INSTITUTO FEDERAL CATARINENSE
CAMPUS BLUMENAU
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E
DESENVOLVIMENTO DE SISTEMAS

MATEUS MORAES BUENO

DESENVOLVIMENTO DE UM SISTEMA DE DESENHO E ANIMAÇÃO VETORIAL 2D

TRABALHO DE CONCLUSÃO DE CURSO

BLUMENAU
2019

MATEUS MORAES BUENO

DESENVOLVIMENTO DE UM SISTEMA DE DESENHO E ANIMAÇÃO VETORIAL 2D

Trabalho de Conclusão de Curso apresentado ao Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Instituto Federal Catarinense, como requisito parcial para a obtenção do título de Tecnólogo.

Orientador: Paulo César Rodacki Gomes

BLUMENAU
2019

RESUMO

BUENO, Mateus. Desenvolvimento de um sistema de desenho e animação vetorial 2D. 2019. 37 f. Trabalho de Conclusão de Curso – Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, Instituto Federal Catarinense. Blumenau, 2019.

A animação é uma forma de expressão e um segmento de indústria que vem de um crescimento constante nos últimos anos. A computação gráfica é massivamente utilizada na criação de mundos, cenários, personagens e situações que eram impensáveis antes da tecnologia. Embora exista uma demanda ávida por imagens fotorrealistas, as quais criam renderizações gráficas mais próximas do mundo real, existe uma demanda crescente por renderizações não-fotorrealistas, por questões como expressão artística ou por simplicidade.

Este trabalho descreve o projeto e implementação de sistema de desenho vetorial e animação 2D. O objetivo do projeto é criar uma ferramenta computacional para que o artista se expresse através de desenhos à mão livre e de modificações das criações, marcando as modificações em quadros-chave e aplicando um método de interpolação, para automatizar o processo de criação de quadros intermediários. Utilizando as funções e classes disponibilizadas pelo *framework Qt*, é possível realizar a criação de janelas e manipulações gráficas, integrando equações matemáticas e estruturas de dados para as operações que o usuário pode executar.

O resultado é um *software* que o usuário pode utilizar como um papel para desenhar livremente e usar como se fossem folhas de quadros. O sistema oferece as ferramentas e a automação de quadros intermediários, permitindo que o artista controle a arte com sua criatividade apenas.

Palavras-chave: Desenho à mão livre. Gráficos vetoriais. Sistema de animação.

ABSTRACT

BUENO, Mateus. . 2019. 37 f. Trabalho de Conclusão de Curso – Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, Instituto Federal Catarinense. Blumenau, 2019.

Animation is a form of expression and an industry segment which is in a crescendo for the past few years. Computer graphics is massively applied in order to create and build worlds, scenario, characters and situations which were unthinkable without technology. Besides the eager demand of images rendered in Physically based rendered images, which aims to create graphics renditions more accurate to the real world, there is a growing demand for non-photorealistic rendering, for reasons like artistic expression or desired simplicity.
This paper describes the design and programming of a 2D vector drawing and animation system. The goal of this project is to provide a computer tool that the artist can use to freely express by freehand drawing and modifying the creations, setting the modifications in keyframes and applying an interpolate method, in order to automate the inbetweening process. By using the functions and classes provided by the *Qt Framework*, it's possible to do window creation and graphics manipulation, integrating mathematical equations and data structures for the operations that the user can execute.
The result is a software that the user can operate as a sheet for drawing freely and to use as frame sheets. The system provides the tools and the automation of inbetween frames, allowing the artist to control art with only creativity.

Keywords: Freehand draw. Vector graphics. Animation system.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

2D	Duas dimensões
3D	Três dimensões
CAD	*Computer-Aided Design*
JPG	*Joint Photographic Experts Group*
NPR	*Non-Photorealistic Rendering*
PBR	*Physically Based Rendering*
PNG	*Portable Network Graphics*
OMG	*Object Management Group*
SVG	*Scalable Vector Graphics*
UML	*Unified Modeling Language*
XML	*eXtensible Markup Language*

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

A animação é uma forma de expressão e um segmento de indústria que vem de um crescimento constante nos últimos anos. A computação gráfica é massivamente utilizada na criação de mundos, cenários, personagens e situações que eram impensáveis antes da tecnologia. Embora exista uma demanda ávida por imagens fotorrealistas, as quais criam renderizações gráficas mais próximas do mundo real, existe uma demanda crescente por renderizações Não-fotorrealistas, por questões como expressão artística ou por simplicidade. Este trabalho apresenta o desenvolvimento e implementação de um sistema de desenho e animação vetorial bidimensional. Neste capítulo introdutório, são apresentadas as motivações que levaram ao projeto e desenvolvimento desse trabalho, bem como os desafios e problemas presentes na área de animação por computação gráfica. Após, são explanados os objetivos gerais e específicos que nortearam o desenvolvimento apresentado. Na sequência, a metodologia escolhida para o desenvolvimento do sistema é introduzida e, depois, finalizando esse capítulo com a apresentação da estrutura documental dessa pesquisa.

1.1 Motivação

A computação gráfica e a animação assistida por computador conduziram uma verdadeira revolução tecnológica, tanto na própria produção de animação, seja para qualquer fim, quanto em efeitos visuais e imagens renderizadas para produtos variados. Avanços nesta área continuam acontecendo, enquanto novas técnicas digitais vão de encontro a públicos acolhedores (PARENT, 2012).

Todavia, ainda há muito potencial a ser explorado, na medida em que mais agentes atuantes na indústria do entretenimento demandam por meios cada vez mais eficientes de expressar seu próprio olhar especial, ao mesmo tempo que cada companhia tenta estabelecer um novo limite competitivo na produção de imagens e audiovisual (PARENT, 2012).

Ainda que os grandes nomes da indústria de animação concentrem seus esforços, cada vez mais, na produção de imagens fotorrealistas, Healey et al. (2004) afirmam que a produção de imagens e vídeos não-fotorrealistas tem uma miríade de potenciais utilizações. Uma das implicações interessantes dessa abordagem é o renascimento de uma mistura entre arte e ciência.

A renderização de imagens e vídeos não-fotorrealistas, ou NPR, começou a receber atenção da comunidade acadêmica nos anos 90. Winkenbach e Salesin (1994) comprovam as várias vantagens de ilustrações estilizadas frente à criação de imagens de cenas físicas com alta complexidade e realismo. Descrever uma cena de forma não-fotorrealista, como uma ilustração, pode transmitir mensagens de forma mais eficiente, pois é possível focar a atenção em características relevantes, omitindo detalhes que não agregam o resultado informativo e/ou

artístico (WINKENBACH; SALESIN, 1994). "Em muitas aplicações, ilustrações adicionam um senso de vitalidade, difícil de capturar com fotorrealismo", Winkenbach e Salesin (1994).

Segundo Leung e Lara (2015), "Desenho à mão livre ainda é uma das formas mais flexíveis e eficientes de expressar ideias criativas". Essa corrente de pensamento, aliada aos interesses midiáticos, artísticos e científicos, mantém a curiosidade e, principalmente, a motivação de se pesquisar e estudar técnicas no campo de imagens e vídeos renderizados não-fotorrealisticamente.

Esse interesse pela área de produção de imagens não-fotorrealistas é, ainda, motivado pela complexidade em se alcançar resultados satisfatoriamente artísticos.

> Existe uma grande quantidade de trabalho para alcançar uma sensação artística, talvez porque muito da beleza e autenticidade das obras de arte resultam da imprecisão e os computadores são incrivelmente precisos.(HEALEY et al., 2004)

O problema reside, primeiro, na forma da renderização do desenho à mão livre e, depois, na técnica para automatizar a criação da animação. Na questão da renderização do desenho, existem duas alternativas de criação dos gráficos em 2D: utilizar representação matricial (ou *raster*) ou gráficos vetoriais? Para entender o dilema é necessário contextualizar cada conceito.

Segundo (FIUME, 1989), gráficos *raster* são:

> Criados a partir de um arranjo de pontos de intensidade empacotados densamente (tipicamente retangulares) chamados de *elementos de imagem* ou *pixels*. Cada *pixel* presente na tela é uma primitiva da tela. Usualmente, um *pixel* pode ser endereçado individualmente, e pode ser aplicado a qualquer intensidade de cor vinda de um espaço de cor prescrito.

Já na criação de gráficos vetoriais, Azevedo e Conci (2003) afirmam que:

> São usados como elementos básicos os pontos, linhas, as curvas, as superfícies tridimensionais ou mesmo os sólidos que descrevem os elementos, que formam as imagens sinteticamente no computador.

As representações vetoriais são comuns em sistemas do tipo *CAD* (*Computer-Aided Design* ou Desenho Assistido por Computador).

Como os gráficos vetoriais se limitam a serem representados por elementos geométricos bem definidos, a criação de formas livres pode ser um pouco mais complicada. Por outro lado, a aplicação de transformações, como: translação, escala e rotação, é feita de forma natural a vetores utilizando transformações geométricas aplicados diretamente as retas e curvas criadas. No caso dos gráficos *rasterizados*, essas transformações geométricas teriam que ser aplicadas diretamente em cada *pixel* e, por mais que seja perfeitamente possível, existe uma perda de informação quando é aplicada uma transformação a uma imagem *rasterizada*.

Na questão da animação, existe uma série de técnicas que podem ser aplicadas para automatizar essa atividade, uma delas sendo **Animação por Quadro-Chave** que pode ser definida como:

> Um processo para criação de animações pelo qual os objetos são posicionados nos quadros críticos. Um quadro-chave (*keyframe*) é qualquer quadro de uma animação onde supostamente ocorre um evento específico importante. Os quadros localizados entre os quadros-chave são chamados de intermediários.

> Esse processo derivado da animação tradicional foi implementado em todos os sistemas de animação por computador. (AZEVEDO; CONCI, 2003).

A animação por quadro-chave é uma técnica que ainda permite bastante liberdade ao artista, porém, a criação de quadros intermediários pode ser desgastante.

1.2 Objetivos

Considerando as motivações ponderadas na seção anterior, o objetivo principal deste trabalho foi o desenvolvimento de um sistema de desenho e animação vetorial bidimensional.

Nas duas seções a seguir, os objetivos são explicitados.

1.2.1 Objetivos gerais

O objetivo geral do projeto é criar uma ferramenta computacional que permita ao artista criar formas livremente, utilizando um mouse ou uma mesa de desenho digital, e animar essas formas, manipulando a posição e o formato dos desenhos em quadros-chave, onde os quadros intermediários são criados automaticamente para preencher toda a sequência de imagens, dando a ilusão do movimento. A ideia é equilibrar a liberdade do artista, assim como ele teria ao desenhar no papel, com as facilidades que a computação gráfica provê, sem interferir no processo criativo do animador. Outros objetivos deste trabalho são: a aplicação de conceitos já presentes na computação gráfica no processo de animação tradicional, criar um ambiente de livre expressão para os artistas, sem inserir complexidades relacionadas à ferramentas ao processo de criação de desenhos, e utilizar recursos computacionais para a assistência de tarefas mais maçantes, como é a criação de quadros intermediários.

1.2.2 Objetivos Específicos

Os objetivos específicos deste trabalho, compreendem os seguintes itens:

- Compreender o escopo do trabalho através da visão do usuário potencial, entendendo o problema da animação de figuras bidimensionais;
- Realizar o levantamento dos requisitos para o desenvolvimento do sistema de desenho e de animação vetorial;
- Modelar o sistema de acordo com os requisitos e utilizando diagramas definidos na *UML*;
- Fazer a implementação do sistema, aplicando a modelagem desenvolvida e utilizando *frameworks* e recursos da linguagem de programação disponíveis para a codificação do sistema;
- Documentar todos os passos para a realização deste trabalho e os resultados obtidos.

