Alfredo José de Paula Barbosa - 16890
Michell Stuttgart Faria - 16930
Paulo Vicente Gomes dos Santos - 15993

SAGA Game Library
Biblioteca para desenvolvimento de jogos eletrônicos 2D

Itajubá - MG

27 de Junho de 2014

Alfredo José de Paula Barbosa - 16890
Michell Stuttgart Faria - 16930
Paulo Vicente Gomes dos Santos - 15993

SAGA Game Library
Biblioteca para desenvolvimento de jogos eletrônicos 2D

Monografia apresentada para o Trabalho de Diploma do curso de Engenharia da Computação da Universidade Federal de Itajubá.

Orientador: Prof. Dr. Enzo Seraphim

Universidade Federal de Itajubá - UNIFEI
Instituto de Engenharia de Sistemas e Tecnologias da Informação
Engenharia da Computação

Itajubá - MG

27 de Junho de 2014

"A maravilhosa disposição e harmonia do universo só pode ter tido origem segundo o plano de um Ser que tudo sabe e tudo pode. Isso fica sendo a minha última e mais elevada descoberta".

Principia, Book III, Isaac Newton

Agradecimentos

A cada mestre das nossas escolas e famílias, que nos guiaram na construção do conhecimento, dos átomos e das células, dos números e das palavras, dos fenômenos da natureza e dos mistérios da alma, da convivência, da responsabilidade e da dedicação. A cada amigo e familiar que estava lá, tanto nas lutas quanto nas vitórias, como esta, pela graça de Deus, de quem é toda a glória.

Sumário

Lista de Figuras

Resumo

Uma biblioteca para jogos ou *game engine* pode ser vista como um conjunto de recursos e ferramentas para a construção de um jogo. Você pode criar um jogo sem uma biblioteca básica, assim como você pode criar uma mesa de madeira sem martelos, chaves de fenda e serras, mas as vantagens que essas ferramentas proporcionam justificam chamá-las de necessárias.

O nível dessas ferramentas varia: algumas *engines* se limitam a códigos, ou seja, constantes, variáveis, funções e classes relacionadas, mas outras contam com interfaces gráficas que possibilitam o desenvolvimento de um jogo sem codificação alguma. De qualquer forma, uma *game engine* oferece ferramentas para manipular arquivos de áudio e de imagem, e prove suporte a dispositivos de entrada (teclado, mouse, etc).

SAGA Game Library é uma *engine* voltada para estudantes da programação, especialmente a programação de jogos, pelo que foi feita priorizando a simplicidade a qualquer outra característica, usando a biblioteca ALLEGRO para acessar as funções básicas do sistema. Essa *engine* oferece algumas ferramentas básicas para o desenvolvimento de um jogo 2D, à medida em que essas não concorrem com a simplicidade da mesma.

Palavras-chave: *game engine*, jogos eletrônicos, ferramentas de desenvolvimento.

Abstract

A game engine is a set of game development resources and tools. One may create a game without a base engine, just like one can create a wooden table without hammers, screwdrivers and saws, but the advantages tools provide legitimate calling them necessary.

These tools' level vary: some engines are only about code, i.e. constants, variables, functions and related classes, but others come with graphic interfaces that enable the development of a game without any coding. Anyway, one game engine provides tools for manipulating audio files and images, and provides support for input devices (keyboard, mouse, etc.).

SAGA Game Library is an engine oriented to students of programming, specially game programming, which is why it was made focusing simplicity instead of any other feature, using ALLEGRO library to access the system's basic functions. This engine offers some game development basic tools, as long as these don't compete with its simplicity.

Keywords: game engine, electronic games, development tools.

1 Introdução

Desde os primórdios da humanidade a competição é uma forma de diversão muito popular. O objetivo de qualquer competição é testar uma habilidade individual ou grupal e destacar quem ganha. Embora essa característica ainda seja a mesma, a competição está condicionada a uma evolução que pode ser notada, por exemplo, na corrida: no começo ela era individual e só testava a velocidade e a resistência da pessoa; hoje uma corrida automobilística testa a resistência, a destreza, a inteligência e a tecnologia da equipe.

Mas essa evolução não se da só na complexidade da competição, ela se manifesta da mesma forma na sua abstração. Os jogos de estratégia que nós conhecemos hoje, por exemplo, são versões abstratas das competições físicas. A aptidão física de cada criatura imaginária é determinada por alguma característica interna do jogo, porque o que o jogo de estratégia testa é o raciocínio e a estratégia da pessoa e não a sua capacidade física propriamente dita. O xadrez internacional, simulando uma guerra da Idade Média, é um caso concreto dessa evolução.

