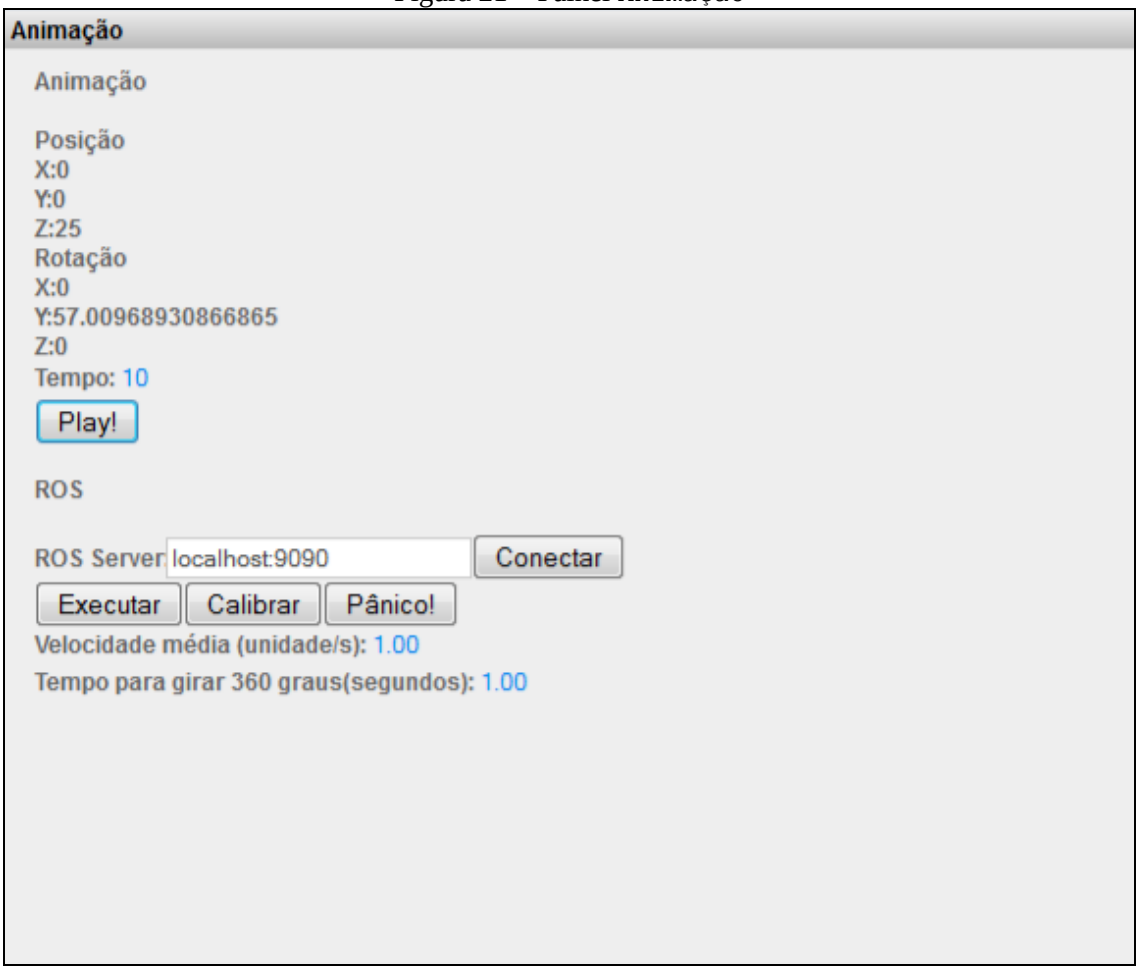
novo componente da interface é divido em dois grupos de componentes de tela. O primeiro grupo é utilizado pelas animações. Neste grupo, como pode ser visto na figura 21, ficam os campos utilizados para exibir em tempo de execução os valores de posicionamento e rotação do `Objeto Gráfico` que está selecionado na `Fábrica de peças`. Logo abaixo, pode ser encontrado um campo chamado `Tempo`. Nele pode ser configurado o tempo em segundos no que cada passo das animações definidas na `Fábrica de peças` consumirá. Para que a animação seja iniciada basta clicar no botão `Play`.

O segundo grupo existente no painel de animação, que pode ser visualizado na figura 21, corresponde ao grupo de componentes para configuração e interação com o ROS. Entre eles existe o campo de configuração de conexão com o servidor ROS. Neste campo deve ser informada a URL de conexão que deve ser utilizada para estabelecer uma conexão com o `rosbridge`, bem como o botão que iniciará essa comunicação. Assim que a conexão estiver estabelecida uma mensagem de sucesso será exibida e os botões de disponíveis para interação com ROS podem ser executados, sendo eles:

a) botão `Executar`: executa as animações especificadas no `Objeto Gráfico` selecionado no editor em um *drone* físico;
b) botão `Calibrar`: executa o calibramento dos sensores do *drone*;
c) botão `Pânico!`: utilizado durante a execução de uma sequência de animações. Enviar o comando para o *drone* pousar independentemente se tenha terminado a execução de todas as animações.

Os demais campos existentes, `Velocidade média` e `Tempo para girar 360 graus`, são utilizados para calcular o tempo em que o *drone* físico deve ficar em movimento para que o movimento seja executado como esperado. O primeiro campo indica quando tempo em segundos que o *drone* leva para percorrer uma unidade de distância utilizada no exercício. O segundo indica o tempo, também em segundos, que o *drone* deve levar para realizar uma voltar completa no seu eixo.

Figura 21 – Painel `Animação`




## 3.4 RESULTADOS E DISCUSSÕES

No decorrer desta seção serão apresentados os resultados obtidos nos testes do VisEdu-Drone. Na seção 3.4.1 é relatado o motivo da alteração dos objetivos durante o processo de desenvolvimento do VisEdu-Drone. Na seção 3.4.2 é apresentado o comparativo entre o simulador desenvolvido e os trabalhos correlatos. Na seção 3.4.3 são descritos os resultados obtidos em testes executados para avaliar a precisão dos movimentos executados pelo *drone* físico controlado pelo VisEdu-Drone e o tráfego de rede envolvendo a aplicação. Na seção 3.4.4 é descrito o problema encontrado na movimentação do *drone* e o que foi feito para tentar resolve-lo.

### 3.4.1 Objetivos

Os objetivos originais descritos na proposta deste trabalho sofreram alterações durante o desenvolvimento. Inicialmente a proposta era desenvolver uma aplicação que iria auxiliar no ensino de matemática, especificamente geometria. Porém, durante o processo de concepção do VisEdu-Drone percebeu-se que o simulador criado não atenderia todas os

requisitos necessários para auxiliar no processo educacional. Por este motivo, foi decidido mudar os objetivos inicialmente pensados.

Considerando essas alterações o VisEdu-Drone atendeu todos os objetivos propostos. Uma vez que o simulador criado estendendo o VisEdu, disponibiliza um ambiente 3D no qual o usuário tem a possibilidade visualizar animações definidas por ele, é integrado com o *framework* ROS e permitindo que as animações definidas pelo usuário no *drone* virtual sejam executadas em um *drone* físico.

### 3.4.2 Comparativo com trabalhos correlatos

No quadro 16 é apresentado um comparativo das características dos trabalhos correlatos e do trabalho proposto

Quadro 16 – Comparativo dos trabalhos correlatos

| Características | Drone Dance | Scratch | RC Drone - Quadcopter | VisEdu-Drone |
|---|---|---|---|---|
| Utiliza tecnologias web | | X | | X |
| Simulação de *drones* | | | X | X |
| Integração com *drone* físicos | X | | | X |
| Conceito de encaixe de peças | X | X | | X |
| Integrado com ROS | | | | X |

Pode ser observado que dois trabalhos possuem mais semelhanças com o VisEdu-Drone tendo duas características em comum cada. São eles o Drone Dance for AR.Drone (KAMMERER, 2013) e o Scratch (LIFELONG KINDERGARTEN GROUP, 2014). O aplicativo Drone Dance for AR.Drone (KAMMERER, 2013) tem como principal objetivo o entretenimento utilizando *drones*. Ele possui o conceito de encaixar peças e criar sequências de movimentos que devem ser executadas por um AR.Drone Parrot físico. Porém, pelo fato do aplicativo desenvolvido por Kammerer (2013) ser compatível apenas com dispositivos que rodem iOS existe uma limitação em relação ao número de usuários que podem utiliza-lo.

O Scratch assim como o simulador criado, também faz analogia a um jogo de encaixe de peças para criar sequencias de execução e também foi criado utilizando tecnologias web, o que permite que usuários de mais de uma plataforma possam utilizá-lo. A principal diferença é que o Scratch não possui integração com *drones* físicos o que limita que fluxos de execuções criados virtualmente possam ser executados por um aparelho físico.

### 3.4.3 Testes

Durante a implementação foi observado que não seria possível controlar o *drone* através de simples comandos de movimentação, onde seria dito qual direção e distância o aparelho deveria se deslocar. Para movimentar o *drone* físico é necessário enviar comandos no quais são passados valores normalizados indicando a velocidade e direção no qual o *drone* deve se movimentar. Por este motivo, foi necessário criar o método descrito na seção 3.3.3.2.2 para calcular o tempo que o *drone* deve se movimentar. Para avaliar a precisão da movimentação do AR.Drone Parrot controlado pelo VisEdu-Drone foram criados três cenários que serão descritos nas seções 3.4.3.1, 3.4.3.2 e 3.4.3.3. Além disso, na seção 3.4.3.4 são descritos os testes realizados para avaliar o tráfego de rede envolvido durante a execução do VisEdu-Drone.

Todos os testes foram feitos em ambiente fechado para evitar qualquer tipo de deslocamento causado por movimentação de ar indesejada. Todos os cenários foram executados por um *drone* controlado pelo VisEdu-Drone e por um operador através do aplicativo oficial, aqui denominado AR.FreeFlight 2.4.10, desenvolvido pela própria Parrot SA (PARROT, 2013). Durante os testes dos cenários 2 e 3 toda a configuração no painel de animação para movimentação do *drone* foi feita considerado a unidade de deslocamento de 2 metros, ou seja, uma unidade de distância do *drone* virtual corresponde a 2 metros.