1.3 Estrutura do trabalho

Este trabalho está estruturado da seguinte forma: o capítulo atual introduz alguns conceitos que serão aplicados no decorrer do trabalho, elucidando as motivações que levaram a criação do projeto, bem como os objetivos que se deseja alcançar com a efetivação do sistema. O capítulo Capítulo 2 explica, de forma geral, todo o desenvolvimento do projeto, mostrando as técnicas de projeto e modelagem aplicadas ao problema para a construção do sistema, incluindo: o levantamento de requisitos analisados na seção Seção 2.1 e a especificação do sistema desenvolvida através de técnicas de modelagem elucidadas na seção Seção 2.2. O capítulo Capítulo 3 explica a implentação do sistema, onde foi efetivado todo o projeto do sistema e aplicados os algoritmos em código. Na conclusão deste documento, o capítudo Capítulo 4 faz uma reflexão de todo o escopo do trabalho, analisando o problema que foi solucionado bem como fazendo considerações sobre o estado do sistema e as potencialidades que podem ser vislumbradas com o programa desenvolvido.

2 METODOLOGIA

Este trabalho descreve o desenvolvimento de um sistema para desenho e animação vetorial bidimensional. Para a construção do escopo do sistema, foi necessária a utilização de recursos, técnicas e ferramentas de análise, projeto e desenvolvimento de sistemas, os quais são elucidados neste capítulo.

Na primeira seção, são descritos os requisitos funcionais e não funcionais do sistema.

A especificação do projeto é descrita na sequência, ilustrada através de técnicas de engenharia de sistemas, como diagramas e modelagens de requisitos.

2.1 LEVANTAMENTO DE REQUISITOS DO SISTEMA

A engenharia de requisitos compreende o largo espectro que envolve tarefas e técnicas, as quais direcionam a um entendimento dos requisitos, construindo, assim, uma ponte entre projeto e construção de um sistema (PRESSMAN; MAXIM, 2016).

Para realizar o levantamento de requisitos, foram utilizadas **histórias de usuário**, uma técnica que "permite descrever saídas desejadas, características técnicas e funcionalidades necessárias no *software* a ser criado"(PRESSMAN; MAXIM, 2016). A partir das histórias de usuários foram definidos os requisitos *funcionais* e *não-funcionais*. As próximas três subseções descrevem esse processo: a primeira relata a construção das histórias de usuário, a segunda lista os requisitos funcionais e, por último, os requisitos não-funcionais são elicitados.

2.1.1 Histórias de Usuário

Segundo Vazquez e Simões (2016), **histórias de usuário** são "uma ou mais sentenças na linguagem de negócio ou cotidiana do usuário final ou usuário do sistema que captura o que um usuário faz ou necessita fazer como parte de sua função de trabalho".

A técnica de especificação de histórias de usuário é utilizada neste projeto com o intuito de ser a base para o escopo de todo o projeto. O primeiro produto a sair do emprego desta técnica é a consolidação das histórias de usuário em requisitos funcionais e não-funcionais. É necessário, então, que as histórias de usuário consigam abranger todo o escopo do sistema, mesmo que de maneira informal e generalista que essa técnica é prescrita.

Para a construção das histórias de usuário, foi utilizado o formato definido pela *Connextra* em 2001 (SZABO, 2017). Esse formato define que as histórias de usuários devem seguir um modelo de "papel-funcionalidade-motivo", descrevendo, na sequência:

1. O tipo de usuário ou o papel que ele exerce no uso do sistema;
2. A funcionalidade esperada pelo usuário, expressa no texto como um desejo;
3. O motivo ou objetivo, o qual o usuário deseja atingir com essa funcionalidade.

Esse modelo permite a construção de um texto na forma: "Como um <**papel**>, eu desejo/quero/preciso <**funcionalidade**> para que <**motivo/objetivo**>".

Cada funcionalidade do sistema é especificada nesse padrão, relacionando a funcionalidade com a finalidade e a origem dela, permitindo, assim, que os resquisitos do sistema sejam desenvolvidos ainda mais naturalmente.

A partir deste formato, foram elaboradas as seguintes histórias de usuário para desenvolvimento do resto do escopo do sistema:

1. Como um desenhista, eu preciso de uma área retangular na cor branca, para que eu possa desenvolver meus desenhos;
2. Como um desenhista, eu quero desenhar figuras geométricas bidimensionais livremente, para compor as formas dos meus desenhos;
3. Como um desenhista, eu preciso de ferramentas que permitam a edição das figuras desenhadas, para corrigir falhas e imperfeições;
4. Como um desenhista e animador, eu preciso de ferramentas que realizem transformações geométricas nas figuras desenhadas, para compor os desenhos e mudar as características deles;
5. Como um animador, eu preciso que cada transformação nos desenhos seja armazenada na sequência temporal em que é aplicada, para que cada transformação forme a animação;
6. Como um animador, eu quero uma ferramenta que me indique visualmente o momento temporal em que a edição está sendo feita, para que cada momento represente um quadro da animação;
7. Como um animador, eu quero que cada momento temporal seja relativo a uma fração específica de um segundo, para que cada quadro represente uma fração de segundo;
8. Como um animador, eu preciso que transformações aplicadas entre um quadro e outro seja corretamente preenchida pelo sistema, para que o trabalho de animação seja automatizado;
9. Como um usuário do sistema, eu preciso que os desenhos e transformações aplicadas temporalmente possam ser reproduzidas numa área separada da área de desenho, para que seja possível verificar como está a execução do trabalho de animação;
10. Como um usuário do sistema, eu preciso que a animação desenvolvida possa ser salva em um formato de vídeo ou sequência de imagens, para que ela seja finalizada;
11. Como um usuário do sistema, eu preciso que o trabalho realizado no sistema seja salvo em formato próprio, para que a animação possa ser trabalhada posteriormente ou em outro computador.

Após a construção das histórias de usuário, são, então, levantados os requisitos funcionais e não funcionais, explanados na seção seguinte.

2.1.2 Requisitos Funcionais

Elaboradas as histórias de usuário, o próximo passo é condensar cada história em um ou mais requisitos. A partir do levantamento feito até então, devidamente analisado, tem-se a

conclusão de que o sistema necessário para atender as demandas do problema deve compreender um software que permita ao usuário desenhar formas geométricas bidimensionais e animar essas formas, com o auxílio de um sistema que permita marcar os quadros-chave e preencher os quadros intermediários, com o objetivo de formar toda a animação, conforme o artista imaginou.

Com a compreensão do tipo de sistema que é necessário para atender a demanda, parte-se, então, para a elaboração dos requisitos, identificando-os a partir das histórias de usuários.

Para desmembrar o sistema nas funcionalidades que ele deve atender, os requisitos funcionais são levantados. Sommerville (2010) afirma que "Os requisitos funcionais de um sistema descrevem o que ele deve fazer. Eles dependem do tipo de software a ser desenvolvido, de quem são seus possíveis usuários e da abordagem adotada pela organização ao descrever os requisitos."Os requisitos serão, então, levantados a partir da análise das histórias de usuário e "[...] descritos de forma abstrata, para serem compreendidos pelos usuários do sistema."(SOMMERVILLE, 2010)

Começando a análise a partir da primeira e segunda história de usuário, conclui-se que um requisito funcional pode ser descrito para atender a demanda de desenvolver desenhos a partir de formas geométricas. A terceira e quarta história de usuário são resumidas em um requisito também. A partir da quinta história de usuário e prosseguindo-se até a história de usuário número 8 (oito), constatamos atividades relativas ao trabalho de um animador que podem ser desmembradas em dois requisitos funcionais. As histórias de usuário 9 (nove) e 10 (dez) tratam da finalização do trabalho, possibilitando serem relacionadas a um requisito funcional. Por último, temos um requisito funcional que pode atender a demanda da última história de usuário.

Os requisitos funcionais seguem, então, da seguinte forma:

1. O sistema deve permitir que o usuário desenhe de forma livre em um ambiente bidimensional;
2. O usuário deve ser capaz de editar os desenhos;
3. O usuário deve conseguir marcar mudanças realizadas nos desenhos;
4. O sistema deve gerar alterações entre as edições marcadas pelo usuário para completar a animação desejada;
5. Ao usuário, deve ser permitido visualizar o trabalho de animação realizado;
6. O sistema deve permitir que o usuário salve o trabalho realizado.

Como as histórias de usuário elicitam, além das funcionalidades desejadas, a forma desejada de como os objetivos devem ser atingidos, os requisitos funcionais podem ser mais sucintos, descrevendo de maneira mais direta, porém mantendo a abstração, as propriedades almejadas.

As formas como essas funcionalidades serão efetuadas, são expostas na próxima seção, com os requisitos não-funcionais.

2.1.3 Requisitos Não-Funcionais

As histórias de usuário, as quais originaram requisitos funcionais, carregam conceitos de usabilidade das funcionalidades, descrevendo como determinada propriedade do sistema deveria agir. Os requisitos funcionais não descrevem como o sistema atuará, se limitando ao escopo objetivo de elicitar o quê será alcançado com cada requisito. Esta outra função fica a cargo dos requisitos não-funcionais.

Pressman e Maxim (2016) descrevem os requisitos não-funcionais como "[..] um atributo de qualidade, um atributo de desempenho, um atributo de segurança ou uma restrição geral em um sistema."A partir dessa conceitualização, pode-se extrair os requisitos não-funcionais das histórias de usuário, identificando quando os usuário demandam por propriedades qualitativas, restritivas, de performance ou de segurança.

1. O sistema deverá ter ícones intuitivos nos botões;
2. O sistema deve suportar desenho através do mouse e tablet de desenho;
3. O sistema executará nos sistemas operacionais Windows, Linux e Mac OS;
4. O sistema não será executado em plataformas móveis;
5. A implementação do sistema deverá ser realizada em linguagem de programação e framework que possa ser compilado para vários Sistemas Operacionais;
6. O sistema será desenvolvido utilizando o paradigma de programação orientado a objetos;
7. O usuário deve poder visualizar o trabalho de animação em uma janela separada da área de desenho;
8. O sistema deve permitir que a animação seja salva em formato de vídeo e em sequência de imagens;
9. O trabalho realizado dentro do sistema pode ser salvo e aberto em outra ocasião ou, até mesmo, em outro computador com o sistema instalado.

Com os requisitos funcionais e não-funcionais concluídos, parte-se para a especificação do sistema, explanada a partir das seções seguintes.

2.2 ESPECIFICAÇÃO DO SISTEMA

Os requisitos funcionais e não-funcionais levantados, com base nas histórias de usuário, engendram a construção de um sistema de desenho e animação vetorial bidimensional. Várias características e funcionalidades do sistema são percebidas na elaboração dos requisitos. Porém, é na especificação do sistema que a proposta do projeto é pormenorizada, modelando cada funcionalidade numa linguagem acessível e ilustrativa e que, ao mesmo tempo, consiga adentrar em alguns detalhes de implementação.

O uso de notações gráficas que sejam apoiadas em um metamodelo único, é uma forma de auxílio à descrição e no projeto de *softwares* baseados em orientação a objetos (FOWLER, 2003). A UML (*Unified Modeling Language*) é um padrão aberto de modelagem orientada a objetos, controlada e especificada pela OMG (*Object Management Group*), como uma forma

de unificar as várias linguagens gráficas de modelagem que coexistiam no final da década de 80 e início dos anos 90 (FOWLER, 2003). A UML especifica uma grande gama de diagramas, apoiando várias perspectivas diferentes através de modelos estruturais, comportamentais, de interação e externos (SOMMERVILLE, 2010).

Para a especifação do sistema proposto neste projeto, utilizam-se os diagramas de casos de uso e de classes, os quais são entendidos como suficientes para a modelagem de um software dessa natureza e são explanados nas próximas subseções. Os diagramas foram todos desenvolvidos no programa *Sketchboard*, disponível online através do site www.sketchboard.io.