Com o avanço da microeletrônica e da computação, no entanto, o jogo de estratégia ganha uma plataforma que pode simular não só um sistema de lógica, mas toda uma realidade virtual. Qualquer jogo pode ganhar uma versão eletrônica. O jogo eletrônico, portanto, não é só uma brincadeira de criança, ele é na verdade o último estágio de um passatempo milenar. Isso se confirma pela movimentação de recursos e ganhos da indústria dos jogos eletrônicos desta geração.

1.1 Motivação

Atualmente o mercado de *games* é consolidado como um dos principais dentro do segmento do entretenimento, junto das indústrias cinematográfica, fonográfica e literária, estando atrás apenas das indústrias bélica e automobilística. Além disso, é o que cresce mais rapidamente; à medida em que o hardware utilizado se torna mais poderoso, uma nova geração de *softwares* é desenvolvida, causando uma experiência nova de interação do jogador. Outros fatores importantes são as mudanças no desenvolvimento dos *games*, que passaram a ter grandes investimentos, e também o crescimento do público-alvo, influenciado pelos meios de comunicação, tendo os Estados Unidos como o principal mercado consumidor [GEDIGames 2010].

Esse setor tem carência de profissionais especializados e experientes. A maioria das empresas é muito jovem; os profissionais brasileiros mais experientes preferem trabalhar fora do país. Conhecidos pela técnica e pelo profissionalismo, além da paixão pelos jogos, eles acabam sendo contratados por empresas fora do país, e muitas vezes não voltam [GEDIGames 2010].

Há pouquíssimo interesse das empresas internacionais se instalarem no Brasil. Os motivos passam pela estrutura do mercado, onde a pirataria tem uma grande influência. Em 2006, o número de produtos piratas representava impressionantes 90% dos jogos comercializados. Outro agravante é a alta tributação, causando a desestruturação do mercado interno, a qual não é justificada, uma vez que não existem *hardwares* de *videogames* de última geração sendo produzidos no Brasil [GEDIGames 2010].

Apesar de tantos problemas, o Brasil é um mercado em potencial. O mercado está em ritmo acelerado de crescimento e os *videogames* são a principal forma de entretenimento para os brasileiros de todas as idades. De acordo com o Banco Nacional de Desenvolvimento Econômico e Social (BNDES), o setor de jogos eletrônicos nacional ainda precisa melhorar, principalmente no que diz respeito a incentivo às

empresas de *games*, a criarem mais propriedades intelectuais, que possam virar franquias e serem comercializadas, tanto aqui quanto no exterior. E ainda melhorar a qualidade profissional na formação e na oferta de empregos [GEDIGames 2010].

De fato, estão sendo feitos esforços para tornar o país mais receptivo para a indústria de games. Surgiram parcerias com empresas internacionais e também com universidades, para dar mais suporte e condições ao recém-formado para futuras vagas no mercado de trabalho. Também há implementação de políticas públicas, para que haja condições de desenvolvimento e proteção aos produtos nacionais. Hoje há 26 universidades com cursos que envolvem jogos, e o público alvo do mercado nacional tem mais de 45 milhões de usuários. Em jogos educativos, o Brasil já é uma referência mundial [GEDIGames 2010].

As empresas especializadas no ramo têm se multiplicado no País (Figura 1.1). Segundo estudo da Associação Brasileira dos Desenvolvedores de Games (ABRAGames), existem 220 empresas no país. Há pouco tempo atrás, em 2008, eram somente 48. O Brasil já é o quarto maior consumidor de jogos eletrônicos, atrás apenas de Ásia, Europa e Estados Unidos. Em contradição com a maioria dos países, o Brasil é o país onde o mercado de jogos eletrônicos mais cresceu em 2012, sendo que houve um aumento de 60% se comparado a 2011. Além disso, São Paulo é o maior mercado consumidor de games originais do mundo [GEDIGames 2010].