#### 3.4.3.1 Cenário 1

Para realização dos testes foi preparada uma área (no caso o chão de uma sala) com dimensões de 4 metros de altura (considerado o eixo Y) por 7 metros de comprimento (considerado o eixo X) por onde o *drone* se deslocava. A área de testes pode ser vista na figura 22.

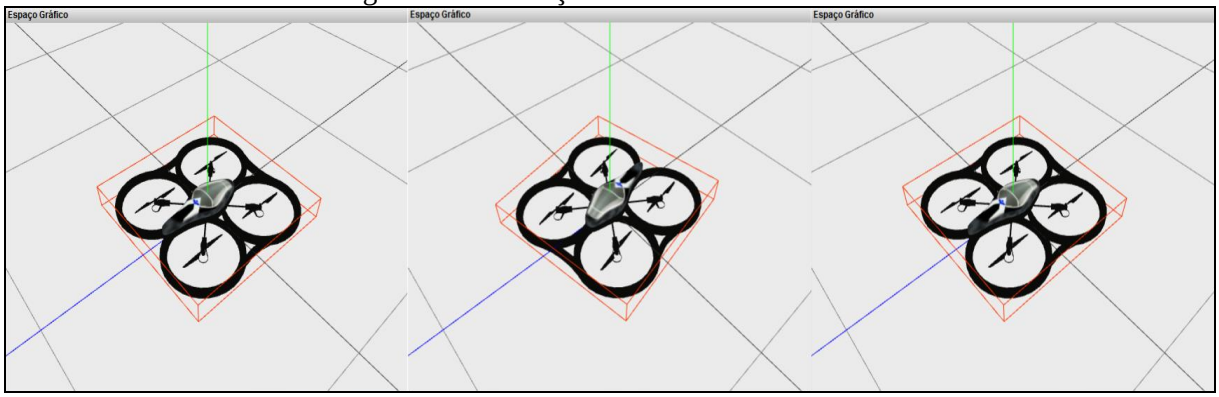
Figura 22 – Área para realizar os testes do cenário 1

No primeiro cenário criado foi avaliada a precisão do *drone* quando enviado comandos para rotação. Neste teste o *drone* foi posicionado no centro da área de testes que fica a 3,5 metros no eixo X e 2 metros no eixo Y, com a frente (lado com a câmera) voltado para o eixo Y (parede da sala). Uma vez posicionado, através do VisEdu-Drone ou do AR.FreeFlight foi enviado o comando para decolar, rotacionar 360º sentido anti-horário em seu próprio eixo e pousar. Na figura 23 é possível visualizar a fábrica de peças do VisEdu-Drone com os itens necessários para execução deste cenário. Foram adicionadas ao item `Objeto Gráfico` um item filho da forma geométrica do *`drone`* (em azul), o item `animação` (em amarelo) para habilitar as animações para esse `Objeto Gráfico` e o item da rotação (em vermelho) propriamente dita.

Figura 23 – `Fabrica de peças` para execução do cenário 1


Na figura 24 pode ser vista a execução deste cenário de testes pelo *drone* virtual. A imagem mais a esquerda apresenta o *drone* antes de começar a execução. Na imagem do meio é observado o *drone* durante a execução e na figura mais a direita pode ser visto o *drone* após terminada a animação.

Figura 24 – Execução do cenário 1 no VisEdu



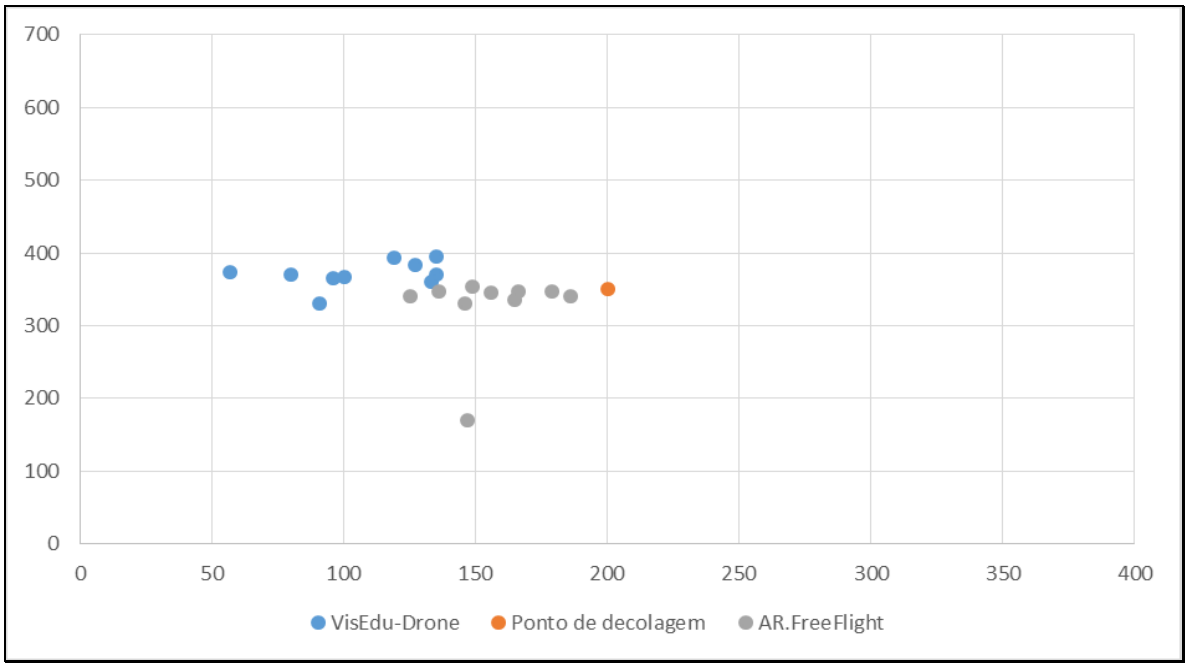

Esse teste foi executado dez vezes com o *drone* físico e os resultados obtidos com a aplicação desenvolvida e com o AR.FreeFlight podem ser observados na tabela 1.

Tabela 1 – Resultados dos testes no cenário 1

| Distância em centímetros do ponto de decolagem | | |
|---|---|---|
| | VisEdu-Drone | AR.FreeFlight 2.4.10 |
| Distância em centímetros do ponto de decolagem | 67,74 | 17,20 |
| | 68,00 | 21,09 |
| | 79,05 | 34,05 |
| | 80,11 | 38,07 |
| | 92,17 | 44,28 |
| | 101,43 | 51,08 |
| | 105,22 | 57,24 |
| | 110,64 | 64,03 |
| | 121,65 | 75,66 |
| | 144,83 | 187,64 |
| Média das distâncias | 97,08 | 59,03 |

Pode ser observado que os testes feitos com o aplicativo AR.FreeFlight, apresentaram um melhor desempenho. A distância média dos pousos em relação ao ponto de decolagem dos voos efetuados pelo AR.FreeFlight foi de 59,03 centímetros, enquanto nos voos efetuados pelo VisEdu-Drone foram de 97,08 centímetros. Além disso, a segunda maior distância registrada nos voos efetuados com o aplicativo AR.FreeFlight foi de 75,66 centímetros, enquanto nos voos controlados pelo software desenvolvido apenas dois testes apresentaram distância menor. Como pode ser observado na figura 25, apesar da diferença entre as distâncias percebe-se que em todos os testes o *drone* sempre se movimentava para uma mesma direção. Em todas as execuções o *drone* pousava mais próximo ao eixo X. O que demonstra que possivelmente o *drone* utilizado tem uma tendência a se deslocar nesta direção quando é rotacionado. Houve um único voo feito através do AR.FreeFlight que apresentou a distância de 187,64 centímetros. Neste caso, houve uma falha humana na execução do teste.

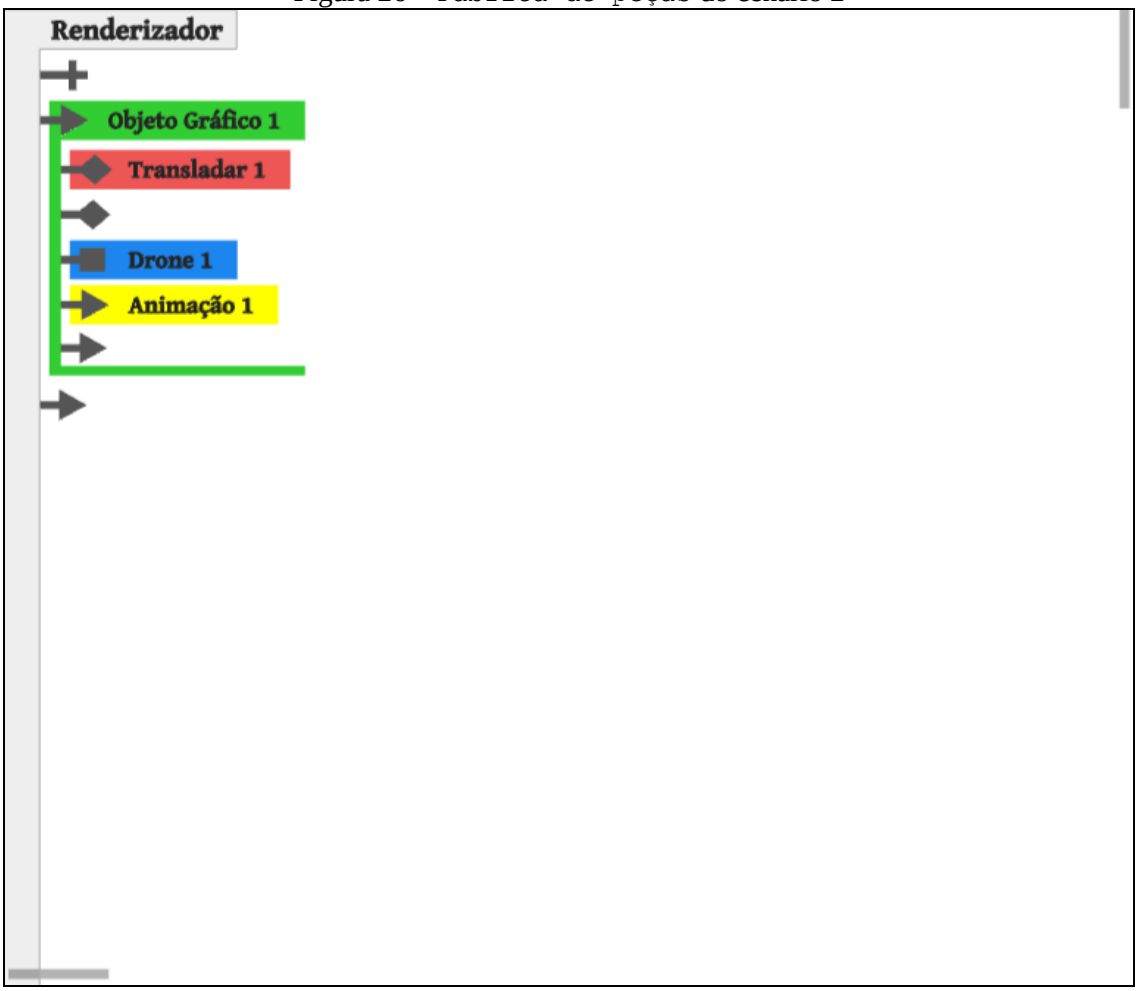
Figura 25 – Pousos realizados no cenário 1