2.2.1 Diagramas de Caso de Uso

Casos de uso podem ser descritos como conjuntos de "[...] cenários amarrados por um objetivo comum de usuário."(FOWLER, 2003), possibilitando descrevê-los de forma textual e, também, em formato de diagrama, onde as duas formas são interessantes de serem utilizadas em conjunto. Fowler (2003) ainda conceitua um cenário como uma interação do usuário com o sistema, descrita com uma sequência de passos.

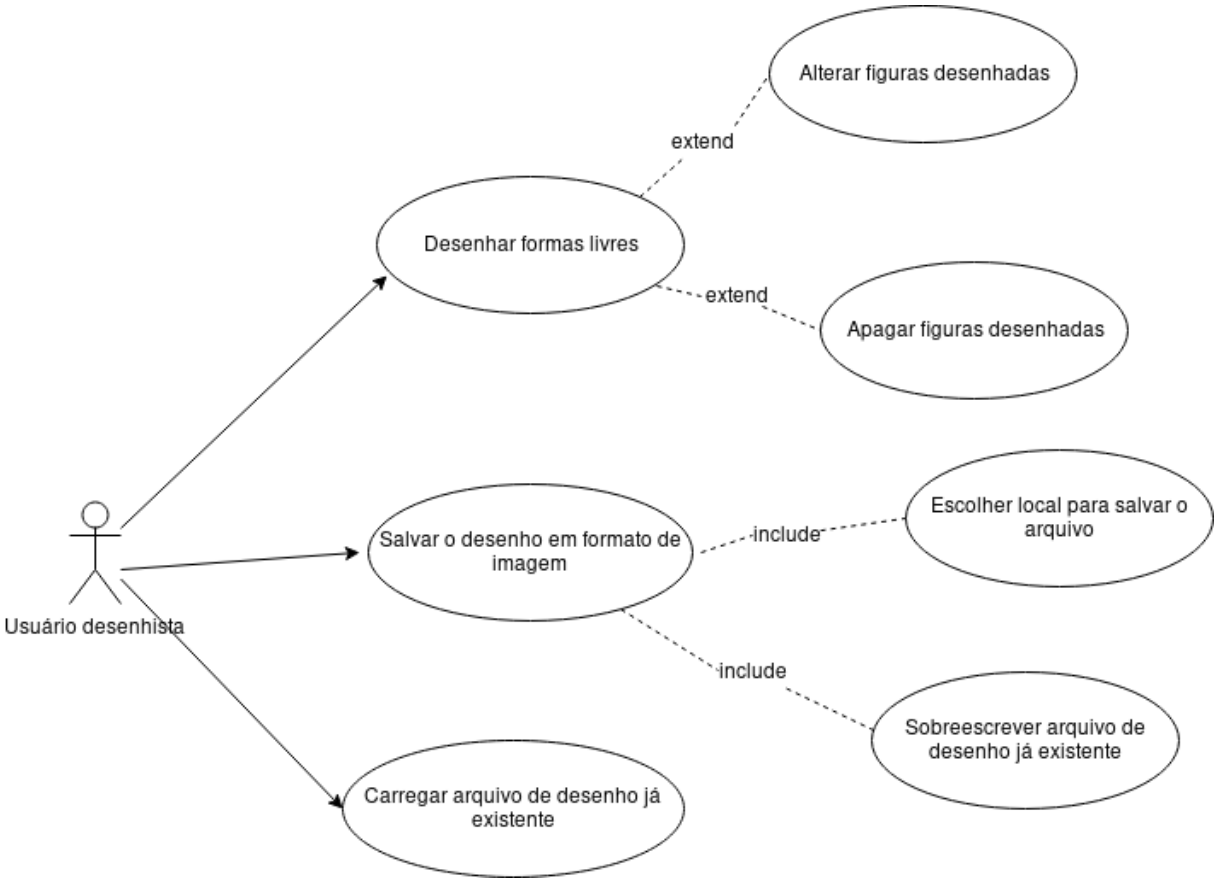
Essas interações preveem as figuras de atores principais e secundários, que são elementos que são satisfeitos pelos casos de uso. Uma vez que foram elaboradas histórias de usuários, antes do processo de levantamento de requisitos, a compreensão de quais são os atores do sistema é atingida de forma natural e consequente. Não necessariamente, os atores principais e secundários devem ser usuários ou papéis de usuários que interagem direta ou indiretamente com o sistema, como também podem ser sistemas, módulos e componentes externos e internos que também se comunicam e esperam resultados do sistema (SOMMERVILLE, 2010).

Foram desenvolvidos dois diagramas de caso de uso para identificar os atores envolvidos com o sistema, o funcionamento geral dos módulos e cada ação prevista. A modularização do sistema é resultado da análise da relação entre os requisitos funcionais e não-funcionais, onde foi concluído que existe, pelo menos, dois subsistemas interagindo para entregar o produto final esperado pelo usuário, ou seja, uma ou mais cenas animadas. Os dois subsistemas, a saber, consistem de: um módulo para desenhos vetorias e outro módulo para o processo de animação do desenho. Organizando desta forma, a modelagem do sistema é mais precisa e, também, simplifica o desenvolvimento, além de especializar cada módulo, definindo suas ações específicas e as limitando no seu escopo. Essa acepção nesta fase da modelagem não significa que, de fato, o programa consistirá de dois subsistemas no fim, embora sempre se busque, em cada passo, chegar mais próximo da ideia do que será, na prática, a implementação do sistema como um todo. Conforme a modelagem for avançando e o projeto seja analisado na forma de outros diagramas, novas decisão em modularização, especialização ou generalização podem ser percebidas e concluídas como recursos mais acurados para a implementação mais ótima dessa proposta.

2.2.1.1 Diagrama de Caso de Uso de Desenho Vetorial

O primeiro diagrama é apresentado na figura Figura 1, onde engloba-se o subsistema de desenho, ou seja, todas as tarefas relativas a efetivar os desenhos desejados pelo usuário.

Figura 1 – Diagrama de Caso de Uso de Desenho Vetorial



Fonte: Do autor

Descreve-se, então, de forma textual, as informações do diagrama da figura Figura 1, para melhor entendimento de cada ação:

- **Ator - Usuário desenhista**: Trata-se de uma forma de nominar o usuário, de acordo com a ação que ele desempenhará nesse subsistema. Essa identificação já limita o escopo das ações e prevê que tipo de funções do sistema o usuário pode acessar nesse nível. Em todo caso, o mesmo usuário pode desempenhar todos os papéis identificados nos diagramas de caso de uso, não sendo necessário e nem imposto que sejam divididas as tarefas do sistema entre vários usuários, apesar de ser perfeitamente cabível em procedimentos mais complexos. Outrossim, a nomenclatura é apenas uma configuração sintática para a finalidade do subsistema de desenho.
- **Caso de Uso - Desenhar formas livres**: Este caso de uso minudencia a atividade de desenhar as figuras, indicando que o desenhista criará formas de maneira livre, as quais comporão o desenho a ser animado.

- **Caso de Uso - Alterar figuras desenhadas**: Após a inserção de formas geométricas, é facultado ao desenhista alterar propriedades das formas inseridas, compreendendo as ações de transladar as figuras, isto é, mudar a posição da forma no plano em que está inserida; escalar o tamanho da figura, seja para diminuir ou aumentar a área ocupada pela forma; e rotacionar a figura sob seus eixos, em sentido horário e anti-horário. Este caso de uso depende da inserção prévia da figura geométrica a qual se pretende aplicar as transformações.
- **Caso de Uso - Apagar figuras desenhadas**: Este caso de uso também depende da inserção da figura geométrica, pois só se pode deletar figuras que tenham sido desenhadas. De todo modo, essa ação é opcional ao desenhista e serve para que ele tenha total controle do desenho a ser efetuado.
- **Caso de Uso - Salvar o desenho em formatos de imagem**: Para que não seja necessário recomeçar o trabalho toda vez que executar o programa e para manter os trabalhos realizados de forma segura, o sistema deve prover um mecanismo de salvamento das imagens nos formatos mais comuns. Os arquivos salvos devem estar em formatos que possam ser abertos em outros programas que visualizam e editam imagens, vetoriais e *raster*.
- **Caso de Uso - Escolher local para salvar o arquivo**: O arquivo de imagem, contendo as figuras geométricas desenhadas pelo usuário, devem ser armazenadas no sistema de arquivos do computador do usuário. Este caso de uso vem logo após a ação de escolher salvar o desenho.
- **Caso de Uso - Sobrescrever arquivo de desenho já existente**: No ato de salvar o desenho, o desenhista pode escolher atualizar um arquivo que ele já tenha salvo previamente, permitindo que o trabalho seja realizado de forma incremental.
- **Caso de Uso - Carregar arquivo de desenho já existente**: Para continuar trabalhos prévios, o usuário pode selecionar do sistema de arquivos, o arquivo de imagem o qual estava trabalhando, dando prosseguimento ao seu desenho.

O subsistema de desenho vetorial fica assim explanado, a partir da visão das ações oferecidas ao usuário. De todo modo, pode-se visualizar esse subsistema como suficiente em si mesmo, pois não requer que, de fato, o subsistema de animação influencie neste, pois o desenho vetorial pode subsistir em si só, sendo utilizado como um programa simples de desenho vetorial. Entretanto, a proposta deste projeto é o desenvolvimento de um sistema de animação bidimensional de desenhos vetoriais. Sendo assim, o poder total do programa é alcançado com o uso do subsistema de animação, juntamente ao de desenho. Subsistema este que é modelado na forma de diagrama de caso de uso, tal-qualmente, na próxima subseção.

2.2.1.2 Diagrama de Caso de Uso de Animação

O diagrama de caso de uso que compreende as ações de animar os desenhos vetoriais é apresentado na figura Figura 2.

Figura 2 – Diagrama de Caso de Uso de Animação 2D

Fonte: Do autor

A partir da visualização do diagrama da figura Figura 2, partimos para a explicação em formato textual, obtendo a compreensão plena da modelagem:

- **Ator - Usuário animador**: Como explanado no diagrama de caso de uso de desenho vetorial, não necessariamente os usuários desenhista e animador serão dois usuários distintos, ainda que esse procedimento seja cabível em determinados cenários e abordagens. Conquanto, ainda é inexorável a necessidade de descrever bem esses papéis. No caso do usuário animador, as suas ações estarão delimitadas ao escopo de realizar modificações nas figuras desenhadas e marcar essas modificações numa linha do tempo. Com o auxílio do sistema, este ator pode expressar sua criatividade e criar a ilusão de movimento e vida para as formas já inseridas.
- **Ator - Sistema de Animação**: No subsistema de animação, existem atividades automatizadas pelo próprio sistema, com a finalidade de auxiliar as ações mais morosas para

o animador, as quais consistiriam das ações de marcar alterações em cada quadro, necessário para a realização de toda a ilusão de movimento. Para elidir atividades dispendiosas, as quais não são necessárias para o processo criativo, é imprescindível um ator de sistema que seja responsável por esses casos de uso específicos. A identificação do sistema de interpolação de quadros como um ator que, também, realiza ações no sistema, clarifica os casos de uso relativos a interpolação de quadros-chave e estabelece relações com as escolhas e ações que estão restritas no escopo do usuário animador. Esta atividade de interpolação de quadros, consiste exatamente na geração automatizada das alterações nas figuras que intermediam os quadros destacados pelo animador.