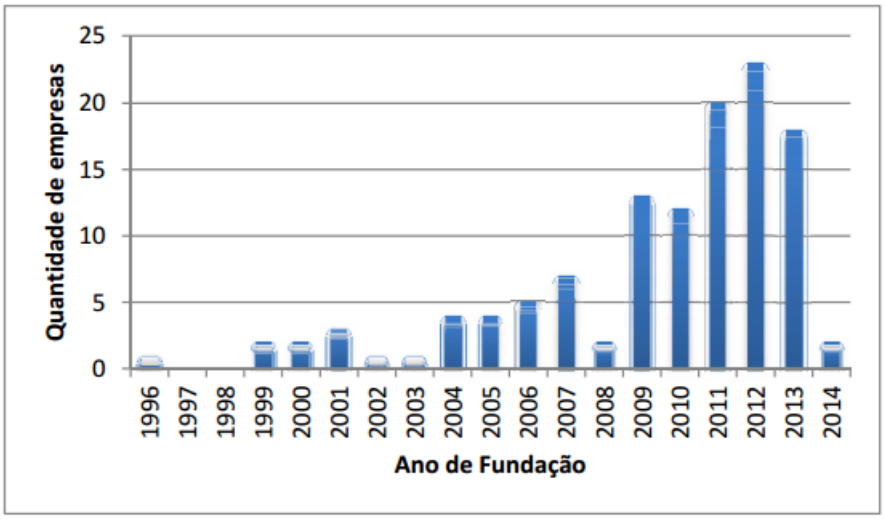
Em cerca de dez anos, reduzimos a pirataria de 90% para próximo de 50%, e na América Latina já somos o país com o melhor desempenho nesse combate. Fato é que há muito ainda a diminuir, mas essa redução já tem um impacto extremamente positivo no mercado de *games*. Cada dólar investido em *software* oficial injeta 437 dólares no setor [GEDIGames 2010].

Um fenômeno de destaque na popularização dos *games* é o uso das licenças virtuais no lugar das mídias físicas e das máquinas virtuais no lugar dos consoles específicos. Por um lado, tem-se o compartilhamento de conteúdo diretamente em lojas *online* e a redução do consumo de mídia física, já que a mídia digital está

ganhando cada vez mais espaço. Por outro lado os *games* para *tablet*, celulares e jogos *online* devem continuar crescendo, por serem opções de baixo custo e que rodam em dispositivos que o usuário já possui [LARA 2014].

Esses fatores inevitavelmente ajudam no combate à pirataria, fazendo com que o usuário prefira comprar o produto original a investir em um produto não-original. Essa mudança de comportamento por parte do consumidor fez com que um mercado, que antes era visto como inseguro, passasse a ser considerado promissor pelas empresas desenvolvedoras de *software*, incluindo as desenvolvedoras de *games*.

Figura 1.1: Quantidade de empresas fundadas por ano.



Fonte: Gráfico retirado de [GEDIGames 2010].

Com o crescimento da área de desenvolvimento de jogos eletrônicos, surge também a necessidade de encontrar mão-de-obra capacitada, necessidade essa que é uma das maiores reclamações das indústrias de desenvolvimento de *games* do Brasil. Por se tratar de uma área nova, é difícil encontrar profissionais capacitados nela. Uma das soluções para essa carência de mão-de-obra é incentivar estudantes, sejam eles de nível técnico ou universitário, a aprender sobre as ferramentas e técnicas mais utilizadas no desenvolvimento de um jogo eletrônico. Assim, torna-

se de suma importância a implementação de ferramentas que facilitem o primeiro contato do estudante com essa complexa área, questão essa que motivou a criação da *SAGA Game Library*.

1.2 Objetivos

O objetivo inicial deste trabalho de conclusão de curso consistia na implementação de um *game online* que utilizasse toda a base teórica vista durante a graduação. Entretanto após analise do conteúdo e recursos necessários para realizar tal projeto, concluiu-se que o mesmo demandaria um custo alto, tanto em relação a mão-de-obra quanto ao conhecimento necessário para fazê-lo. Com base nessa conclusão, seguimos em busca de outros temas de projeto ainda relacionados à área de jogos eletrônicos.

Após alguns estudos sobre o mercado de desenvolvimento de jogos, chegamos ao consenso de desenvolver uma biblioteca de desenvolvimento de jogos voltada para a atividade didática, com o nome do jogo que seria feito: SAGA. Considerando tanto a carência de profissionais na área dos *games* quanto a importância da profissionalização na tecnologia da informação em geral, o objetivo dessa *engine* é incentivar o entusiasta dos jogos na programação e o estudante de programação, seja por conta própria ou por meio de uma instituição de ensino, na programação de jogos.