### 3.4.3.2 Cenário 2

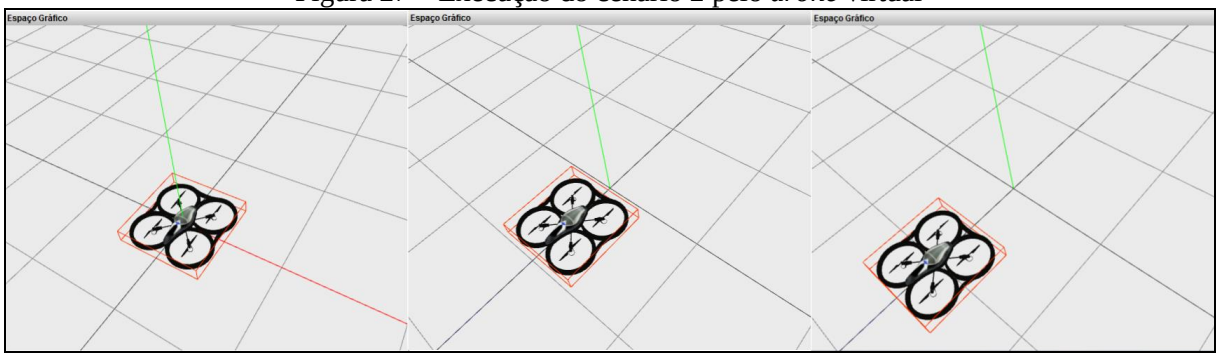
No segundo cenário foi avaliada a movimentação do *drone* em linha reta. Neste cenário o *drone* percorreu distâncias de 2, 6, e 10 metros 10 vezes cada. Todas as execuções foram realizadas em um corredor de 2,80 metros de largura com mais de 20 metros de comprimento. Para limitar as medições, o mundo no qual o *drone* poderia se movimentar possuía 2,80 metros de largura (eixo X) por 14 metros de comprimento (eixo Y). Como ponto de decolagem foi escolhido um ponto centralizado na extremidade inferior do mundo, sendo posicionado em 1,43 metros em X e 0 metro em Y. Esse ponto de decolagem foi escolhido por estar alinhado com a fuga presente no chão do corredor, facilitando a medição. Na figura 26 podem ser observadas as peças na `Fábrica de peças` utilizadas para executar os testes. Para cada distância testada a propriedade de peça translação (em vermelho) era alterada para a distância desejada.

Figura 26 – `Fábrica de peças` do cenário 2




A execução deste cenário de teste com o *drone* virtual pode ser visualizado na figura 27. Na primeira imagem observa-se o *drone* em sua posição inicial. Depois pode ser visto executando o movimento de translação. Na última imagem o *drone* já está parado na posição final depois de ser translado.

Figura 27 – Execução do cenário 2 pelo *drone* virtual




Nos testes com *drone* físico os movimentos que o operador pode fazer durante a execuções com o aplicativo AR.FreeFlight foram limitados. Durante os testes, o operador era permitido decolar, se deslocar para frente e pousar logo após o *drone* chegar no ponto de

destino. Nenhum tipo de ajuste era permitido. Os resultados obtidos podem ser vistos nas tabelas 2, 3 e 4.

Tabela 2 – Resultados dos testes no cenário 2 com distância de 2 metros

| Distância em centímetros do ponto de destino | | |
|---|---|---|
| | VisEdu-Drone | AR.FreeFlight 2.4.10 |
| Distância em centímetros do ponto de destino | 6,40 | 80,39 |
| | 10,77 | 100,00 |
| | 13,00 | 100,17 |
| | 13,15 | 115,10 |
| | 14,03 | 124,32 |
| | 15,23 | 125,06 |
| | 15,26 | 140,06 |
| | 19,31 | 157,31 |
| | 25,07 | 162,43 |
| | 26,07 | 214,59 |
| Média das distâncias | 15,82 | 131,94 |

Voos nos quais o *drone* percorria dois metros controlado pela aplicação desenvolvida obtiveram um melhor desempenho dos que foram controlados por um operador nesta mesma distância. Acredita-se que o motivo do melhor desempenho do VisEdu é que o *drone* é parado no momento correto para ficar o mais próximo possível do destino após se estabilizar, enquanto voos controlados por uma pessoa possuem um tempo de resposta maior, devido que o *drone* só é parado quando o operador percebe que ele está sobre o destino.

Os testes nas demais distâncias demonstraram um baixo desempenho do VisEdu-Drone, como pode ser visto na tabela 3. O desempenho da aplicação desenvolvida foi inferior comparado com os primeiros testes em uma distância de dois metros. Quando submetido a testes em uma distância de seis metros o *drone* controlado pela aplicação VisEdu-Drone parou sempre entre 5 e 6 metros do ponto de destino. Não foram realizados testes de voo em distâncias acima de seis metros com o VisEdu-Drone porque em todas as tentativas o *drone* parava fora da área de testes e com risco de colidir com as paredes.

Tabela 3 – Resultados dos testes no cenário 2 com distância de 6 metros

| Distância em centímetros do ponto destino | | |
|---|---|---|
| | VisEdu-Drone | AR.FreeFlight 2.4.10 |
| Distância em centímetros do ponto destino | 505,39 | 175,51 |
| | 541,95 | 189,93 |
| | 542,44 | 211,09 |
| | 545,15 | 223,02 |
| | 549,35 | 229,51 |
| | 570,00 | 233,99 |
| | 571,93 | 244,84 |
| | 587,04 | 252,17 |
| | 600,00 | 252,34 |
| | 600,00 | 273,65 |
| Médias das distâncias | 561,32 | 228,61 |

O problema encontrado é causado pelo fato de que no cálculo do tempo no qual o *drone* deve se manter em movimento não está considerando a velocidade atual do *drone*, ou seja, a medida que o *drone* se desloca a velocidade aumenta e consequentemente o tempo necessário para percorrer determinada distância diminui. Na seção 3.4.4 o problema encontrado é relatado com mais detalhes e são demonstradas algumas soluções que foram aplicadas tentando resolver essa deficiência.

O desempenho em seis e dez metros nos voos controlados pelo aplicativo oficial do Parrot SA apresentaram resultados semelhantes aos testes realizados com dois metros. Como pode ser observada nas tabelas 2, 3 e 4, a média de distância em relação ao ponto de destino para voo de 2 metros foi 1,31 metros, 6 metros foi 2,28 metros e 2,79 metros para 10 metros.

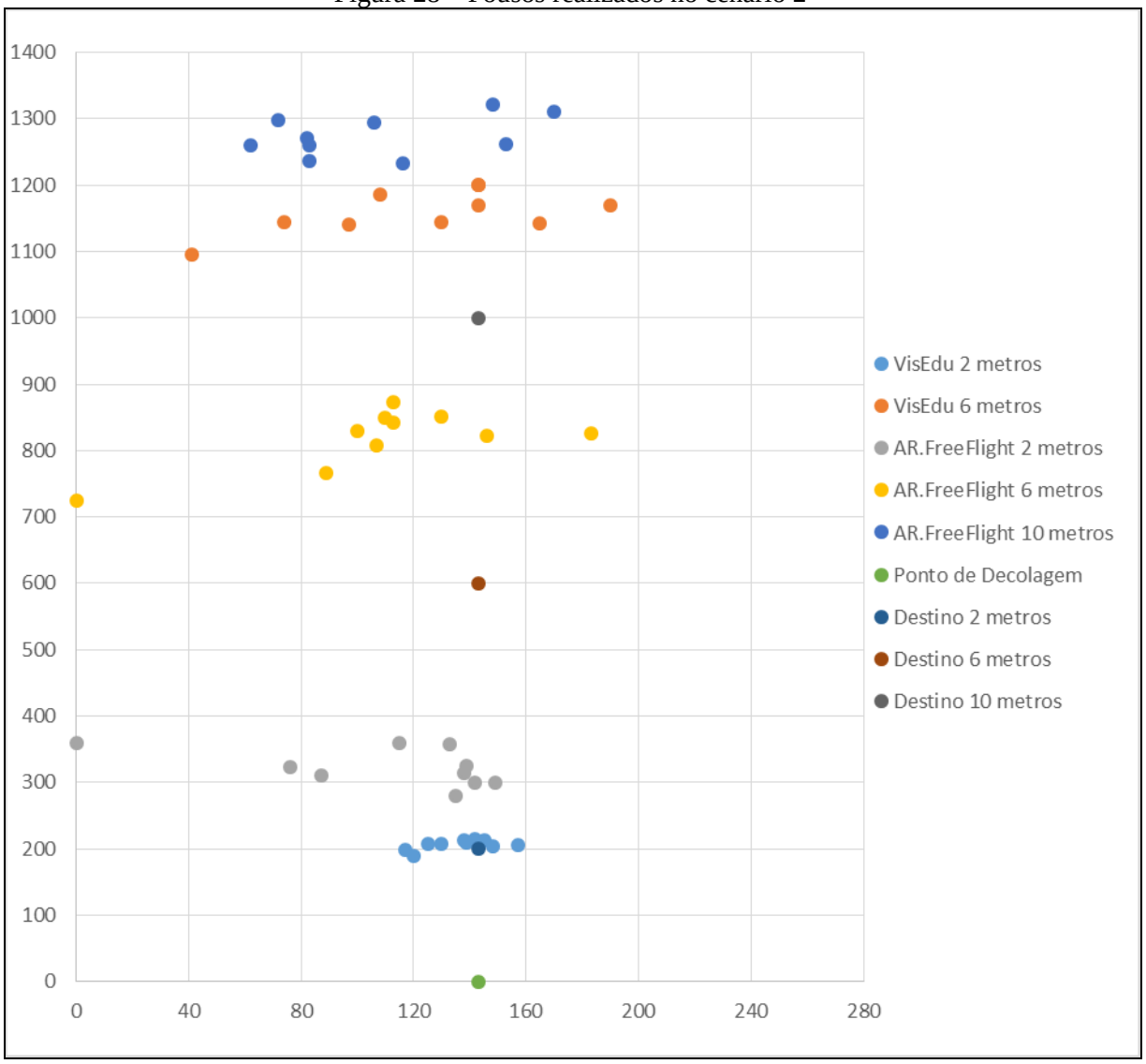
De forma similar aos testes na menor distância os testes com distância maiores apresentaram a mesma dificuldade pelo fator humano, onde o tempo de resposta para parar o *drone* após ele ter alcançado o destino é maior quando comparado a voos controlados pela aplicação.