- **Caso de Uso - Alterar propriedades dos objetos desenhados**: Este caso de uso ilustra a atividade principal do animador. As alterações nas figuras, juntamente com o caso de uso que consiste em marcar essas alterações numa linha do tempo, são responsáveis pela ilusão do movimento. As propriedades que podem ser alteradas nos objetos são alusivas às ações de manipulação da posição e da formas dos desenhos. A distinção deste caso de uso, para o caso de uso similiar no diagrama do desenhista, repousa na intenção de cada papel: o desenhista altera as formas para alcançar a figura pretendida; o animador faz o mesmo, mas com o intuito de simular as várias fases do movimento almejado.
- **Caso de Uso - Selecionar quadros em uma linha do tempo**: O sistema deverá exibir uma linha do tempo, preferencialmente, logo abaixo da área de desenho. Essa linha do tempo deve prover um mecanismo de navegação entre as demarcações da linha do tempo. Cada uma dessas demarcações representará um quadro-chave situado na unidade de tempo representada pela linha do tempo, isto é, se a linha do tempo representar 10 segundos e houver 240 demarcações, cada demarcação corresponderá a $1/24$ quadro por segundo. Deslizando o mecanismo de navegação, o animador pode selecionar diferentes quadros no tempo e realizar as alterações em qualquer momento temporal que desejar.
- **Caso de Uso - Marcar quadro-chave**: Isocronicamente à atividade de realizar alterações nas figuras, é imprescindível que essas alterações sejam indicadas ao sistema, de forma a destacar que elas compõem um quadro-chave a ser interpolado. No comando de marcação do quadro-chave, a demarcação indicará visualmente que se trata de um quadro-chave. O usuário animador pode marcar quantos quadros-chave desejar, não sendo nem obrigatório um intervalamento constante entre os quadros-chave, possibilitando, inclusive, que vários quadros-chave em sequência sejam marcados. É claro que, a consistência na intervalação, impactará de certas formas na precisão da animação. Em todo o caso, o sistema automatizado de interpolação criará os quadros-intermediários mais precisos para realizar toda a animação.
- **Caso de Uso - Escolher método de interpolação**: Ao usuário animador, é disposto mais um mecanismo para assegurar um pouco mais de controle ao resultado pretendido na animação. A escolha do método de interpolação visa garantir mais opções de precisão

e, também, de estética ao movimento desenvolvido. Dependendo do tipo de equação de interpolação, propriedades como a trajetória e a suavidade, ou rijeza, dos movimentos, são impactados e podem apresentar resultados diferentes. O usuário animador deve entender os diferentes tipos de interpolação e aplicar o método mais condizente com o resultado que espera.

- **Caso de Uso - Interpolar quadros-chave**: Este é um dos casos de uso que é relacionado ao ator de sistema de interpolação de quadros. Trata-se de uma atividade automatizada e que será aplicada ao trabalho desenvolvido pelo usuário. A interpolação dos quadros-chave seguirá o método escolhido pelo animador e dependendo da equação selecionada, um tipo de interpolação será aplicado, gerando os quadros intermediários de acordo com a curva da equação, ou seja, a quantidade de quadros com alterações entre os quadros-chave será determinada pelo tipo da equação. Essas alterações irão compor o que for necessário para gerar o movimento entre o quadro-chave inicial com o quadro-chave final. Essa automatização será melhor explicada na implementação e algoritmos do sistema.
- **Caso de Uso - Pré-visualizar animação**: Em alguns momentos do trabalho do animador, é preciso checar se os resultados estão dentro do esperado e, principalmente, se a animação está condizente com o trabalho realizado. O sistema deverá prover um recurso para visualizar a animação realizada até então. Este recurso é dependente do sistema de interpolação, pois a geração do vídeo é composta dos quadros-chave e intermediários sendo reproduzidos sequencialmente, obedecendo a linha temporal estabelecida. Visualizando o trabalho desenvolvido, o animador pode fazer os ajustes necessários e entender como poderá ficar o produto final.
- **Caso de Uso - Gerar animação**: A renderização da animação consistirá da agregação de todos os trabahos realizados, consistindo dos desenhos inseridos e alterados, as criações e automatizações dos quadros, constituindo um vídeo ou sequência de imagens representando todo a animação desenvolvida. A geração da animação é parecida com a pré-visualização, com a exceção de que o produto originado deste caso de uso já será direcionado para o armazenamento no formato escolhido pelo usuário.
- **Caso de Uso - Exportar em formato de vídeo**: Com a geração da animação, o usuário pode escolher o tipo de arquivo para conter a animação. A exportação do trabalho seguirá o formato escolhido e, no caso do vídeo, os formatos oferecidos devem seguir o padrão usual, o qual possibilita que seja reproduzido nos *softwares* mais comuns no mercado.
- **Caso de Uso - Exportar em sequência de arquivos de imagem**: O mesmo aplicado ao caso de uso da exportação da animação em formato de vídeo, aplica-se também a este caso de uso. A diferença é que várias imagens serão geradas, cada uma representando um quadro da animação. O formato das imagens também deve seguir os padrões mais utilizados. Tanto no caso da exportação em vídeo, como neste caso da exportação em formato de sequência de imagens, o(s) arquivo(s) gerado(s) pode(m) ser reproduzido(s) ou carregado(s) em *softwares* de composição, a fim de realizar mais correções ou editar

a animação em um trabalho mais complexo e composto de mais fases, fora do escopo da proposta deste projeto.

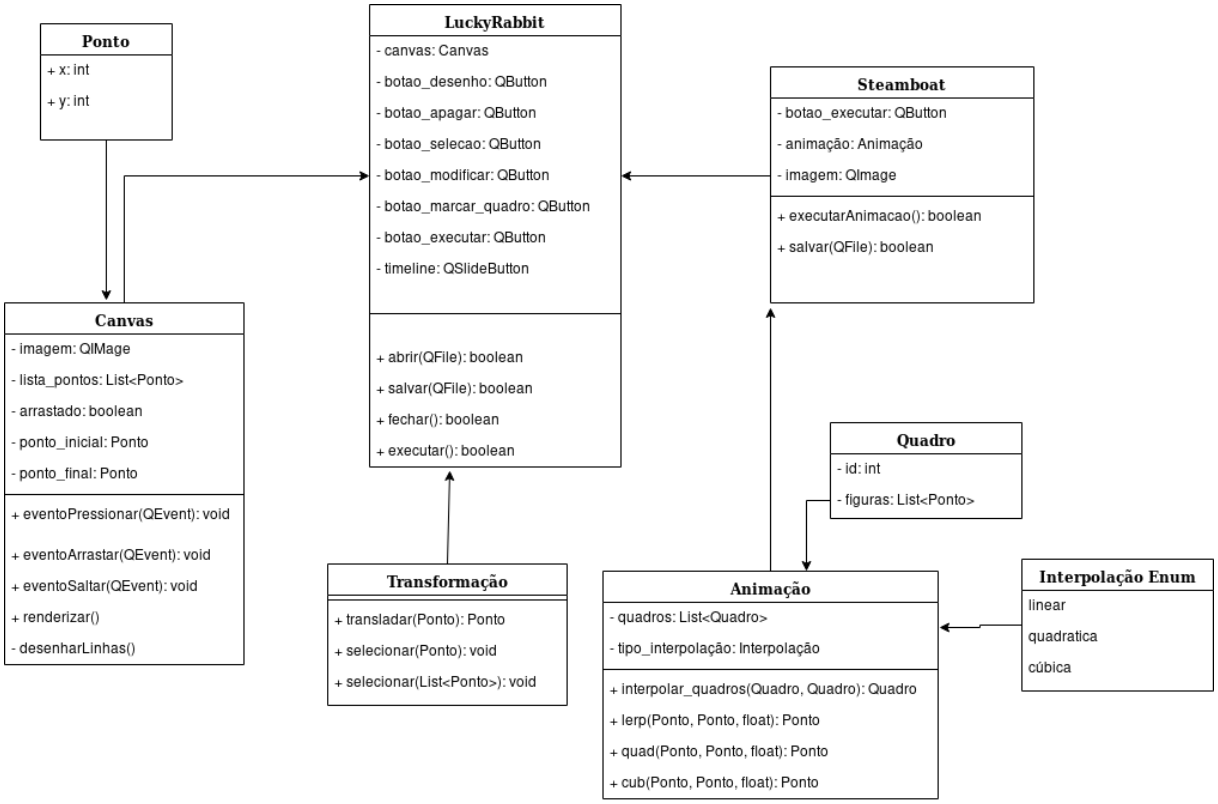
- **Caso de Uso - Salvar trabalho de animação**: Similar ao caso de uso de salvar os desenhos, presente no diagrama de caso de uso do desenhista, o armazenamento do trabalho de animação também se faz presente aqui. Dessemelhante à exportação da animação, o salvamento do trabalho de animação será em formato específico do programa, sendo carregado apenas neste ou na eventualidade de algum programa futuro ser compatível com o da proposta deste projeto.
- **Caso de Uso - Escolher local de armazenamento**: A escolha do local de salvamento é restrita ao próprio sistema de arquivos do sistema operacional do usuário. As funções mais comuns de armazenagem de arquivo é disposta ao usuário, consistindo da nomeação do arquivo (apenas o nome, a extensão é fixa) e o local onde será armazenado o arquivo.
- **Caso de Uso - Sobrescrever arquivo existente**: Da mesma forma que o local é escolhido para armazenar o arquivo próprio do sistema, o usuário pode armazenar as alterações realizadas em um trabalho já carregado e que foi inicializado em outro momento. A sobrescrita segue as mesmas regras de armazanamento do sistema de arquivos. A exceção para o caso de uso anterior é a de que o mesmo nome de arquivo será utilizado e o trabalho será sobrescrito.
- **Caso de Uso - Abrir um trabalho**: Carregar um trabalho anteriormente criado consiste de abrir arquivos com o formato específico do programa. Este caso de uso serve ao propósito de dar continuidade ao desenvolvimento do animador e desenhista.

Explanados, então, os dois subsistemas organizados na análise em formato de casos de uso, prosseguimos na modelagem do trabalho, a partir de outros diagramas que delinearão outros aspectos do projeto, com o intuito de elucidar ainda mais a implementação.

2.2.2 Diagrama de Classes

A análise dos requisitos do sistema delineia uma série de ações relacionadas a agentes atuantes no sistema, ações estas identificadas como casos de uso e agentes nominados atores. De certa forma, essa modelagem, até então, ainda se situa, praticamente, ao mesmo nível de abstração das histórias de usuário. A medida que a modelagem se aproxima de quesitos mais próximos à implementação do sistema em si, as ferramentas de modelagem devem refletir elementos mais condizentes com conceitos envolvidos na própria programação da solução. Neste capítulo, será dilucidado o diagrama de classes construído com base na análise de todo o material desenvolvido até então. A utilização dessa ferramenta de modelagem é importante nesta etapa da especificação pois, como Fowler (2003) define, "Um diagrama de classe descreve os tipos de objetos presentes no sistema e os vários tipos de relacionamento estáticos existentes entre eles."Os objetos, no caso, serão instâncias das classes definidas e que se relacionam a partir dos vários tipos de associação entre as classes modeladas. A figura Figura 3 demonstra o diagrama de classes construída para esse projeto.

Figura 3 – Diagrama de Classes



Fonte: Do autor

Para melhor entendimento do diagrama e as relações entre as classes, segue uma explicação breve sobre cada classe:

- **Classe - Lucky Rabbit**: É o sistema principal de desenho. Cria a interface gŕafica com o usuário, dispondo os botões e eventos associados e instancia a classe Canvas, que é a representação da tela de desenho. Dispõe de um menu com operações de armazenamento de arquivos e associa o sistema de animação com o botão de execução da animação.
- **Classe - Canvas**: A classe Canvas é a representação da tela de desenho. Toda forma criada livremente pelo usuário é inserida em um objeto do tipo Imagem, sendo que essa instância é desenhada na tela em toda chamada ao evento de renderização. Mantém propriedades necessárias para o desenho e eventos relacionados à pintura pelos dispositivos utilizados pelo usuário.
- **Classe - Ponto**: O Ponto é uma primitiva gráfica de desenho, sendo a classe mais fundamental para todas as operações relacionadas ao processo de desenho e animação. Essa classe é instanciada várias vezes pela classe Canvas, conforme o usuário move o dispositivo pressionado, alimentando coordenadas de pontos por onde o dispositivo percorrer, sendo que esses pontos são as diretivas para o desenho da forma desejada.
- **Classe Abstrata - Manipulação**: Essa classe abstrata provê um conjunto de operações associadas à edição das figuras desenhadas. As funções de translação e de seleção são implementadas no sistema de desenho, mais especificamente, aos botões de manipulação,

permitindo que o usuário escolha a opção que lhe convém de seleção para manipular os pontos e formas a partir da função de translação.