Uma vez que a biblioteca é voltada para entusiastas e estudantes que possuem pouca ou mesmo nenhuma experiência na área de programação de jogos eletrônicos, torna-se essencial que ela seja de fácil uso, de modo que o usuário sinta-se a vontade para desenvolver seus projetos e não desestimulado por utilizar uma ferramenta muito complexa. As principais funcionalidades dessa biblioteca foram escolhidas em função da importância das mesmas na maioria dos jogos 2D: exibição de imagens, animação de *sprites*, tratamento de colisão, reprodução de áudio, tratamento de dispositivos de entrada (principalmente teclado e mouse) e exibição

de textos. Qualquer outra característica ou funcionalidade seria abandonada se fosse contrária à simplicidade do uso dessa biblioteca.

É certo que já existem muitas *game engines*, inclusive em C++, mas o estudo é o piso de todas as descobertas científicas, o que justifica e motiva o desenvolvimento de uma biblioteca de jogos didática. Esta é a nossa proposta: uma camada de orientação a objetos envolvendo a Allegro de uma forma simples e didática, para facilitar o aprendizado tanto da programação orientada a objetos quanto da programação de jogos. Essa *engine* oferece não a capacitação em si, mas um meio para ajudar o estudante ou profissional a alcançar essa capacitação.

2 Revisão Bibliográfica

2.1 Projeto do sistema

Uma vez definidos os requisitos do sistema, a próxima etapa era dar início ao desenvolvimento do projeto. O primeiro passo foi definir, com base nos requisitos e no conhecimento dos envolvidos, qual seria a linguagem de programação mais adequada ao problema e qual API deveria ser usada para as rotinas de renderização, acesso ao dispositivos de entrada, reprodução de áudio, etc.

2.1.1 Linguagem de Programação

Até a década de 90 cada jogo tinha a sua *engine*, feita para possibilitar a maior eficiência no uso da memória e da unidade de processamento possível, de acordo com as exigências de cada jogo. Um jogo que só usava formas geométricas, por exemplo, não precisava tratar imagens na sua *engine*. O nível da microeletrônica e da computação já possibilita o uso de *engines* genéricas, mas o desenvolvimento de uma ainda demanda uma programação muito próxima da máquina. É por isso que a escolha da linguagem de programação precisa ser feita com cuidado.

Considerando o conhecimento da equipe e o propósito do projeto, que era uma *engine* didática, para influenciar o desenvolvimento de jogos de acordo com o nosso alcance, as linguagens de programação selecionadas para a análise foram Java e C++, de acordo com a simplicidade, o poder e a portabilidade de cada uma.

Estudo Comparativo

Java é uma linguagem orientada a objetos desenvolvida pela Sun, hoje possuída pela Oracle. A sua fama de espaçosa e pesada não é coerente com a realidade: hoje a linguagem conta com a Compilação na Hora ou *Just in Time Compilation* (*JIT Compilation* ou só JIT), para que a sua execução não seja mais interpretada. Mas a sua principal característica é a portabilidade: a Máquina Virtual do Java ou Java Virtual Machine (JVM) é uma plataforma virtual compatível com a maioria das plataformas mais populares como o Microsoft Windows, Unix/Linux e MacOS X.

Por mais evoluída que seja a JVM, no entanto, o Java não admite o acesso à máquina necessário para o desenvolvimento de uma *game engine*, senão com o uso do C++, por meio da Interface Nativa do Java ou Java Native Interface (JNI). Em outras palavras, para usar o Java, nesse caso, nós teríamos que usar o C++. Esta, por sua vez, não é a mais simples na codificação, mas não tem limitação alguma tanto em termos de portabilidade, pois conta com um padrão oficial e compiladores para várias plataformas, quanto em termos de acesso à máquina.

C++ (C Mais Mais ou C Plus Plus) é uma linguagem de programação multi-paradigma, com suporte para a programação imperativa e a programação orientada a objetos, de uso geral, desenvolvida por Bjarne Stroustrup, para formar uma camada de orientação a objetos sobre a linguagem de programação C. O C++ possibilita a programação de baixo nível assim como a programação de alto nível e por isso é considerado uma linguagem de programação de nível médio em termos de proximidade da máquina [MIZRAHI 2006] .

Após o estudo comparativo, concluímos que a linguagem C++ seria a melhor opção para o projeto em questão. Tal decisão foi baseada em diversos fatores, como experiência e domínio da linguagem pelos envolvidos no projeto, o desempenho das aplicações escritas nesta linguagem e o fato de C++ ser a linguagem de programação mais usada no desenvolvimento de jogos. Uma vez que o objetivo da biblioteca é ser uma porta de entrada para essa área, a escolha da linguagem faz

com que o usuário já se familiarize com a mesma.