Tabela 4 – Resultados dos testes no cenário 2 com distância de 10 metros com o AR.FreeFlight

| | AR.FreeFlight |
|---|---|
| Distância em centímetros do ponto de destino | 233,56 |
| | 244,47 |
| | 262,19 |
| | 265,85 |
| | 272,32 |
| | 277,78 |
| | 297,31 |
| | 305,36 |
| | 311,17 |
| | 322,03 |
| Média das distâncias | 279,20 |

Na figura 28 pode ser visualizado onde o *drone* pousou dentro da área de testes em cada teste executado.

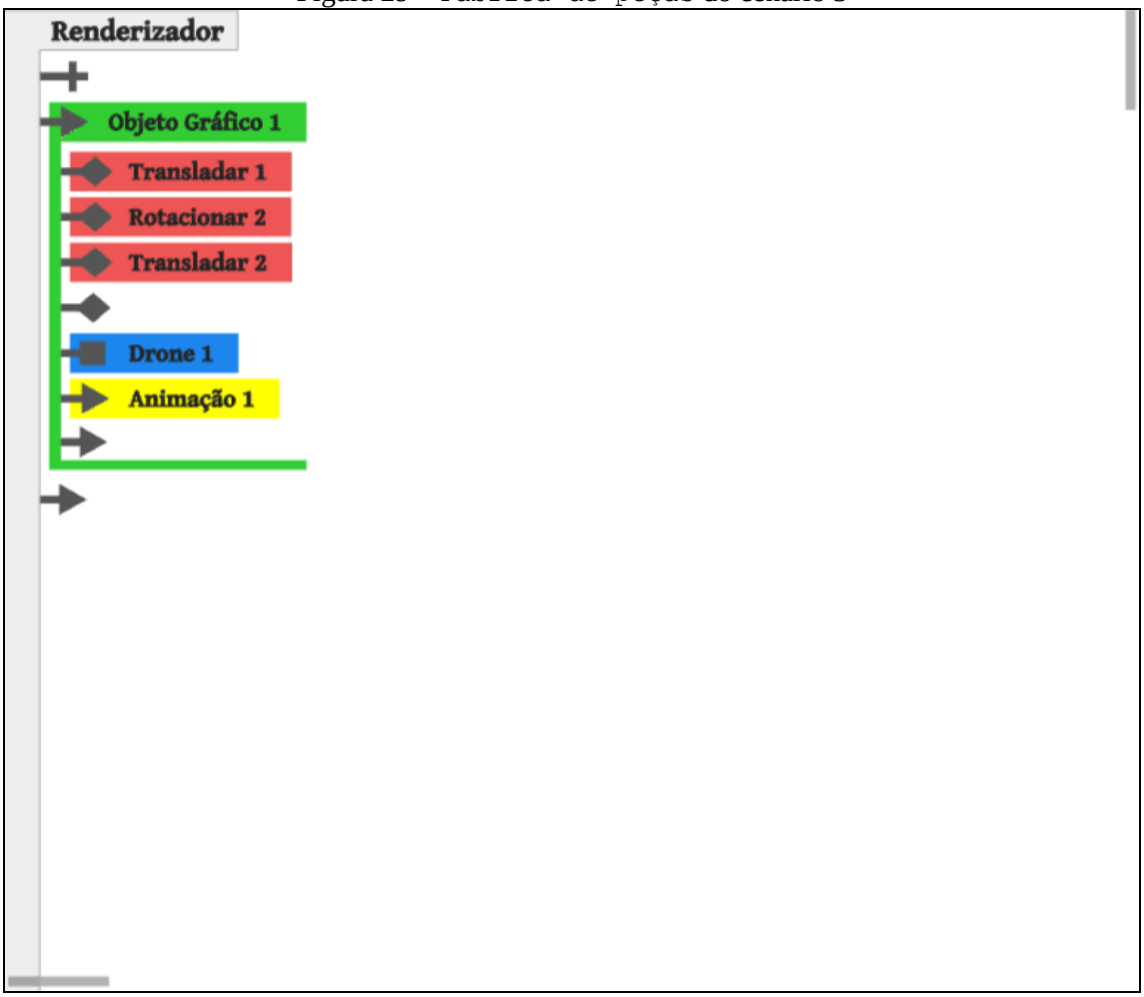
Figura 28 – Pousos realizados no cenário 2




### 3.4.3.3 Cenário 3

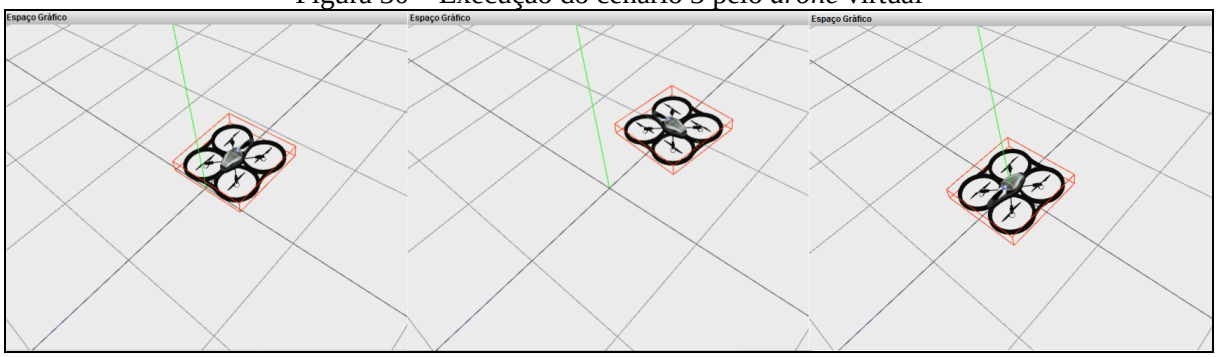
No terceiro e último cenário de teste foi avaliado o desempenho do *drone* quando movimentado e rotacionado. Neste cenário o *drone* era permitido decolar, se deslocar uma unidade de distância no eixo X, rotacionar 180º, deslocar uma unidade de distância no sentido contrário da primeira movimentação no eixo X e pousar. Desta forma o destino fica na mesma posição da origem após o *drone* completar os testes. A `Fábrica de peças` do VisEdu-Drone com as interações para este cenário pode ser vista na figura 29.

Figura 29 – `Fábrica de peças` do cenário 3




A execução deste cenário pelo *drone* virtual pode ser visualizada na figura 30. Na imagem do *drone* mais a esquerda pode ser observada a execução da primeira translação. A imagem do meio corresponde a animação de rotação e a última é o *drone* ao final da execução.

Figura 30 – Execução do cenário 3 pelo *drone* virtual



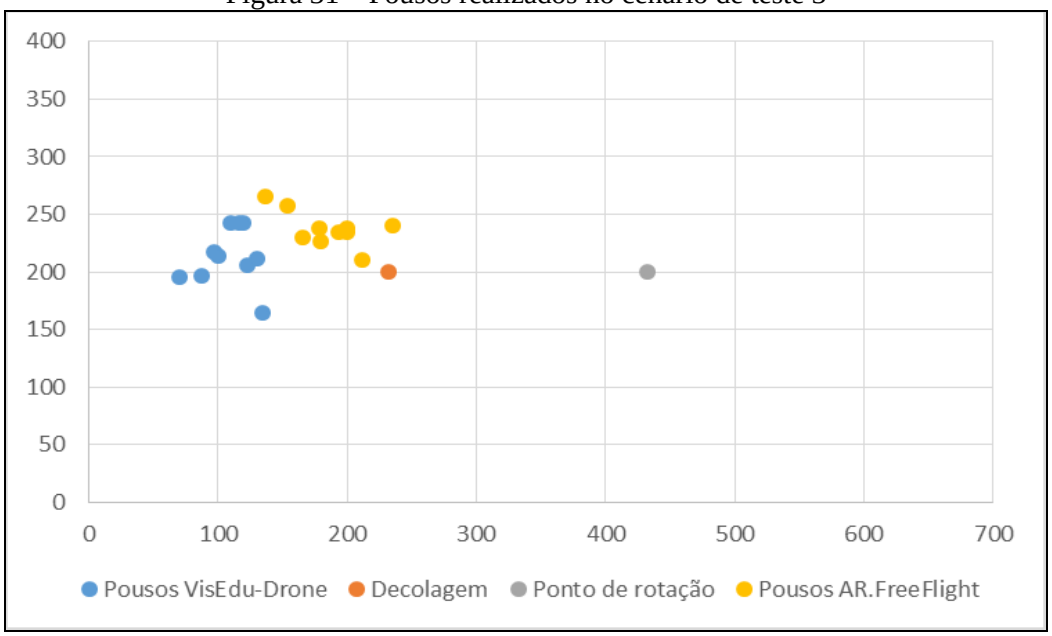

Para executar esses testes com o *drone* físico foi utilizado a mesma área de testes do cenário 1 (figura 22). Porém, o ponto de decolagem e pouso ficaram posicionados no ponto (232, 200). Na tabela 5 pode ser observado qual foi o desempenho do VisEdu-Drone e do AR.FreeFlight no terceiro cenário.