- **Classe - Steamboat**: A classe Steamboat é a classe principal para o sistema de animação. Reúne todas as operações e classes relacionadas ao processo de animação dos desenhos. Sua principal função é criar uma tela de execução da animação, mas também oferece os botões e eventos relacionados ao armazenamento no sistema de arquivos e implementa a instância da classe Animação, contendo as funções para gerar os quadros intermediários.
- **Classe - Animação**: A classe da animação é o núcleo de todo o sistema de animação. Nessa classe, são implementadas as funções de interpolação para criação dos quadros intermediários. Essa classe é instanciada pela classe principal do sistema de animação e chamada já na primeira execução, interpolando os desenhos com a função escolhida.
- **Classe - Quadro**: A classe Quadro é a definição dos quadros de animação, sendo instanciado toda vez que se define um quadro novo. Essa instanciação acontece sempre que um quadro-chave é definido e, também, quando os quadros intermediários são gerados. O quadro tem dois campos, sendo um identificador que é um número sequencial relativo à sequência temporal do quadro. O outro campo é uma lista de figuras, que nada mais são que conjuntos de pontos, os quais definem as posições das figuras em cada quadro específico. Todas essas informações são utilizadas para a criação de cada imagem, relativa a cada quadro.
- **Enumeração - Interpolação**: É um tipo de dado para definição dos valores possíveis para a interpolação. Serve especificamente para a definição de qual método de interpolaçao será utilizado para a animação, ficando a cargo do usuário essa escolha de acordo com sua percepção artística. Os valores possíveis são Linear, Quadrática e Cúbica. Esses valores definidos servem, depois, para a chamada da função de interpolação relativa à instância do Enum definida. O Enum é utilizado na classe Animação e definido por uma lista disposta na interface gráfica com o usuário.

O diagrama de classes, assim construído, permite que a estrutura das classes a serem utilizadas no sistema sejam bem visualizadas. No próximo capítulo, a implementação do sistema é explicada, pontuando todas as funções e algoritmos utilizados para a criação das operações e classes definidas na modelagem.

3 DESENVOLVIMENTO

A codificação do sistema segue a modelagem realizada nas seções anteriores. As classes e comportamentos projetados são implementados no sistema, de acordo com os recursos disponíveis da plataforma onde o sistema operará e utilizando as ferramentas providas pela linguagem de programação e frameworks utilizados.

Nas seguintes subseções serão apresentados os detalhes de implementação do sistema, concernentes à arquitetura utilizada, na seção Seção 3.1. Na seção Seção 3.2 é esmiuçada a implementação do subsistema de desenho vetorial e na seção Seção 3.3 são explicados os algoritmos aplicados para geração das animações.

3.1 Arquitetura do sistema

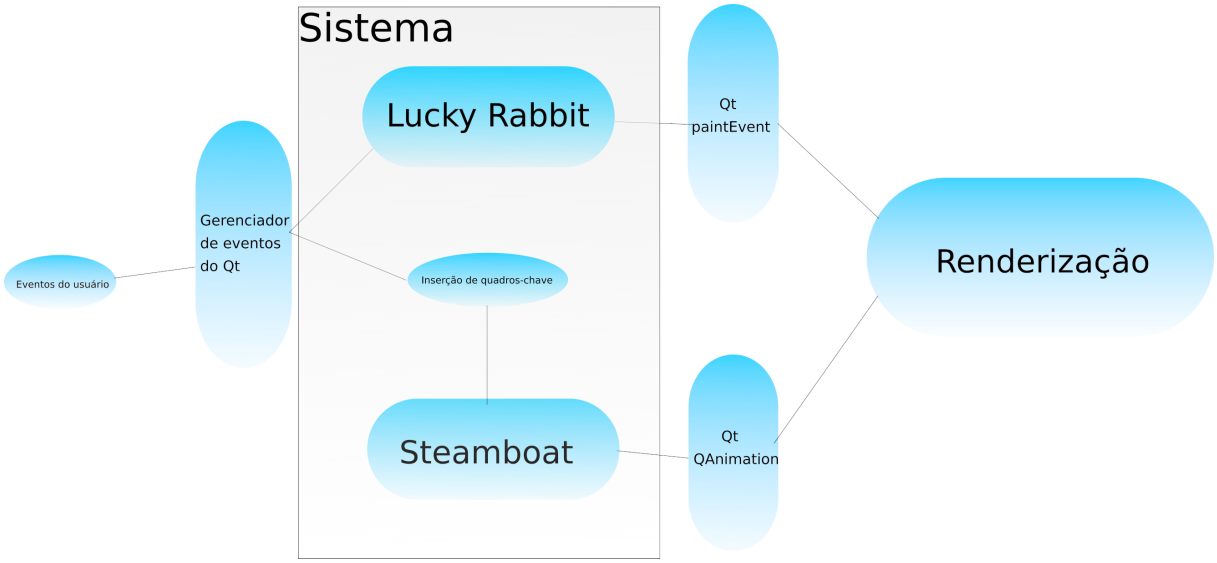
Esse sistema de animação de desenhos vetoriais foi criado, tendo em mente, a execução nas plataformas mais comuns de computação em mesa, compreendendo os Sistemas Operacionais *Microsoft Windows*, *Apple Mac OS* e as distribuições baseadas no *kernel Linux* que contenham a suíte gráfica *Qt*. Para garantir uma compatibilidade de código entre esses três sistemas operacionais, isto é, para que o mesmo código possa ser compilado da mesma forma nos três sistemas; e, ao mesmo tempo, utilizar os recursos mais otimizados de cada plataforma, foi escolhido o *framework* de desenvolvimento gráfico *Qt*, da *Qt Company* (COMPANY, 2018). Esta biblioteca de código traz estruturas e funções prontas para serem aplicadas na construção de sistemas baseados em janelas, com manipulação de componentes através de eventos do mouse e teclado e desenhos de várias formas utilizando as *APIs* gráficas nativas de cada plataforma.

A versão do *Qt* utilizada para a construção desse sistema é a 5.2 e a linguagem de programação, padrão para a biblioteca, e na qual o código foi escrito é a *C++*. Como os três sistemas operacionais alvo tem compiladores nativos de *C++*, os quais traduzem o código escrito nessa linguagem, diretamente em linguagem de máquina, o mesmo código pode ser compilado nas diferentes plataformas, gerando executáveis para cada sistema operacional. O editor utilizado para a escrita do código foi o *Qt Creator*, uma *IDE* disponibilizada pela *Qt Company*, que é compatível com *Microsoft Windows*, *Apple Mac OS* e *Linux*, além de compilar aplicativos para sistemas embarcados, Android e iOS. O conjunto utilizado para este sistema compreende as bibliotecas necessárias para criação de aplicações baseadas em desktop, já que o alvo são usuários de computadores de mesa e notebooks, que tenham à disposição um dispositivo de entrada baseado em ponteiro, como mouses e tablets de desenho.

A componentização do sistema e a comunicação com o *framework* ficaram, assim, desenvolvidos para a completude desse sistema na figura Figura 4.

Basicamente, existem dois módulos que operam o programa como um todo: o módulo

Figura 4 – Diagrama do sistema



Fonte: Do autor

Lucky Rabbit (responsável pelo desenho vetorial) e o módulo **Steamboat** (responsável pela animação dos desenhos). O módulo de desenho vetorial recebe entradas do mouse ou da caneta do *tablet* de desenho e mostra, em tempo real, o desenho resultante da combinação do clique do mouse ou toque da caneta, junto ao movimento do dispositivo nos limites da área de desenho. Cada uma das alterações marcadas nos desenhos realizados, alimenta o módulo de animação que constrói uma sequência de imagens de acordo com o conjunto de alterações realizados no desenho. Esses entradas vão para a saída gráfica de cada sistema operacional, que fica com a responsabilidade de mostrar a renderização gráfica na tela e gerar os arquivos de imagem e vídeo, para armazenamento no sistema operacional. O módulo de desenho vetorial é explicado mais profundamente na próxima seção.

3.2 Subsistema de desenho vetorial

O módulo responsável pelo desenho vetorial é o módulo que trabalhará diretamente com a interação principal do usuário, que é criar as formas que compõem o desenho. Este módulo é nomeado Lucky Rabbit em homenagem ao personagem *Oswald The Lucky Rabbit*, criado por *Walt Disney* e distribuído pela *Universal Studios* nas décadas de 1920 e 1930 (THOMAS; JOHNSTON, 1981). Nas subseções a seguir são elucidados todos os item que compõem esse módulo.

3.2.1 Desenvolvimento da área de desenho

A principal interação do usuário com o sistema se dá através do clique, ou pressão da caneta sob a mesa digitalizadora, com o posterior movimento do mouse, ou movimento da caneta sob a superfície do tablet. Esses movimento são capturados como eventos pelo sistema.

Os eventos, então, capturam as coordenadas do dispositivo utilizado e uma função de desenho renderiza as formas que o usuário pretende inserir. A renderização acontece em uma tela branca, denominada no sistema como Canvas, representada por uma classe de mesmo nome.

A classe canvas é composta de algumas propriedades que possibilitam a renderização do desenho, entre elas, as medidas de altura e largura do canvas, o estilo do pincel (que define o tracejado e largura das linhas dos desenhos), botões para a função de desenho, de modificação das formas e de execução da animação, uma lista de algoritmos de interpolação para geração da animação, uma linha do tempo com um botão deslizante para representação dos quadros, um menu para abrir e salvar trabalhos e as respectivas funções utilizadas para cada componente. O código da classe Canvas segue no próximo item.

```
class Canvas : public QWidget
{
  Q_OBJECT
public:
  QImage image;
  QList<Point> points_list;
  void setDragged(bool dragged) { this->dragged = dragged; }
  bool isDragged() const { return this->dragged; }
  void setFirstPoint(Point *point);
  Point getFirstPoint()const { return this->first_point; }
  void setLastPoint(Point *point);
  Point getLastPoint() const { return this->last_point; }
protected:
  void mousePressEvent(QMouseEvent *event) override;
  void mouseMoveEvent(QMouseEvent *event) override;
  void mouseReleaseEvent(QMouseEvent *event) override;
  void paintEvent(QPaintEvent *event) override;
private:
  bool dragged;
  Point first_point;
  Point last_point;
  QPen pen;
  void drawLineBetweenPoints();
signals:
public slots:
};
```

Essa classe é instanciada pela classe Lucky Rabbit, que é a executada quando o programa roda. As subseções a seguir explicam um pouco mais de cada componente, que permitem que os desenhos sejam renderizados na tela.