2.1.2 Allegro

Nas nossas pesquisas para escolher uma biblioteca com a qual trabalhar, duas se destacaram: a Allegro e a SDL. A SDL (*Simple Direct Media Layer* ou *Camada de Mídia Direta Simples*) é uma biblioteca multimídia simples de usar, multiplataforma, de código aberto, e amplamente usada para fazer jogos e aplicações multimídia [SDL 2014]. Ela também poderia atender às nossas necessidades, mas a Allegro se destacou por ter um código mais limpo e intuitivo, e rotinas específicas para o desenvolvimento de jogos, como renderização acelerada por hardware e suporte nativos a diversos formatos de imagens e arquivos de áudio. Por esta razão ela foi escolhida.

Allegro é uma biblioteca gráfica multiplataforma, de código fonte aberto e feita na sua maioria em C, mas utilizando internamente também Assembly e C++. Seu nome é um acrônimo recursivo que representa "*Allegro Low Level Game Routines*" ("Rotinas de jogo de baixo nível Allegro"). Funciona em diversos compiladores e possui rotinas para a manipulação de funções multimídia de um computador, além de oferecer um ambiente ideal para o desenvolvimento de jogos, tornando-se uma das mais populares ferramentas para esse fim atualmente. Originalmente desenvolvida por Shawn Hargreaves, ela se tornou um projeto colaborativo, com colaboradores de todo o mundo [ALLEGRODOC 2014].

Ela possui suporte nativo para rotinas de que utilizam gráficos 2D, embora seja possível utilizá-la em conjuntos com outras APIs, como OpenGL e DirectX, para desenvolvimento de aplicações que utilizem gráficos 3D. Apesar de não ser suficiente para o completo desenvolvimento de um jogo, existem pequenas bibliotecas adicionais (*add-ons*), feitas para serem acopladas à Allegro, permitindo assim a sua extensão. Através desses *add-ons* é possível, por exemplo, obter suporte a arquivos MP3, GIF, imagens JPG e vídeos AVI [ALLEGRODOC 2014].

Atualmente, a biblioteca se encontra na sua quinta versão. Allegro 5 foi completamente reescrita e não apresenta compatibilidade com as suas versões anteriores. Foi feito um esforço para tornar a API mais consistente e segura, o que trouxe melhorias funcionais e uma grande mudança na sua arquitetura, sendo agora orientada a eventos e possuindo suporte nativo a aceleração por hardware. Possui suporte a eventos gerados por dispositivos de entradas como teclado, *mouse* e *joystick* além de funções para desenho de primitivas gráficas, leitura e gravação de seu próprio tipo de arquivo de configuração (muito útil para armazenar configurações e dados de jogos) e é totalmente modular [ALLEGRODOC 2014].

A Allegro 5.0 suporta as seguintes plataformas:

- Windows (MSVC, MinGW);
- Unix/Linux;
- MacOS X;
- iPhone;
- Android (Suporte provido pela Allegro 5.1, que ainda se encontra instável).

A API atualmente se encontra nas versões 5.0.10 (estável) e 5.1.2 (instável). A *SAGA Game Library* foi desenvolvida usando como base a versão 5.0.10.

Características Técnicas

Uma vez que a Allegro é codificada na sua maioria em C, a principal característica dessa biblioteca é a estrutura imperativa. Embora algumas funções internas usem outras técnicas, a interface da Allegro é completamente imperativa, o que determina que a programação de qualquer jogo desenvolvido nessa biblioteca seja imperativa ou híbrida, a não ser que ela própria seja encapsulada.

Outra característica da Allegro, profundamente relacionada com a linguagem C, é o uso de ponteiros, o que torna a codificação complicada e consequentemente

propícia a erros, e a execução ligeiramente mais lenta. Esses detalhes exigem um cuidado especial no processo de encapsulamento da Allegro.

Principais Recursos e Funções

A seguir, encontramos um conjunto dos principais recursos da Allegro 5, começando com as funções mais gerais da biblioteca [ALLEGRODOC 2014].