Tabela 5 – Resultados dos testes no cenário de teste 3

| Distância em centímetros do ponto de decolagem | | |
|---|---|---|
| Distância em centímetros do ponto de decolagem | VisEdu-Drone | AR.FreeFlight 2.4.10 |
| | 102,70 | 22,36 |
| | 103,46 | 40,11 |
| | 109,16 | 47,42 |
| | 119,61 | 49,67 |
| | 123,36 | 51,73 |
| | 129,35 | 58,59 |
| | 132,74 | 66,03 |
| | 136,06 | 72,49 |
| | 145,03 | 96,60 |
| | 162,07 | 115,67 |
| Média das distâncias | 126,35 | 62,06 |

Observa-se que como nos testes realizados no primeiro cenário, que o aplicativo AR.FreeFlight possui uma maior estabilidade, principalmente quando são envolvidos movimentos de rotação. O aplicativo desenvolvido pela Parrot SA obteve uma distância média do ponto de decolagem de 62,06 centímetros, que é menos da metade da distância média dos testes realizados pelo VisEdu-Drone. Levando em consideração que a aplicação desenvolvida não utilizada praticamente nenhum sensor disponível no *drone* físico para planejar e realizar a movimentação, acredita-se que o melhor desempenho apresentado pelo AR.FreeFlight deve-se principalmente a um sistema de navegação que utiliza de forma apropriada dos sensores. Na figura 31 é apresentado graficamente onde o *drone* físico pousou ao final de cada teste.

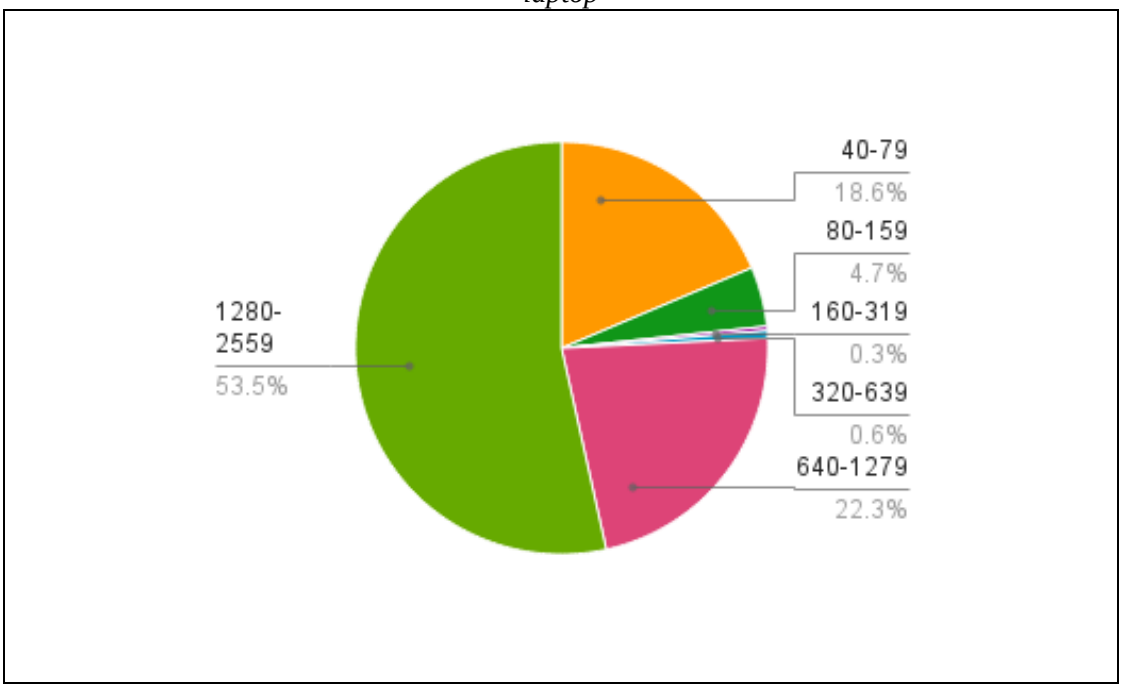
Figura 31 – Pousos realizados no cenário de teste 3



### 3.4.3.4 Tráfego de rede

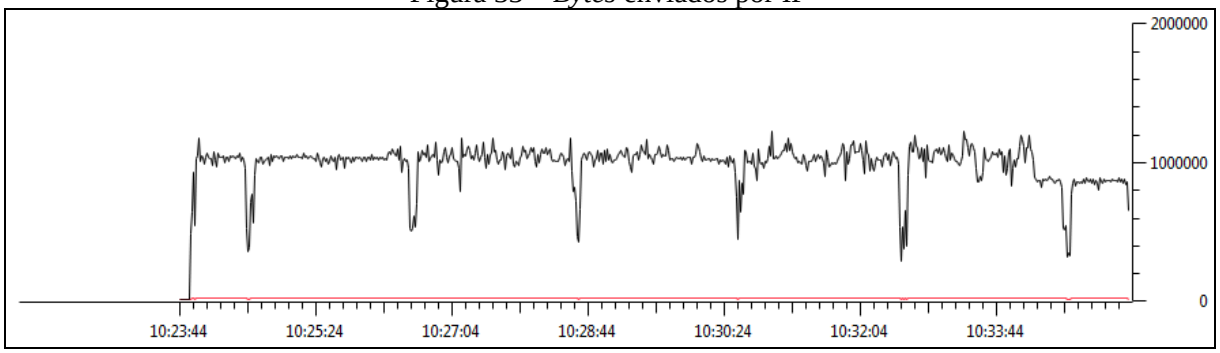
Para verificar o desempenho da aplicação no que diz respeito à utilização de tráfego de rede foi realizada uma análise da troca de pacotes de rede dentro do grafo do ROS. Também é analisado o tráfego de rede entre um *laptop* e o AR.Drone Parrot. Para coletar essas informações foi utilizada a ferramenta Wireshark em sua versão 1.12.5. Para obter os dados necessários foram monitoradas as interfaces de rede de um *laptop* durante a utilização do VisEdu-Drone enquanto interagia com o *drone* AR.Drone Parrot em um período de doze minutos. Durante esse tempo não foi utilizado nenhum outro tipo de aplicação que utilizasse de rede. Com os dados obtidos foi possível gerar os gráficos das figuras 32, 33 e 34.

Figura 32 – Gráfico demonstrando o tamanho em bytes dos pacotes transferidos entre o *drone* e o *laptop*




O gráfico que pode ser visto na figura 32 mostra o tamanho em *bytes* dos pacotes que transitaram entre o *laptop* e o *drone* ao qual ele estava conectado. Cada fatia representa uma faixa de tamanho dos pacotes que foram analisados. Como pode ser observada nas partes verde claro e rosa, a maioria (75,8%) dos pacotes transferidos entre o *drone* e o computador no qual estava sendo executado o VisEdu-Drone ficou entre 640 e 2.559 *bytes*. A terceira maior parcela, destacada em amarelo, condiz com 18,6% e os pacotes no qual representa possuíam entre 40 e 79 *bytes*. Do restante, 4,7% era de tamanho 80 a 159 *bytes* e 0.3% ficaram na faixa de 160 a 319 *bytes*, o que demonstra que a comunicação entre o *drone* e o *laptop* é realizada através de pequenos pacotes de rede. Com o intuito de identificar qual era a maior fonte de pacotes de rede foi gerado o gráfico que pode ser visto na figura 33.

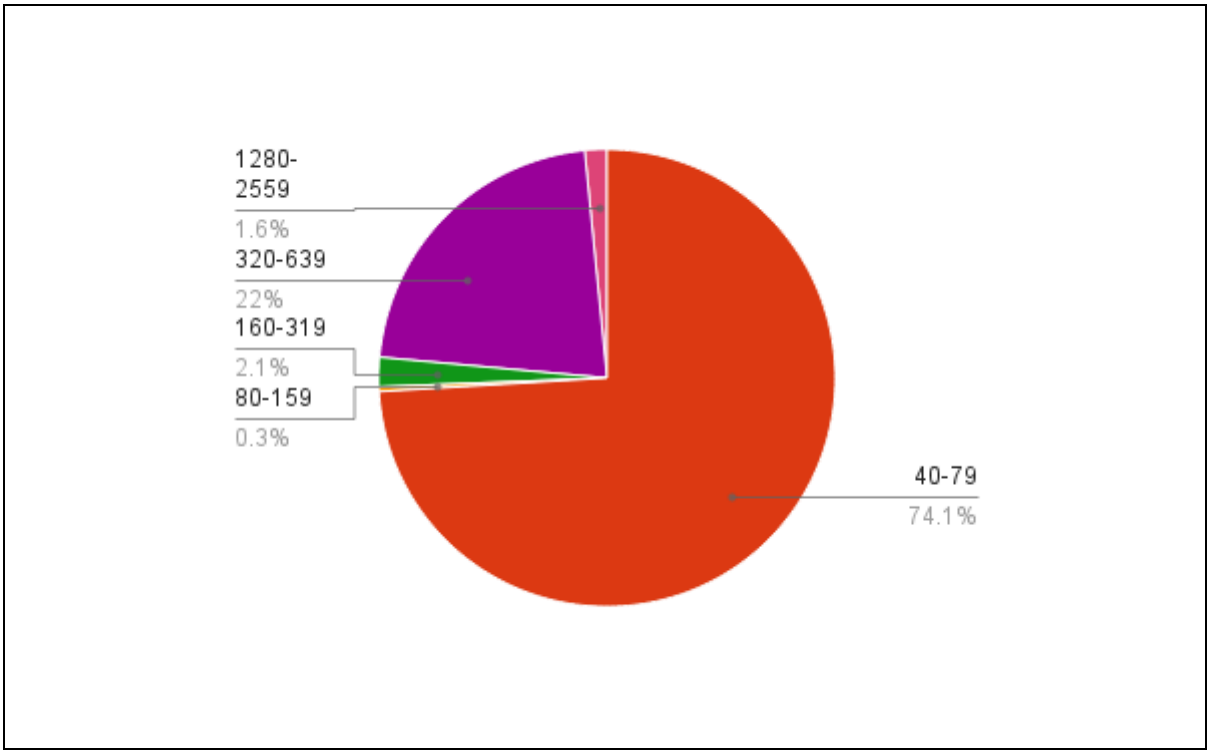
Figura 33 – *Bytes* enviados por IP




No gráfico podem ser observadas duas linhas (cores preta e vermelha). Cada uma delas representa a quantidades de *bytes* enviados de um IP na rede, ou seja, pacotes originados do *drone* ou do *laptop*. A linha vermelha representa o tráfico de rede gerado do computador monitorado para o *drone*. Nela observa-se que o tráfico gerado permaneceu praticamente inalterado. Enquanto a linha preta, que representa o tráfego gerado do *drone* para o computador, apresenta variações bem visíveis. Observou-se que os momentos nos quais o *drone* apresentou uma baixa transmissão de pacotes corresponde aos períodos de tempo nos quais ele estava parado.