3.2.2 Renderização das formas desenhadas

A composição do módulo de desenho vetorial segue a modelagem de classes descrita anteriormente, sendo que a estrutura mais fundamental para representação dos desenhos é a classe Ponto. A nomenclatura segue a definição da geometria analítica, onde o ponto pode denominado como uma representação de uma posição no espaço, o qual determinamos a partir de suas coordenadas. Como o desenho situa-se em um plano bidimensional, as coordenadas do ponto são os valores do par ordenado (x,y), sendo x a distância horizontal da origem (0,0) e y a distância vertical da origem (0,0). A origem, no caso da tela de desenho, situa-se no canto superior esquerdo. A implementação da classe Ponto é mostrada no trecho de código a seguir.

```
struct Point {
  int x, y;
};
```

O programa trabalha com vários pontos obtidos pelo desenho realizado com o botão esquerdo do mouse ou pelo tablet de desenho. A partir do clique do mouse sob a tela de desenho, ou encostando a caneta digital na superfície do tablet de desenho, um primeiro ponto é instanciado. Se o usuário mantém o botão do mouse pressionado, ou se ele mantém a caneta pressionada no tablet e movimentar o dispositivo com o qual estiver desenhando, um novo ponto é instanciado. Esses dois pontos, o primeiro que é instanciado no clique do mouse ou na pressão da caneta, e o último são inseridos em uma *QList* da classe Ponto, para representação da forma que está sendo desenhada. As funções implementadas para obtenção dos eventos acionados pelo usuário e a inserção do ponto na lista são demonstrados nos trechos de código a seguir.

```
void Canvas::mousePressEvent(QMouseEvent *event)
    {
      if(event->button() == Qt::LeftButton) {
        dragged = true;
        this->first_point.x = event->pos().x();
        this->first_point.y = event->pos().y();
      }
    }
```

Onde cada vez que o usuário pressiona o botão esquerdo (este evento também serve para o momento que a mesa de desenho for tocada pela caneta) as coordenadas do dispositivo são atribuídas para as coordenadas (x,y) do primeiro ponto. A *flag* que indica que o dispositivo foi movido é definida para o valor *true* (verdadeiro). Com isso, a função de movimento é definida a seguir.

```
void Canvas::mouseMoveEvent(QMouseEvent *event)
    {
      if(this->dragged) {
        this->last_point.x = event->pos().x();
        this->last_point.y = event->pos().y();
        this->points_list.append(first_point);
        this->drawLineBetweenPoints();
      }
    }
```

No evento de movimento do mouse, o último ponto, que representa o segundo par ordenado da reta a ser desenhada, é definido para os valores da posição do dispositivo. O primeiro ponto é adicionado na lista de pontos para a representação da figura em memória. Por fim, o evento chama a função para desenhar a linha entre os pontos, para criar a reta até a coordenada movida.

```
void Canvas::mouseReleaseEvent(QMouseEvent *event)

    {
      if(this->dragged) {
        this->last_point.x = event->pos().x();
        this->last_point.y = event->pos().y();
        this->points_list.append(last_point);
        this->drawLineBetweenPoints();
        this->dragged = false;
      }
    }
```

Quando o usuário solta o botão esquerdo ou afasta a caneta da mesa de desenho, o último ponto é atualizado para os novos valores de coordenadas do dispositivo e adicionado na lista de pontos para que todos os pares ordenadas estejam na lista. A função de desenho de retas é chamada de novo e a *flag* de movimento do mouse é definida para o valor *false* (falso).

Para o desenho efetivo da forma que o usuário deseja inserir na tela, é aplicada uma função para todos os pares sequenciais dos pontos obtidos dos eventos do mouse e da caneta digital. Os parâmetros dessa função são dois pontos, denominados ponto *P0* e *Pn*, sendo *n* sempre *1*. A cada evento de movimento do dispositivo de entrada, essa função é chamada passando o ponto inicial, que pode ser o ponto obtido no momento do clique ou da pressão da caneta ou pode ser o ponto relativo à ultima coordenada da qual foi movido o dispositivo. O outro objeto passado é o ponto atual onde foi movido o mouse ou caneta. A função, então, desenha uma linha entre esses dois pontos. O pseudocódigo a seguir representa a ideia geral dessa função.

Algoritmo 1: Desenho de retas entre pontos

Input: Os pontos $P0(x0,y0)$ e $Pn(xn,yn)$
Output: Uma reta $R(P0,Pn)$ entre os pontos $P0(x0,y0)$ e $Pn(xn,yn)$
desenha reta R nas coordenadas ($P0$, Pn)
atualiza a tela
$P0 \leftarrow Pn$

Onde o ponto *P0* é o primeiro ponto e o ponto *Pn* é o último ponto, os quais são passados para a função de desenho de retas como coordenadas. Depois de atualizada a tela, as coordenadas do primeiro ponto são redefinidas para as coordenadas do último. Como a função é chamada durante a pressão da caneta ou do botão esquerdo do mouse pressionado, isso garante que a linha seja atualizada conforme o usuário for movendo o dispositivo de entrada, sendo que o último ponto sempre será relativo à coordenada atual do ponteiro e o primeiro ponto seja o último ponto da reta desenhada posteriormente.

Esse pseudocódigo é implementado com o Qt utilizando os recursos de desenho vetorial dispostos pela própria biblioteca. A seguir, o código da função implementada é demonstrado.

```
void Canvas::drawLineInBetweenPoints()
    {
      QPainter painter(&this->image);
      painter.setPen(this->pen);
      painter.drawLine(first_point.x, first_point.y, last_point.x, last_point.y);
      update();
      this->first_point.x = this->last_point.x;
      this->first_point.y = this->last_point.y;
    }
```

O código segue a definição do algoritmo, desenhando com o objeto *painter* da classe *QPainter* disponível no *Qt*. A caneta é configurada na execução do programa com valores padrões e pode ser redefinida pelo usuário, configurando a largura do tracejado.

3.2.3 Armazenamento do desenho no sistema de arquivos

Os trabalhos de desenho realizados no sistema podem ser armazenados no sistema de arquivos do sistema operacional. Os formatos disponíveis para salvar os desenhos são: *PNG*, *SVG* e *JPEG*. Como se trata de um programa que cria e edita imagens vetoriais, o formato *SVG* é recomendado para que se mantenha as informações geométricas e a qualidade da imagem, da forma como ela foi desenhada. O formato *SVG* é um arquivo que é excrito em uma linguagem de marcação bem simples, baseada em *XML*. O código que cria um arquivo *SVG*, utiliza a classe *QSvgGenerator* do *Qt* para criar o arquivo *SVG* a partir do objeto *imagem* que representa todos os desenhos da tela, conforme pode ser observado no seguinte código.

```
boolean LuckyRabbit::saveAsSvg()
        {
          QString path = QFileDialog::getSaveFileName(this, "Salvar como SVG", this->path,
tr("SVG files (*.svg)"));
          QSvgGenerator svg_generator;
          if (path.isEmpty()) {
            return false;
          } else {
            generator.setFileName(path);
            generator.setSize(canvas->size());
            generator.setViewBox(canvas->image);
          }
        }
```

Para fins de compatibilidade com outros programas de edição e visualização de imagens, os formatos *PNG* e *JPEG* também estão disponíveis para o usuário. No caso do formato *PNG*, se trata de um formato portável de imagens, com compactação de dados sem perda, sendo uma representação raster bem fiel as figuras desenhadas.

O formato *JPEG*, por sua vez, comprime mais a imagem com alguma perda. É um formato ideal para salvar as imagens com menos bytes de armazenamento. O código que gera os arquivos *PNG* e *JPEG* é mostrado a seguir, com o adendo de que as funções de criação de imagens nesse formato disponíveis no *Qt* são utilizadas a fim de reutilizar o código do próprio *framework*.

```
bool LuckyRabbit::saveRaster()
        {
          QString path = QFileDialog:getSaveFileName(this, "Salvar como", this->path,
tr("Image files (*.png|*.jpg)"));
```

```
    if (image.save(path)) {
        return true;
    } else {
        return false;
    }
}
```

A opção de Salvar utiliza as funções de criação de imagens de acordo com o formato informado pelo usuário e exibe uma janela com o explorador de arquivos aberto para que o usuário defina uma pasta e o nome do arquivo. Já o item Abrir, do menu principal, também exibe o explorador de arquivos para que o usuário abra uma imagem no formato *SVG*. Como se trata de um programa de animação de desenhos vetoriais, o único formato disponível para manipulação é o *SVG*. A seguir, os códigos da função Abrir é mostrado.

```
bool LuckyRabbit::openImage()
    {
        QString path = QFileDialog:getOpenFileName(this, "Abrir", this->path, tr("SVG
files (*.svg)"));
        QImage image_opened;
        if (image_opened.load(path)) {
            image = image_opened;
            update();
            return true;
        } else {
            return false;
        }
    }
```

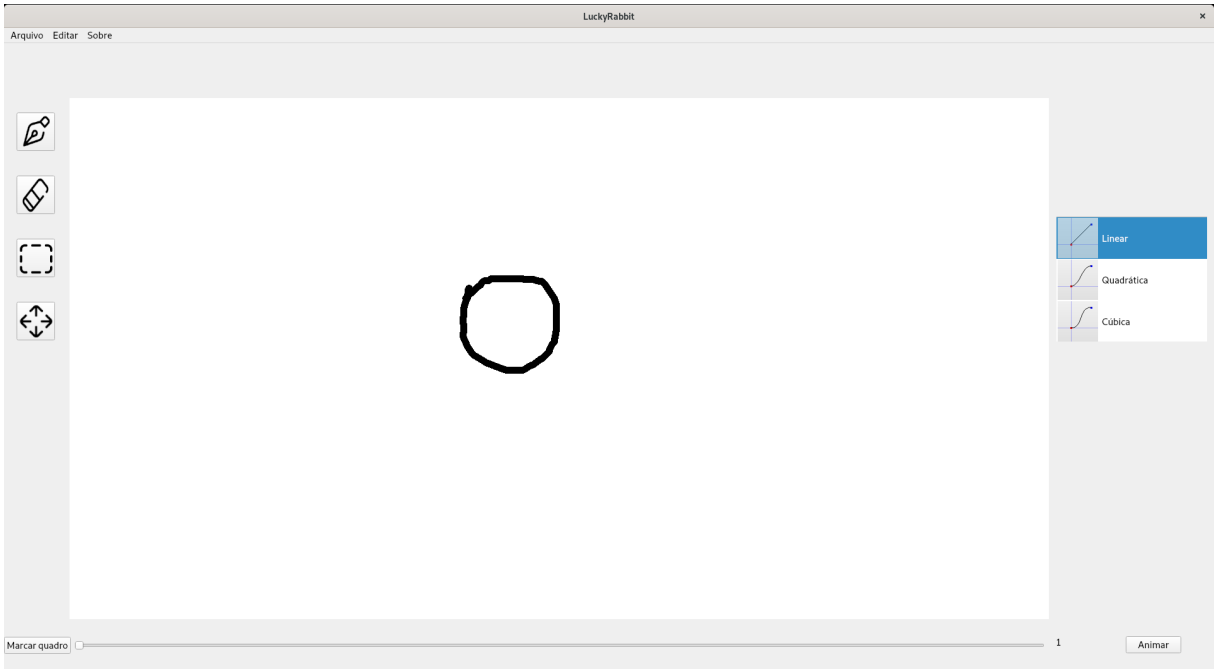
Para fins de clareza, a captura de tela na Figura 5 mostra o programa principal com todos os botões e ferramentas e um desenho realizado.

Na próxima seção é explicado o módulo de animação e suas propriedades.

3.3 Subsistema de animação

O módulo de animação é denominado **Steamboat** em homenagem ao curta animado *Steamboat Willie*, dirigido por *Walt Disney* e *Ub Iwerks*, produzido pela *Walt Disney Studios* e distribuído pela *Celebrity Productions*, no ano de 1928 (THOMAS; JOHNSTON, 1981). Esse módulo define algumas classes e componentes que são utilizados pelo próprio módulo e, também, algumas classes que são instanciadas pela classe Canvas, presente no módulo Lucky

Figura 5 – Captura de tela da operação de desenho



Fonte: Do autor

Rabbit.

As classes implementadas nesse módulo, são as representações dos quadros de animação, classes com operações de manipulação das formas desenhadas, algoritmos de interpolação e a estrutura para conter os quadros de animação. Nas próximas subseções, são explicados mais claramente sobre cada um desses componentes.