- **al_init()**: Inicializa a biblioteca Allegro, dando valores a algumas variáveis globais e reservando memória. Deve ser a primeira função a ser chamada.
- **al_exit()**: Encerra a Allegro. Isto inclui retornar ao modo texto e remover qualquer rotina que tenha sido instalada. Não há necessidade de chamar essa função explicitamente, pois, normalmente, isto é feito quando o programa termina.

As rotinas de vídeo:

- **ALLEGRO_DISPLAY**: Tipo que representa a janela principal. A biblioteca permite que se trabalhe com múltiplas janelas.
- **al_create_display(width, height)**: Cria uma instância da janela, retornando um ponteiro para ALLEGRO_DISPLAY. Os parâmetros indicam as dimensões em pixels.
- **al_flip_display()**: Função para atualizar a tela.
- **al_destroy_display(var)**: Finaliza a instância *var* do tipo ALLEGRO_DISPLAY ×
.

As rotinas para manipulação de arquivos de imagem:

- **ALLEGRO_BITMAP**: Tipo que representa o arquivo de imagem carregado pela Allegro.

- **al_init_image_addon()**: Inicializa o add-on da Allegro 5 para utilização de imagens.

- **al_load_bitmap("example.jpg")**: Carrega a imagem indicando no parâmetro o nome e tipo. Ela deve estar previamente salva na pasta do programa. Recebe o caminho relativo ou absoluto da imagem a ser carregada, retornando um ponteiro para o tipo ALLEGRO_BITMAP.

- **al_draw_bitmap(bitmap, x, y, mirror)**: Função para desenhar a imagem na tela. Os parâmetros são o bitmap a ser desenhado, as posições x e y e as flags de espelhamento (0, ALLEGRO_FLIP_HORIZONTAL, ALLEGRO_FLIP_VERTICAL).

As rotinas de áudio:

- **ALLEGRO_SAMPLE**: Tipo que representa arquivos pequenos, geralmente efeitos sonoros.

- **ALLEGRO_AUDIO_STREAM**: Tipo para representar arquivos grandes, de forma que o arquivo não é carregado de uma vez para a memória. Geralmente representa os arquivos que irão compor uma trilha sonora.

- **al_install_audio()** e **al_init_acodec_addon()**: A primeira inicializa as funções relativas ao áudio. A segunda inicializa os codecs necessários para carregar os diversos formatos de arquivo suportados. Fornece suporte a alguns formatos, como Ogg, Flac e Wave.

- **al_set_audio_stream_playing(musica, true)**: Função que recebe o arquivo de áudio já carregado no primeiro parâmetro e um tipo booleano no segundo (true para fazê-la tocar ou false, em caso contrário).

- **al_destroy_audio_stream(musica)** e **al_destroy_sample(sample)**: Funções de desalocação dos arquivo de áudio carregados pela Allegro.

A Allegro ainda possui muitos recursos que não foram citados devido a sua quantidade. Posteriormente, a API será explorada mais a fundo conforme os recursos da *SAGA Game Library* forem mostrados.