A figura 34 demonstra o tamanho dos pacotes transferidos entre os nós do grafo do ROS. Igual ao gráfico da figura 32, cada fatia representa uma faixa de tamanho dos pacotes que foram analisados durante o teste.

Figura 34 – Gráfico demonstrando o tamanho em bytes dos pacotes transferidos dentro do grafo do ROS



As aplicações que estavam sendo executadas dentro do grafo no momento que os dados foram coletados eram o servidor ROS, `rosbridge` e o *driver* para AR.Drone Parrot. Todas elas estavam sendo executadas na mesma máquina. Como observado no gráfico da figura 34, os tamanhos dos pacotes de rede oscilaram entre 40 e 2.559 *bytes*, sendo que a maioria deles (96,1%), destacados em vermelho e roxo, era de tamanho entre 320 a 639 e 40 a 79 *bytes*. Os maiores pacotes transferidos tiveram tamanho entre 1.280 e 2.559 *bytes* e representam 1,6% do montante total de pacotes. Os 2,4% restante apresentaram tamanho menor do que 159 *bytes*, o que demonstra que assim como os pacotes transferidos entre o *laptop* e o *drone* a grande maioria dos pacotes transferidos dentro do grafo possui um tamanho reduzido e consequentemente o tempo para sua transferência também é pequeno.

### 3.4.4 Deficiência na movimentação

Durante os testes descritos na seção 3.4.3 foi detectada uma deficiência no software desenvolvido no que se diz respeito à movimentação do *drone*. Quando o *drone* se deslocava distâncias mais longas, o algoritmo de cálculo de tempo necessário para realizar a movimentação não se mostrou eficiente e que precisava de ajustes. No primeiro momento, a primeira atitude tomada foi descobrir sua causa, depois estudar uma possível solução e tentar aplicá-la.

O problema detectado ocorre porque o algoritmo responsável por esse cálculo não considera a velocidade e aceleração do *drone*. Assim, conforme o *drone* se desloca sua velocidade aumenta e o tempo necessário para percorrer a distância desejada diminui, causando a deficiência na precisão da movimentação. Inicialmente foi buscado resolver o problema através de algum pacote existente no ROS. Foi encontrado o `tum_ardrone` (TUM COMPUTER VISION GROUP, 2015), que se trata de um pacote que permite voos autônomos para AR.Drone Parrot baseados em navegação visual (TUM COMPUTER VISION GROUP, 2015). Porém, após uma análise mais aprofundada notou-se que o pacote não permitia que os comandos de navegação fossem enviados diretamente sem intervenção de um software adicional disponibilizado pelo próprio pacote. Apesar do `tum_ardrone` fornecer a solução ideal para o problema encontrado as alterações e possíveis customizações necessárias para sua utilização tornaria seu uso inviável, visto que não haveria tempo hábil para realizar todas as implementações necessárias. Por esse motivo buscou-se uma segunda opção.

A segunda solução encontrada foi utilizar os dados estimados fornecidos pelo odômetro disponibilizado pelo *driver* do *drone* para calcular a movimentação efetuada. Essa

implementação foi feita, mas o resultado ainda não foi satisfatório e não resolveu os problemas da movimentação, principalmente pelo fato que os dados estimados fornecidos pelo *drone* requerem um tratamento antes de serem utilizados, compensando leituras errôneas ou afetadas por fatores ambientais, possíveis falhas ou erros acumulados dos sensores. Por isso, optou-se em manter a implementação original baseado no tempo de movimentação e foram criadas duas extensões para resolver o problema encontrado. Uma delas sugere a integração do VisEdu-Drone com o `tum_ardrone` e o outra propõem a criação de um sistema de navegação se utilizando de forma mais adequada dos dados retornados pelo o odômetro. Além disso, foi adicionada outra extensão para realizar testes mais aprofundados comparando o VisEdu-Drone com outro software similar e não mais com o *drone* sendo controlado por um operador através do AR.FreeFlight. Essa extensão foi proposta pensando em realizar uma avaliação entre dois algoritmos e não mais uma pessoa e um algoritmo e verificando se a deficiência encontrada se repete em outros softwares.

# 4 CONCLUSÕES

O principal objetivo do trabalho, estender o VisEdu na criação de um simulador de *drone* integrado com o *framework* ROS foi atendido. Além disso, todos os objetivos específicos foram atendidos, pois o VisEdu foi utilizado como base para a implementação. O usuário pode controlar um *drone* em um ambiente de realidade virtual e envia comandos para um *drone* físico através do ROS. Considerando que o objetivo do projeto VisEdu é ser utilizado como ferramenta de auxílio no processo educacional, a falta de precisão quando interagido com o *drone* AR.Drone Parrot ocasiona problemas se desejado a criação de uma aplicação para esse propósito. Por isso, foi incluído como extensão do trabalho a implementação de um sistema de navegação mais preciso aproveitando melhor os sensores existentes no aparelho, ou ainda, a integração com algum pacote existente na plataforma ROS que realize essa tarefa.

De modo geral, as ferramentas utilizadas para a criação do VisEdu-Drone se mostraram ser adequadas. Utilizar o VisEdu como base para implementação mostrou-se uma boa escolha pois agilizou o processo de desenvolvimento. Uma vez que já havia todo o ambiente necessário para renderização 3D, não foram encontradas grandes dificuldades para inclusão de novos itens e ajustes nos que já existiam no projeto. Além disso, o mesmo já utilizava as bibliotecas Three.js e Tween.js que foram as principais bibliotecas utilizadas para manipulação do espaço 3D, abstração do WebGL e criação das animações.

Os pacotes do *driver* para AR.Drone e o `rosbridge` existentes para plataforma ROS utilizados para comunicação com o *drone* bem como a biblioteca Javascript `roslibjs` também se mostraram de grande valia, pois possibilitaram de forma eficiente a comunicação entre o simulador rodando no navegador de Internet e o servidor ROS utilizado para controle do AR.Drone Parrot 2.0.

Um dos principais pontos fortes e contribuição ao projeto do VisEdu da extensão desenvolvida foi a utilização do *framework* ROS. Por se tratar de uma plataforma genérica para desenvolvimento de aplicações voltadas para robótica já possui diversos módulos que podem ser adicionados ao VisEdu, ampliando as opções de robôs ou *drones* que podem ser integrados ao projeto sem ser necessário realizar grandes alterações na estrutura atual da aplicação. Entretanto ainda são necessárias melhorias relacionadas ao controle de acesso ao servidor ROS onde qual é feita a comunicação com o *drone*, evitando que dois usuários possam tentar interagir com o *drone* ao mesmo tempo.

## 4.1 EXTENSÕES

Nesta seção são listadas extensões que podem ser feitas para melhorar o VisEdu-Drone. São elas:

a) integrar o VisEdu-Drone com o `tum_ardrone`;
b) implementar um sistema de navegação tratando adequadamente e utilizando os dados do odômetro;
c) criar um pacote no ROS para encapsular o controle do *drone* ou qualquer outro tipo de robô controlado pelo VisEdu-Drone;
d) remover as configurações de tempo necessárias para utilizar o *drone* físico;
e) criar um visualizador dos status do *drone*. Exibindo dados retornados pelo *drone* como posição estimada, entre outros;
f) implementar reconhecimento da marcadores;
g) realizar testes comparando o VisEdu-Drone com outra aplicação similiar. Identificar se os problemas detectados durante os testes ocorrem com softwares similares;
h) melhorar tratamento entre unidade de distância do *drone* virtual e unidade de distância do *drone* físico.

# REFERÊNCIAS

CARRIJO, Gilberto A. et al. **Associando realidade virtual ao ensino de matemática fundamental**. Uberlândia, 2007. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/svr/2007/0050.pdf>. Acesso em: 24 ago. 2014.

EDX. **AUTONAVx autonomous navigation for flying robots.** [Monique], 2014. Disponível em: < https://autonav.in.tum.de/handouts/lecture_1_part_1.pdf >. Acesso em: 16 set. 2014.

GRUPO DE PESQUISA EM COMPUTAÇÃO GRÁFICA, PROCESSAMENTO DE IMAGENS E ENTRETERIMENTO DIGITAL. **VisEdu:** Visualizador de material educacional. [Blumenau], 2014. Disponível em: < http://gcg.inf.furb.br/?page_id=1147 >. Acesso em: 30 out. 2014.

KAMMERER, Tommy. **Drone Dance for AR.Drone**. [S.l.], 2013. Disponível em: < https://itunes.apple.com/br/app/drone-dance-for-ar.drone/id433120773?l=en&mt=8 >. Acesso em: 28 out. 2014.

KHRONOS GROUP. **Getting Started – WebGL Public Wiki.** [S.l.], 2011. Disponível em: < http://www.khronos.org/webgl/wiki/Getting_Started >. Acesso em: 17 set. 2014.