3.3.1 Estrutura e funções para manipulação dos desenhos

Para que a animação dos desenhos seja gerada corretamente, é necessário que as formas desenhadas sejam manipuladas de alguma forma. Essas alterações devem ocorrer da seguinte forma: movendo o botão deslizador da linha do tempo, o animador indicará um novo quadro no tempo onde será realizada a modificação. Clicando no botão "Mover", o programa entra em modo de seleção de pontos a serem movidos. Como cada forma tem um conjunto de pontos associados, estruturados em uma QList de pontos, qualquer lugar selecionado sob o tracejado do desenho pode ser selecionado e movido. Quando o usuário clica sob o tracejado da curva linha ou forma, o ponto onde foi clicado fica selecionado, exibindo um círculo vermelho preenchido, indicando que aquele ponto pode ser movido. Neste caso, segurando o clique, o usuário pode mover o ponto para qualquer direção que desejar. Neste caso, a função que desenha as linhas entre os pontos será chamada duas vezes para recalcular a nova reta. A primeira enviando o ponto na QList anterior ao ponto movido e o próprio ponto alterado. Depois, é enviado o ponto movido e o ponto imediatamente posterior ao movido. Desta forma, as duas retas são redesenhadas corretamente.

O usuário pode, também, selecionar a figura inteira a ser movida ou parte dela. Clicando no botão de "Seleção", o usuário pode clicar com o botão esquerdo (ou pressionar a caneta de desenho) em qualquer área, arrastar o mouse/caneta e soltar o botão (ou afastar a caneta do tablet), para delimitar uma área retangular a ser selecionada. As figuras ou pontos que ficarem dentro dessa área estarão selecionáveis, ficando com a cor vermelha no tracejado, tanto da figura (ou figuras) inteiras, quanto das retas que compõem a figura que foram selecionadas. Neste caso, é só colocar o dispositivo em cima do tracejado selecionado, em qualquer lugar e a figura ou retas moverão para qualquer direção que for arrastado e soltado o dispositivo. Todos os pontos que estiverem dentro da área de seleção são recalculadas utilizando um iterador.

O próximo trecho de código demonstra a função Transladar, dentro da classe Transformação, do módulo Steamboat.

```
void Transformation::translate(int delta_x, int delta_y)
        {
          QList<Point>::iterator it;
          for(it = this->selected_points.begin(); it != this->selected_points.end(); ++it) {
            *it.x += delta_x;
            *it.y += delta_y;
            canvas->update();
            this->x = delta_x;
            this->y = delta_y;
          }
        }
```

A função pega o conjunto de pontos selecionados (que podem ser vários ou apenas um ponto, dependendo da seleção) e soma com o deslocamento da posição do mouse ou caneta. Esse deslocamento é calculado com as três funções de evento listadas a seguir.

```
void Transformation::mousePressEvent(QMouseEvent *event)
        {
          if(event->button() == Qt::LeftButton) {
            if(this->move) {
              this->pressed = true;
              this->x = event->pos().x();
              this->y = event->pos().y();
              if(this->selected_points.size()==0) { //Se a lista estiver com mais de um
elemento, significa que já foram selecionados com a caixa de seleção
                  if(std::any_of(begin(this->points_list), end(this->points_list), [](Point const
&p)  return p.x == this->x && p.y == this->y; ))
```

```
                this->selected_points.append(p);
            }
        } else {
            qDeleteAll(this_selected_points);
            this->box_select_origin = event->pos();
            this->rubber_band = new QRubberBand(QRubberBand::Rectangle, this);
            this->rubber_band->setGeometry(QRect(this->box_select_origin, QSize()));
            this->rubber_band->show();
        }
    }
}
```

O evento de pressão é implementado, utilizando uma *flag* para indicar se o evento tratará do movimento do(s) ponto(s) (valor verdadeiro) ou se tratará da seleção dos pontos. Caso trate da manipulação de um único ponto, é adicionado na lista de seleção de pontos apenas o ponto que coincide com a posição do dispositivo quando o evento ocorrer. Caso contrário (valor falso), a transformação está em modo de seleção por caixa, então, primeiramente são removidos todos os elementos que estiverem na lista de pontos selecionado e é definido a origem da seleção na posição onde ocorreu o evento de dispositivo e criado o retângulo de seleção nessa origem, com o auxílio da classe *QRubberBand* do Qt.

```
void Transformation::mouseMoveEvent(QMouseEvent *event)
{
    if(this->pressed) {
        if(this->move) {
            int delta_x = event->pos().x();
            int delta_y = event->pos().y();
            delta_x -= this->x;
            delta_y -= this->y;
            this->translate(delta_x, delta_y);
        } else {
            this->rubber_band->setGeometry(QRect(this->box_select_origin, event->pos().normalized
        }
    }
}
```

O evento de movimento do mouse calcula o deslocamento em x e y do dispositivo, subtraindo a posição atual do movimento com a posição anterior. Esses valores são passado como parâmetro para a função de translação para todos os elementos da lista de pontos, seja

um só ou vários pontos. Caso o modo esteja em seleção (com o valor da *flag* de movimento em falso), o retângulo de seleção é desenhado na posição normalizada do dispositivo.

```
void Transformation::mouseReleaseEvent(QMouseEvent *event)
{
  if(this->pressed) {
    if(this->move) {
      int delta_x = event->pos().x();
      int delta_y = event->pos().y();
      delta_x -= this->x;
      delta_y -= this->y;
      this->translate(delta_x, delta_y);
      this->pressed = this-> move = false;
      qDeleteAll(this->selected_points);
    } else {
      this->rubber_band->hide();
      for(it = this->points_list(); it != this->points_list.end(); ++it) {
        if(QRect::contains(QPoint(*it.x,*it.y))
          this->selected_points.append(*it);
      this->move = true;
      }
    }
  }
}
```

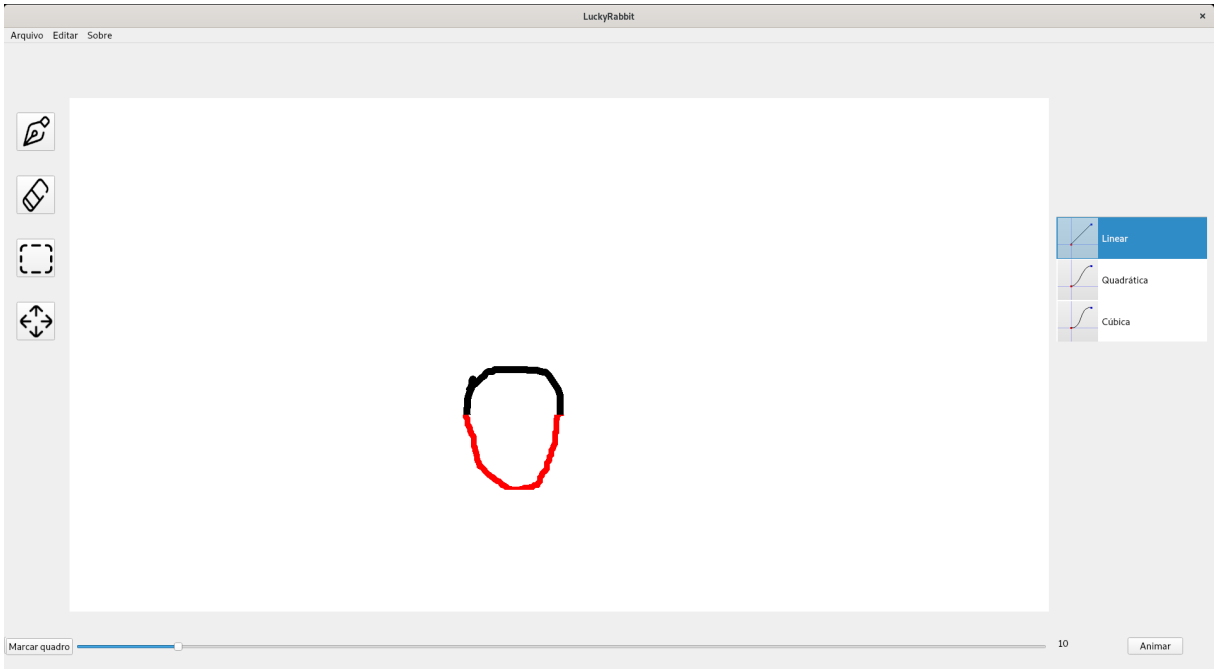
Por fim, quando o dispositivo entra no evento de soltura, os pontos são transladados para as posições atuais e as *flags* são definidas para o valor falso. Caso o modo esteja em seleção, a caixa de seleção é ocultada e a lista de pontos é iterada, comparando se as posições estão dentro dos limites da caixa de seleção e, neste caso, insere-se esses pontos na lista de pontos selecionados e a *flag* de movimento é definida para verdadeiro, afim de que a próxima verificação de evento de pressão já realize o movimento dos pontos selecionados. A manipulação dos desenhos por meio dessas ferramentas é demonstrado na Figura 6.

Após isso, é necessário marcar que a operação foi feita, a fim de que a animação possa ser gerada. Na próxima subseção, é explicado um pouco mais sobre como funciona cada quadro de animação dentro do sistema.

3.3.2 Estruturas para representação dos quadros

A classe Quadro, no módulo Steamboat, é a representação de cada quadro de animação presente no projeto. Como o número de quadros é um campo personalizável, a instanciação de quadros depende da quantidade de quadros-chave definida pelo usuário, não devendo ultrapassar o número de quadros disponíveis pela linha do tempo, sendo o padrão, quando do início da

Figura 6 – Captura de tela da manipulação do desenho



Fonte: Do autor

execução do programa, de 120 quadros. Como a animação gerada é de 24 quadros por segundo, essa quantidade padrão é relativa a 5 segundos de animação. De qualquer forma, o campo de quadros totais é personalizável, então o usuário pode aumentar ou diminuir conforme a necessidade.

Essa classe tem um identificador, indicando qual número sequencial que o quadro se encontra dentro do conjunto de quadros e uma QList de todas as formas, se tratando de uma lista do tipo *Point* já que esse tipo é utilizado como um container para os desenhos. A classe Quadro pode ser melhor entendida quando comparada com cada folha de acetato, utilizado na animação tradicional, que trazia consigo os desenhos e um número sequencial para identificar o tempo em que o desenho que está na folha se relaciona com a animação como um todo.

A implementação da classe Quadro é mostrada no próximo trecho de código.

```
struct Frame {
  int id;
  QList<Point> shapes;
}
inline bool operator<(const Frame &first_frame, const Frame &second_frame)
{
  return first_frame.id < second_frame.id;
}
```

A classe que representa o quadro tem uma função implementada para o operador "<". Essa operação é utilizada para o tipo *std::set* que conterá o conjunto de quadros da animação.

Cada vez que o usuário cria um quadro-chave, esse quadro é inserido em um conjunto do tipo Quadro. Esse conjunto de quadros é a que será passada para as funções de interpolação, escolhida pelo usuário. O botão de criação de quadro-chave fica próximo a linha do tempo. As funções associadas a esses componentes são as seguintes.

```
void LuckyRabbit::addKeyFrame(Frame *frame)
        {
          this->frame_set.insert(frame);
        }


void LuckyRabbit::changeFrame(Frame *frame)
        {
          this->setImage(frame.image);
        }
```

A função de inserção de quadro-chave adiciona, no conjunto de quadros, o quadro selecionado pelo botão de inserção de quadro-chave. Já a função de troca de quadro, associada ao botão deslizante da linha do tempo, troca o quadro ativo do programa, colocando o conteúdo da imagem relativa ao índice do quadro.

Na próxima subseção, são explicados os algoritmos de interpolação utilizados para geração dos quadros intermediários.