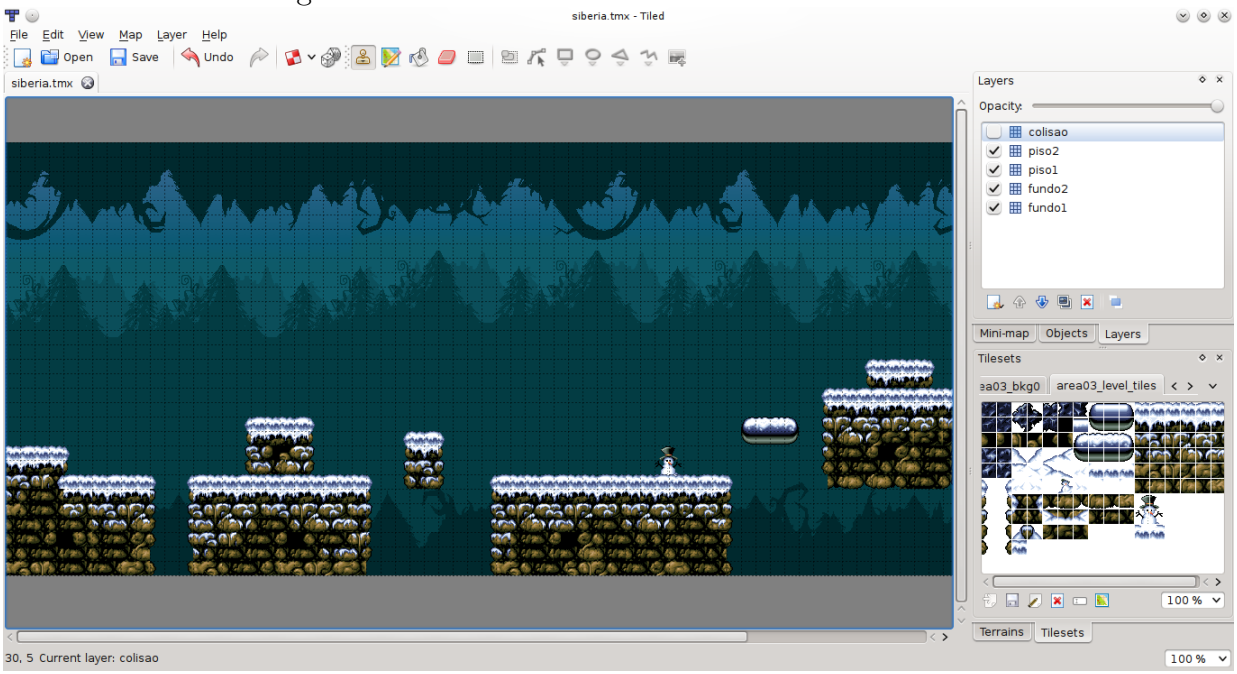
2.1.3 Suporte ao Tiled

A edição de níveis ou cenários é uma das tarefas mais complexas no desenvolvimento de um jogo. Todo o processo, desde a elaboração artística do cenário até a maneira como o jogador irá interagir com o mesmo, exige muito planejamento e demanda muito esforço em sua implementação. Com o objetivo de facilitar esse trabalho, surgiram os editores de níveis.

Os editores de níveis são *softwares* destinados à construção de cenários em jogos, o que faz deles ferramentas de grande importância para uma *game engine*. Neste ponto, existem duas opções disponíveis para quem deseja implementar uma *engine*: desenvolver seu próprio editor ou utilizar um editor externo, desenvolvido por terceiros. Desenvolver um editor próprio demanda muito estudo e tempo de desenvolvimento, além do que já é necessário para implementar a *engine*. Tendo isso em mente, chegou-se à conclusão de que o melhor seria oferecer suporte a um editor externo, no que o editor Tiled foi escolhido.

O software Tiled, cuja interface pode ser visualizada na Figura 2.1, é uma ferramenta gratuita desenvolvida em C++ para a criação de *layouts* e mapas usando *tilesets*, baseado na técnica de *Tilemap*. Ele suporta mapas com projeções ortogonais e isométricas e ainda permite que objetos personalizados sejam salvos como imagens na resolução que se desejar. Tem suporte também a comandos externos, *plugins* e formatos usados por outros editores [THORBJORN 2014].

Figura 2.1: Detalhes da interface do Tiled.



Fonte: *Printscreen* da aplicação no sistema operacional Linux Mint.

O Tiled faz a edição de várias camadas de *tiles* e salva tudo em um formato padronizado de extensão ".tmx". Uma das principais vantagens do formato TMX é sua organização, detalhamento e praticidade, sendo que seu conteúdo pode ser lido através do uso de um *parser* para arquivos XML. XML (*Extensible Markup Language* ou *Linguagem de Marcação Extensível*) é um formato de texto simples e flexível que permite a criação de documentos contendo dados organizados de maneira hierárquica [XML 2014].

A biblioteca TinyXML

É uma biblioteca escrita em C++, que analisa uma sequência de entrada no formato XML, permitindo o acesso aos dados contidos nesse último. Em outras palavras, ela realiza o *parsering* de uma arquivo .xml e armazena a informação em objetos C++ que podem ser manipulados livremente. Ela pode ser facilmente in-

tegrada em outros programas, bastando apenas adicionar seus arquivos ao projeto. Com ela é possível realizar o acesso aos dados direta ou iterativamente, a alteração da estrutura através de inserção e remoção de elementos, a remoção de espaços duplicados e a gravação para ficheiros em formato XML [RODRIGUES 2014].

TinyXML é uma estrutura extremamente compacta e robusta, elaborada para um rápido e fácil aprendizado. Pode ser usada para fins de código aberto ou comerciais. Ela é compatível com UTF-8, de modo a permitir que arquivos XML sejam manipulados em qualquer linguagem humana. Também apresenta grande simplicidade em seu uso e rapidez na leitura de dados [RODRIGUES 2014]. Devido a essas características, a TinyXML foi escolhida para realizar o *parsering* dos arquivos .tmx gerados pelo Tiles.