_____. **WebGL**. [S.l.], 2014. Disponível em: <https://www.khronos.org/registry/webgl/specs/1.0/#1>. Acesso em: 11 jul. 2015

KRAUSS, José R. **VISEDU-Mat:** Visualizador de material educacional, módulo de matemática. 2013. 64 f. Trabalho de Conclusão de Curso (Bacharel em Ciência da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau, Brasil.

LIFELONG KINDERGARTEN GROUP, **Scratch**. [S.l], 2014. Disponível em: < www.scratch.mit.edu/about >. Acesso em: 28 set. 2014.

_____. **ScratchEd**. [S.l], 2015. Disponível em: < http://scratched.gse.harvard.edu/about >. Acesso em: 11 jul. 2015.

LIVE SCIENCE, **9 totally cool uses for drones.** [S.l.], 2013. Disponível em: <http://www.livescience.com/28137-cool-uses-for-drones.html>. Acesso em: 08 jul. 2015.

MONTIBELLER, James P. **VISEDU-CG:** Aplicação didática para visualizar material educacional, módulo de computação gráfica. 2014. 106 f. Trabalho de Conclusão de Curso (Bacharel em Ciência da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau, Brasil.

MOZILLA FOUNDATION. **Javascript Overview**. [S.l.], 2014a. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/JavaScript_Overview>. Acesso em: 9 set. 2014.

NUNES, Samuel A. **VISEDU-CG 3.0:** Aplicação didática para visualização material educacional, módulo de computação gráfica. 2014. 89 f. Trabalho de Conclusão de Curso (Bacharel em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau, Brasil.

OPEN SOURCE ROBOTICS FOUNDATION. **ROS concepts.** [S.l.], 2014a. Disponível em: <http://wiki.ros.org/ROS/Concepts>. Acesso em: 06 maio 2015

_____. **ROS connection header.** [S.l.], 2011. Disponível em: <http://wiki.ros.org/ROS/Connection Header>. Acesso em: 06 maio 2015.

_____. **ROS introduction.** [S.l.], 2014b. Disponível em: <http://wiki.ros.org/ROS/Introduction>. Acesso em: 06 maio 2015.

_____. **ROS messages.** [S.l.], 2012a. Disponível em: <http://wiki.ros.org/Messages>. Acesso em: 06 maio 2015.

_____. **ROS packages.** [S.l.], 2014c. Disponível em: <http://wiki.ros.org/Packages>. Acesso em: 06 maio 2015.

_____. **ROS services.** [S.l.], 2012b. Disponível em: <http://wiki.ros.org/Services>. Acesso em: 06 maio 2015.

_____. **ROS technical overview.** [S.l.], 2014d. Disponível em: <http://wiki.ros.org/ROS/Technical Overview>. Acesso em: 06 maio 2015.

_____.**ROS Master**.[S.l], 2012c. Disponível em: <http://wiki.ros.org/Master>. Acesso em: 20 maio 2015.

_____.**ROS Topics**.[S.l], 2014e. Disponível em: <http://wiki.ros.org/Topics>. Acesso em: 20 maio 2015.

PARROT. **AR.Drone 2.0 Parrot.** [Paris], 2014. Disponível em: <http://ardrone2.parrot.com/>. Acesso em: 16 set. 2014.

_____. **AR.FreeFlight 2.4.10**. [Paris], 2013. Disponível em: <https://play.google.com/store/apps/details?id=com.parrot.freeflight>. Acesso em: 23 jun. 2015.

POLYESTERGAMES PTY. LTD. **RC Drone – Quadcopter**. [S.l.], 2014.Disponível em: <https://itunes.apple.com/br/app/rc-drone-quadcopter/id713872886?mt=8>. Acesso em: 4 set. 2014.

TUM COMPUTER VISION GROUP. **Visual Navigation for the Parrot AR.Drone**. [Munique], 2015. Disponível em: <http://vision.in.tum.de/data/software/tum_ardrone>. Acesso em: 19 jun. 2015.

TWEEN. tween.js user guide.[S.l.], 2014. Disponivel em: <https://github.com/tweenjs/tween.js/blob/master/docs/user_guide.md>. Acesso em: 14 jun. 2015

UNMANNED AERIAL VEHICLE SYSTEMS ASSOCIATION, **Advantages of UAS**. [S.l.], 2015. Disponível em: <http://www.uavs.org/advantages>. Acesso em: 08 jul. 2015

W3C. **HTML5**. [S.l.], 2015. Disponível em: <http://www.w3.org/TR/html5/>. Acesso em: 11 jul. 2015.

W3C SCHOOLS. **HTML5 Canvas**. [S.l.], 2015. Disponível em: <http://www.w3schools.com/html/html5_canvas.asp>. Acesso em: 16 jun. 2014.

_____. **HTML5 Introduction**. [S.l.], 2014b. Disponível em: <http://www.w3schools.com/html/html5_intro.asp>. Acesso em: 10 set. 2014.

_____. **Javascript introduction**. [S.l.], 2014a. Disponível em: <http://www.w3schools.com/js/js_intro.asp>. Acesso em: 9 set. 2014.

## APÊNDICE A – Detalhamento dos novos casos de uso e dos alterados

Neste apêndice são descritos os casos de usos que tiveram alterações para ficarem de acordo com as extensões realizadas pelo trabalho desenvolvido. Além disso, está detalhado dois novos casos de usos que contemplam as novas características adicionadas a aplicação. Os quadros 17, 18, 19, 20, 21 e 22 descrevem os casos de uso que sofreram alterações durante o desenvolvimento do VisEdu-Drone. Enquanto os quadros 23 e 24 são os novos casos de usos adicionados durante o desenvolvimento teste trabalho.

Quadro 17 – Detalhamento do caso de uso UC01

| UC01 – Criar novas peças para construção do exercício | |
|---|---|
| Descrição | As peças do exercício são aquelas que são encaixadas no `painel de montagem` para possibilitar a geração da cena 3D e do código fonte. Cada peça representa um comando ou conceito de computação gráfica, como por exemplo: o desenho de um cubo, polígono ou spline, um objeto gráfico (empilhamento e desempilhamento de matrizes de transformação), as transformações de escala, rotação e translação, ou ainda a iluminação, animação de cena e a visão de uma câmera em um cenário 3D ou 2D. O painel `Fábrica de Peças` é composto por dois sub painéis: a `fábrica` e o `painel de montagem`. O painel `fábrica` possui todas as peças que podem ser utilizadas pelo usuário e o `painel de montagem` apresenta um ambiente para realizar a montagem do exercício através do encaixe das peças. O Ele possui uma peça principal (`Renderizador`) que disponibiliza encaixes para as peças `Câmera`, `Iluminação` e `Objeto Gráfico`, permitindo a inicialização do exercício.<br>A peça `Objeto Gráfico` irá disponibilizar encaixes internos para peças filhas, com encaixes para as peças `Cubo`, `Polígono`, `Spline`, `Drone`, `Destino`, `Transladar`, `Rotacionar`, `Escalar`, `Iluminação`, `Animação` e `Objeto Gráfico`. |
| Cenário Principal | O `Usuário` clica com o botão esquerdo do mouse em uma peça da `fábrica` e a arrasta para um dos encaixes livres compatíveis, dentro do `painel de montagem`, mantendo o botão do mouse pressionado. |
| Cenário alternativo 1 | Se no passo 1 do cenário principal o `Usuário` soltar a peça dentro da `fábrica`, a peça sendo arrastada será excluída. |
| Cenário alternativo 2 | Se no passo 1 do cenário principal o `Usuário` soltar a peça dentro do `painel de montagem`, fora de um encaixe compatível, a peça sendo arrastada ficará na posição que foi solta. |
| Pós-condição | O `Usuário` deve poder visualizar a nova peça encaixada no `painel de montagem`. A peça deverá ter um nome automático, conforme um contador interno de inclusão de peças do mesmo tipo. Peças encaixadas alteram o resultado gráfico exibido na cena 2D/3D do exercício. O `Usuário` poderá visualizar a peça conforme o cursor do mouse é movimentado e a peça deve acompanhar o cursor. |

Quadro 18 – Detalhamento do caso de uso UC02

| UC02 – Reencaixar ou encaixar peças existentes no painel de montagem | |
|---|---|
| Descrição | As peças já encaixadas no `painel de montagem` podem ser movidas para outros encaixes compatíveis com o encaixe da peça, alterando assim o resultado obtido pelo exercício. Também é possível encaixar peças que foram soltas no `Painel de Montagem` e que não estão encaixadas em nenhum encaixe. Essas peças não encaixadas não afetam o resultado do exercício até serem encaixadas em um encaixe. |
| Pré-condição | UC01 |
| Cenário Principal | O `Usuário` clica com o botão esquerdo do mouse em uma peça do `Painel de Montagem` e a arrasta para um dos encaixes livres compatíveis, mantendo o botão do mouse pressionado. |
| Cenário alternativo 1 | Se no passo 1 do cenário principal o `Usuário` soltar a peça dentro do painel `Fábrica`, a peça sendo arrastada será excluída. |
| Cenário alternativo 2 | Se no passo 1 do cenário principal o `Usuário` soltar a peça dentro do `painel de montagem`, fora de um encaixe compatível, e ela não estiver encaixada em outra peça (flutuando no painel), a peça sendo arrastada ficará na posição que foi solta. |
| Cenário alternativo 3 | Se no passo 1 do cenário principal o `Usuário` soltar a peça dentro do `Painel de Montagem`, fora de um encaixe compatível, e ela estiver encaixada em outra peça, a peça sendo arrastada continuará encaixada no encaixe de origem e o movimento da peça será cancelado. |
| Pós-condição | O `Usuário` deve poder visualizar a peça posicionada no seu novo encaixe no `Painel de Montagem`. Peças encaixadas alteram o resultado gráfico exibido na cena 2D ou 3D do exercício. O usuário poderá visualizar a peça conforme o cursor do mouse é movimentado e a peça deve acompanhar o cursor. |