3.3.3 Algoritmos de interpolação

A lista de quadros-chave é passada como parâmentro para as funções de interpolação. A classe Animação, que herda a classe QPropertyAnimation do Qt 5, define as funções de interpolação que podem ser usadas para a geração de quadros-chave. Essas funções podem ser de três tipos distintos, sendo: Linear, Quadrática e Cúbica. Essas funções definem a curva paramétrica que será aplicada para a geração de quadros intermediários. De acordo com a forma da curva, o intervalo entre cada quadro será diferente. A classe tem o tipo de curva que será aplicada, utilizando um Enum com valores Linear, Quadrática e Cúbica (PARENT, 2012) e (AZEVEDO; CONCI, 2003), e as funções relativas a cada tipo de curva.

Na função de interpolação linear, cada ponto *P'(F(x),F(y))* é dado pelas funções da Equação (1) e da Equação (2).

$$F(x) = (1 - t)x + tx \tag{1}$$

$$F(y) = (1 - t)x + tx \tag{2}$$

Onde *t* é o parâmetro no intervalo *[0,1]* correspondente ao momento temporal intermediário entre os quadros-chave. A implementação no código fica da seguinte maneira.

```
Point Steamboat::linear_interpolation(Point first_point, Point second_point, float t_param)
    {
      Point lerp_point;
      lerp_point.x = (int)((1 - t_param) * first_point.x + t * second_point.x);
      lerp_point.y = (int)(1 - t_param) * first_point.y + t * second_point.y);
      return lerp_point;
    }
```

Na interpolação quadrática, a definição é bem parecida. A mudança fica por conta do parâmetro *t* que não é mais constante, mas sim, descontínuo, em função da exponenciação em base 2, conforme pode ser visto na Equação (3) e na Equação (4).

$$F(x) = (1 - t^2)x + t^2x \tag{3}$$

$$F(y) = (1 - t^2)y + t^2x \tag{4}$$

No código, a implementação segue parecida com a interpolação linear.

```
Point Steamboat::quadratic_interpolation(Point first_point, Point second_point, float t_param)
    {
      Point quad_point;
      quad_point.x = (int)((1 - pow(t_param,2)) * first_point.x + pow(t,2) * second_point.x);
      quad_point.y = (int)(1 - pow(t_param,2)) * first_point.y + pow(t,2) * second_point.y);
      return quad_point;
    }
```

Por último, a interpolação cúbica é bastante intuitiva, uma vez que o parâmetro *t* é elevado a 3 e as equações Equação (5) e Equação (6) seguem, basicamente, da mesma forma.

$$F(x) = (1 - t^3)x + t^3x \tag{5}$$

$$F(y) = (1 - t^3)y + t^3x \tag{6}$$

E a implementação, segue no seguinte trecho.

```
Point Steamboat::cubic_interpolation(Point first_point, Point second_point, float t_param)
    {
```

```
    Point cub_point;
    cub_point.x = (int)((1 - pow(t_param,3)) * first_point.x + pow(t,3) * second_point.x);
    cub_point.y = (int)(1 - pow(t_param,3)) * first_point.y + pow(t,3) * second_point.y);
    return cub_point;
}
```

As curvas ficam dispostas numa lista, no canto direito da tela, conforme pode ser visto na Figura 5 e na Figura 6. Após selecionada a curva paramétrica relativa a função de interpolação que se deseja aplicar, o usuário pode gerar a animação, onde cada par de pontos de cada dois quadros sequenciais são enviados para função de interpolação escolhida para gerar o quadro intermediário. Após isso, os quadros intermediários são adicionados no conjunto de quadros da animação, prontos para serem executados ou armazenados, como pode ser visto na próxima seção.

3.3.4 Armazenamento da animação no sistema de arquivos

Ao final da geração da animação, na janela de execução da animação é apresentada com a opção de salvar a animação gerada como um vídeo ou como uma sequência de imagens.

Para gerar a sequência de imagens, é utilizado uma lista do tipo *QPixmap*, com os caminhos de imagem a serem salvas. Então é criado uma instância de *QPainter*, enviando as coordenadas das linhas para cada *QPixmap*, gerando a sequência de imagens. A função a seguir demonstra como é criada a imagem para cada lista de pontos.

```
void renderImageSequence(QList<Points> points_list, QPixmap pixmap)
{
    QPainter painter(&pixmap);
    painter.setPen(this->pen);
    for(int i = 0; i < points_list.size(); i++) {
        painter.drawLine(points_list[i]);
    }
}
```

Para cada lista de par de pontos nos quadros, é passada uma função para criar a sequência de imagem relativa ao quadro onde a função será aplicada. A instância do *QPainter* é criada com o endereço de memória do *QPixmap*. Como o *QPixmap* é criado com o caminho do arquivo de imagem, o *QPainter* renderizará as linhas direto para o arquivo de imagem. O usuário define o nome dos arquivos a partir do botão de salvar no menu principal.

Para a geração do vídeo, é utilizado a biblioteca *libqtavi*, onde as imagens são organizadas como vídeo no formato AVI. Cada quadro é inserido em uma instância de QAviWriter,

conforme o código a seguir.

```
bool renderVideoFile(QList<QString> image_sequence_files, QString video_file)
        {
          QAviWriter avi_writer(video_file, QSize(this->image->size(), 24, "MJPEG");
          if(avi_writer.open()) {
            for(std::QList<String>::iterator it; it = image_sequence_files.begin(); it !=
image_sequence_files.end(); ++it) {
              avi_writer.addFrame(QImage(*it));
            avi_writer.close();
          } else {
            return false;
          }
        }
```

Uma lista de nomes de arquivos é passada para a função de criação de vídeo, de acordo com os nomes de arquivo que o usuário definir. Cada imagem é passada para a função *addFrame()* da classe *QAviWriter*, que fica responsável por criar o vídeo de acordo com a sequência de imagens.

Com a geração da sequência de imagens ou com a exportação para vídeo, o usuário conclui o fluxo do processo de animação.

4 CONCLUSÃO

Neste trabalho foi apresentado o projeto e implementação de um sistema de animação de desenhos vetoriais. O sistema permite a criação de desenhos vetoriais à mão livre, utilizando tanto o mouse quando uma mesa digitalizadora em conjunto com a caneta digital. As formas desenhadas pelo usuário podem ser animadas, criando os quadros-chave do movimento desejado, a partir das transformações que podem ser aplicadas aos desenhos. Esses quadros-chave alimentam um sistema de interpolação, no qual o usuário pode escolher o método desejado de interpolação (que pode ser Linear, Quadrática ou Cúbica) e os quadros intermediários são criados automaticamente para compor o movimento completo. A animação é reproduzida em uma janela separada e é facultado ao usuário a opção de salvar o vídeo como um arquivo de vídeo ou como uma sequência de imagens.

Com a implementação desse sistema, foi possível alcançar os objetivos de criar uma ferramenta gráfica que permita ao artista se expressar livremente ao mesmo tempo que automatiza algumas tarefas mais maçantes, sempre oferecendo um leque de opções para a criatividade do usuário ser aplicado da forma que mais desejar. Utilizando desenhos vetoriais, pode-se usufruir de todas as vantagens que esse método gráfico provê, como a rapidez de aplicar transformações geométricas e a versatilidade de utilizar um conjunto de pontos para definir as coordenadas das imagens. Com o método de desenho, utilizando linhas entre os pontos advindos das coordenadas do dispositivo de entrada usado, o usuário nem percebe que não se trata do mesmo método utilizado por um programa do tipo *Raster*, como o *Microsoft Paint* (CORPORATION, 2018), que é o método mais usual para esse tipo de software. Após isso é oferecido uma forma de manipulação simples das imagens, podendo mover os pontos do desenho ou um conjunto de linhas, a fim de mudar as propriedades das formas em cada quadro para criar a ilusão de movimento. Por fim, o usuário pode escolher o método de interpolação mais adequado para a sua animação, para que o sistema automatize da maneira correta o processo de criar a sequência de transformações e gerar a animação desejada.

Para a posteridade e enriquecimento da plataforma, seria interessante a adição de mais opções de interpolação e uma curva de interpolação customizada, para que o usuário defina seu próprio método e ter mais controle criativo. O programa podia oferecer, também, a colorização do desenho, para já criar um tratamento final para o vídeo, sem depender de mais ferramentas para isso. Outras adições pertinentes seria a possibilidade de adicionar áudio às animações e oferecer um ambiente de edição e composição dos desenhos. Com esse tipo de implementação, o programa se tornaria ainda mais completo e poderia ser utilizado em todo o processo de criação de animações.

Referências

AZEVEDO, E.; CONCI, A. **Computação gráfica: teoria e prática**. Elsevier, 2003. ISBN 9788535212525. Disponível em: <https://books.google.com.br/books?id=Y1mhBAAACAAJ>. Citado 3 vezes nas páginas 2, 3 e 31.

COMPANY, Q. **www.qt.io**. 2018. Disponível em: <https://www.qt.io>. Acesso em: 03 de dezembro de 2018. Citado na página 18.

CORPORATION, M. **www.microsoft.com/en-us/p/paint-3d/9nblggh5fv99?activetab=pivot:overviewtab**. 2018. Disponível em: <https://www.microsoft.com/en-us/p/paint-3d/9nblggh5fv99?activetab=pivot:overviewtab>. Acesso em: 02 de dezembro de 2018. Citado na página 35.

FIUME, E. L. **The Mathematical Structure of Raster Graphics**. 1. ed. Toronto: Academic Press, 1989. 236 p. Citado na página 2.

FOWLER, M. **UML Distilled: A Brief Guide to the Standard Object Modeling Language**. 3. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321193687. Citado 3 vezes nas páginas 8, 9 e 15.

HEALEY, C. G. et al. Perceptually based brush strokes for nonphotorealistic visualization. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 23, n. 1, p. 64–96, jan. 2004. ISSN 0730-0301. Disponível em: <http://doi.acm.org/10.1145/966131.966135>. Citado 2 vezes nas páginas 1 e 2.

LEUNG, J.; LARA, D. M. Grease pencil: Integrating animated freehand drawings into 3d production environments. In: **SIGGRAPH Asia 2015 Technical Briefs**. New York, NY, USA: ACM, 2015. (SA '15), p. 16:1–16:4. ISBN 978-1-4503-3930-8. Disponível em: <http://doi.acm.org/10.1145/2820903.2820924>. Citado na página 2.

PARENT, R. **Computer Animation: Algorithms and Techniques**. 3. ed. Waltham: Morgan Kaufmann, 2012. 542 p. Citado 2 vezes nas páginas 1 e 31.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software - 8ª Edição**. [s.n.], 2016. ISBN 9788580555349. Disponível em: <https://books.google.com.br/books?id=wexzCwAAQBAJ>. Citado 2 vezes nas páginas 5 e 8.

SOMMERVILLE, I. **Software Engineering**. 9th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0137035152, 9780137035151. Citado 2 vezes nas páginas 7 e 9.

SZABO, P. W. **User Experience Mapping**. [S.l.]: O'Reilly, 2017. ISBN 9781787123502. Citado na página 5.

THOMAS, F.; JOHNSTON, O. **The Illusion of Life: Disney Animation**. 1. ed. New York: Disney Editions, 1981. 578 p. Citado 2 vezes nas páginas 19 e 25.

VAZQUEZ, C.; SIMÕES, G. **Engenharia de Requisitos: software orientado ao negócio**. BRASPORT, 2016. ISBN 9788574527901. Disponível em: <https://books.google.com.br/books?id=gA7kDAAAQBAJ>. Citado na página 5.

WINKENBACH, G.; SALESIN, D. H. Computer-generated pen-and-ink illustration. In: **Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques**. New York, NY, USA: ACM, 1994. (SIGGRAPH '94), p. 91–100. ISBN 0-89791-667-0. Disponível em: <http://doi.acm.org/10.1145/192161.192184>. Citado 2 vezes nas páginas 1 e 2.