Quadro 19 – Detalhamento do caso de uso UC03

| UC03 - Selecionar objeto gráfico | |
|---|---|
| Descrição | Através da seleção dos objetos gráficos o `Usuário` tem acesso as propriedades do mesmo no painel `Propriedades da Peça`. |
| Pré-condição | UC01. |
| Cenário Principal | O `Usuário` clica com o botão esquerdo do mouse em uma peça do `Painel de Montagem`. |
| Cenário alternativo 1 | O `Usuário` clica com o botão esquerdo do mouse sobre o nome da peça na `Lista de Peças`. |
| Cenário alternativo 2 | O `Usuário` realiza duplo clique com o botão esquerdo do mouse no objeto gráfico diretamente no `Espaço Gráfico`. |
| Pós-condição | No `Painel de Montagem` a peça terá uma coloração diferente das demais peças. O painel |

| | |
|---|---|
| | `Propriedades da Peça` será exibido sobre o painel `fábrica`, exibindo a(s) propriedade(s) da peça selecionada. O painel `Comandos em JOGL` será atualizado com o código fonte da peça, exceto para as peças `drone`, `destino` e `animação`. |

Quadro 20 – Detalhamento do caso de uso UC04

| UC04 - Visualizar código JOGL da peça selecionada | |
|---|---|
| Descrição | A aplicação permite que o `Usuário` visualize o código fonte de:<br>a) uma peça: selecionando a mesma;<br>b) de um conjunto de peças: selecionando uma peça que tenha outras peças encaixadas a ela (peças filho);<br>c) uma classe Java pronta para compilar o exercício em um projeto configurado com a biblioteca JOGL: selecionando a peça `Renderizador` do `painel de montagem`. Na classe será necessário ajustar o caminho das texturas, caso o exercício faça uso de texturas.<br>O código fonte não é gerado para seguintes peças:<br>`Drone`<br>`Destino`<br>`Animação` |
| Pré-condição | UC03 |
| Cenário principal | O Usuário seleciona uma peça. |
| Pós-condição | O painel `Comandos em JOGL` será atualizado com o código fonte correspondente da peça. |

Quadro 21 – Detalhamento do caso de uso UC05

| UC05 - Alterar e consultar as propriedades da peça selecionada | |
|---|---|
| Descrição | É possível consultar e alterar as propriedades das peças existentes no exercício, alterando assim o resultado no cenário 2D ou 3D e no código fonte. Com isto é possível visualizar o impacto de cada peça e de cada propriedade existente.<br>A peça `Renderizador` permite modificar a cor de limpeza, que pode ser visualizada pelo painel `Visão da Câmera`.<br>A peça `Câmera` permite a definir um nome para a peça, sua posição (`x`, `y`, `z`), o *look at* (x, y, z), o *near*, o *far* e o F*ield Of View* (FOV).<br>A peça `Objeto Gráfico` permite definir um nome para a peça e se ela será visível na cena. A peça também permite visualizar a matriz resultante das peças de transformações geométricas encaixadas nela.<br>A peça `Cubo` permite definir um nome para a peça, o seu tamanho (x, y, z), a sua posição (x, y, z), a sua cor ou a sua textura e se ela será visível na cena.<br>A peça `Polígono` permite definir um nome para a peça, a sua quantidade de pontos, o ponto que está selecionado (x, y, z), a sua primitiva gráfica, |

| | |
|---|---|
| | além da sua cor e se ela será visível na cena.<br>A peça `Spline` permite definir um nome para a peça, o seu tipo e a quantidade de pontos. Além disso é possível definir a posição (x, y, z) de cada um dos quatro pontos de controle, além da sua cor ou se ela será visível na cena. É possível também habilitar o desenho do seu `Poliedro`, definindo também uma cor especifica para ele.<br>A peça `drone` permite definir um nome para a peça, a sua posição (x, y, z) e se ela será visível na cena.<br>A peça `Destino` permite definir um nome para a peça, o seu tamanho (x, y, z), a sua posição (x, y, z), a sua cor ou a sua textura e se ela será visível na cena.<br>A peça `Iluminação` permite a definir um nome para a peça, a sua posição (x, y, z), a sua cor e se ela será visível na cena. Além disso é possível definir a intensidade da luz. Quanto o tipo de luz for *Directional* ou *SpotLight* é possível definir a posição (x, y, z) do alvo da luz.<br>A peça `Animação` permite definir um nome para a peça e qual a função que irá gerar os valores que serão somados a propriedade do objeto 3D durante a execução da animação.<br>As peças `Transladar`, `Rotacionar` e `Escalar` permitem definir um nome para as peças, o valor da transformação (x, y, z) e se a transformação será visível na cena. |
| Pré-condição | UC03 |
| Cenário Principal | O Usuário seleciona uma peça.<br>O Usuário altera a propriedade desejada. |
| Pós-condição | Os valores definidos para as propriedades deverão ser atualizados no código fonte e nas cenas do resultado do exercício. |

Quadro 22 – Detalhamento do caso de uso UC07

| UC07 - Visualizar visão da peça `Câmera` encaixada no exercício | |
|---|---|
| Descrição | Através do painel `Visão da Câmera` é possível visualizar a cena 2D ou 3D formada pelo exercício a partir de um segundo ponto de vista, que é determinado pelas propriedades da peça `Câmera` encaixada no `painel de montagem`. Os componentes de orientação dentro do painel `Espaço Gráfico` (os indicadores dos eixos x, y e z, a grade e a pirâmide da câmera) não serão enxergados neste painel. Assim, os únicos objetos que serão reproduzidos na cena serão o `Cubo`, `Polígono`, `Spline`, `Drone` ou `Destino` assim como a influência do objeto `Iluminação` nestes objetos.<br>A cor de limpeza da cena exibida no painel `Visão da Câmera` usa a cor de limpeza definida nas propriedades da peça `Renderizador` do `painel de montagem`. |

| | |
|---|---|
| Pré-Condição | UC01, UC02 ou UC05 |
| Cenário Principal | Encaixar na peça `Renderizador` a peça `Câmera` ou editar as propriedades da peça `Câmera` já encaixada. |
| Pós-Condição | O painel `Visão da Câmera` deverá exibir a visão da cena 2D ou 3D do exercício (também visualizada no painel `Espaço Gráfico`) a partir do ponto de vista definido pela peça `Câmera` encaixada na peça `Renderizador`. |

Quadro 23 – Detalhamento do caso de uso UC13

| UC13 – Executar animações | |
|---|---|
| Descrição | Através do painel `animação` é possível iniciar a execução das animações. Podendo configurar o tempo desejado para completar cada passo da animação. Neste mesmo painel é possível, durante a animação, visualizar os atributos de posição e rotação do `objeto gráfico` selecionado. |
| Pré-condição | UC01 |
| Cenário principal | O usuário seleciona um a peça do tipo `Objeto Gráfico` com um encaixe com outra peça do tipo `Animação`.<br>O usuário configura o tempo para cada passo da animação.<br>O usuário inicia a animação. |
| Cenário alternativo 1 | O usuário configura o tempo para cada passo da animação.<br>O usuário inicia a animação. |
| Pós-condição | O painel `Espaço Gráfico` e `Visão da Câmera` deverão exibir a cena mostrando a animação sendo executada. O painel `Animação` devera exibir os valores dos campos de posição (x,y,z) e rotação (x,y z) sendo atualizados durante a execução da animação. |

Quadro 24 – Detalhamento do caso de uso UC14

| UC14 – Executar comando em *drone* AR.Drone Parrot | |
|---|---|
| | Através do painel de `Animação` é possível configurar a conexão com um servidor ROS. Permitindo que as mesmas transformações configuradas como animações sejam executadas por um *drone* real. |
| Pré-condição | UC01, UC13 |
| Cenário principal | O usuário configura a URL de conexão com o servidor ROS.<br>O usuário calibra os sensores do *drone*.<br>O usuário seleciona um a peça do tipo `Objeto Gráfico` no qual possui um encaixe com outra peça do tipo `animação`.<br>O usuário inicia a execução com o *drone* real. |
| Cenário alternativo 1 | O usuário configura a URL de conexão com o servidor ROS.<br>O usuário seleciona um a peça do tipo `Objeto Gráfico` no qual possui um encaixe com outra |

| | |
|---|---|
| | peça do tipo `Animação`.<br>O usuário inicia a execução com o *drone* real. |
| Cenário alternativo 2 | O usuário configura a URL de conexão com o servidor ROS.<br>O usuário seleciona um a peça do tipo `Objeto Gráfico` no qual possui um encaixe com outra peça do tipo `animação`.<br>O usuário inicia a execução com o *drone* real<br>O usuário para a execução com o *drone* real utilizando o botão de pânico. |
| Pós-condição | Depois que o usuário iniciar a execução das animações com o *drone* físico a aplicação manda os comandos necessários, através do ROS, para a realiza-las. Se o usuário pressionar o botão de pânico, a aplicação manda o comando de pouso para o *drone*. |

## ANEXO A – Detalhamento das partes que compõem o VisEdu-CG

Neste anexo encontra-se o detalhamento das partes que compõem o VisEdu-CG. Segundo Nunes (2014, p.81) o VisEdu-CG é composto pelos seguintes elementos:

a) `Fábrica de peças`: painel composto por dois sub painéis, `painel de montagem` e o `painel da fábrica`. É onde são disponibilizadas as peças para criação do exercício;
b) `Comandos em JOGL`: painel no qual é exibido o código fonte Java, utilizando a biblioteca JOGL, necessário para gerar o resultado gráfico de peça selecionada;
c) `Lista de peças`: painel no qual é exibida a lista de peças encaixadas no exercício que permite a seleção rápida das mesmas;
d) `Espaço gráfico`: painel no qual é exibida a cena 3D de exercício
e) `Visão da câmera`: painel que permite a visualização da cena 3D através da visão da câmera adicionada no exercício. Sendo as propriedades utilizadas são as especificadas da peça "câmera" na `fábrica de peças`;
f) Painel `arquivo`: possui as opções de exportar e importar os exercícios;
g) `Propriedades da peça`: painel onde são exibidas as propriedades da peça selecionada;
h) Painel `ajuda`: local onde pode ser consultada informações sobre o funcionamento da aplicação.