Tiled: Codificação e compactação de dados

Um dos recursos mais interessantes do Tiled é a possibilidade de exportar os dados contidos no arquivo .tmx de forma compactada e codificada. A principal vantagem do uso dos algoritmos compactação e codificação é a redução do tamanho em disco do arquivo .tmx resultante e o aumento da velocidade de carregamento do cenário, uma vez que a biblioteca carrega os dados codificados e/ou compactados e os decodifica/descompacta a nível de software, ao invés de realizar a leitura dos mesmo em disco.

A compactação de dados fornecida pelo Tiled é realizada pelas bibliotecas ZLIB [1] e GZIP [2], enquanto a codificação é realizada pelo algoritmo Base64 [3], também fornecido por uma biblioteca externa. São fornecidas as opções de exportar os dados no arquivo .tmx na forma codificada e compactada ou apenas na forma codificada. Também são oferecidas as opções de exportar os dados em formato puro (sem codificação) XML ou no formato CSV (*Comma Separated Value* ou

[1] ZLIB: http://www.zlib.net/
[2] GZIP: http://www.gnu.org/software/gzip/gzip.html
[3] Algoritmo Base64: http://www.adp-gmbh.ch/cpp/common/base64.html

Valores Separados por Vírgula), que consiste simplesmente em um arquivo de texto onde os dados são separados por vírgulas [4].

A *SAGA Game Library* provê suporte às 5 opções de exportação acima. Ela, assim como o Tiled, também faz uso das bibliotecas GZIP e ZLIB para descompactação e também realiza a decodificação dos dados em Base64 contidos no arquivo .tmx (detalhes de uso serão abordados posteriormente).

[4]CSV: http://creativyst.com/Doc/Articles/CSV/CSV01.htm#FileFormat

3 Desenvolvimento

A *SAGA Game Library* é uma *engine* orientada a objeto desenvolvida em C++ sobre a API Allegro 5. Ela possui suporte a diversos formatos de arquivos de imagem (PNG, JPEG e GIF, por exemplo), a criação de cenários usando o editor de níveis Tiled, suporte ao uso de fontes de texto TTF e a reprodução de diversos formatos de arquivos de áudio. Também possui gerenciamento automático de recursos (imagem, fontes e áudio), o que possibilita economia significativa de memória. A biblioteca fornece uma estrutura para implementação do *game loop* e suporte para eventos de teclado, *mouse* e *joystick* também estão presentes.

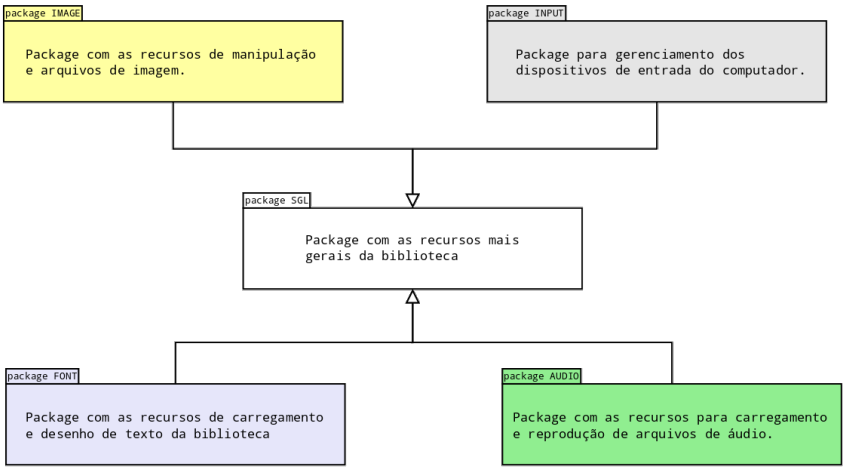
Após a fase de projeto, deu-se sequência à fase de implementação do mesmo, de modo a torná-lo orientado a objetos. Internamente, a *SAGA Game Library* consiste em um *package* denominado *sgl* e subdivididas em 4 outros *packages* com funções específicas: *image, font, audio e input.*

- image: consiste nas classes exclusivamente relacionadas à parte gráfica da *engine*, o que inclui manipulação de arquivos de imagens e carregamento e renderização de cenários do jogo;
- font: é o responsável pela parte textual da biblioteca. Ele carrega uma fonte no formato TTF ("True Type Font") padrão do Windows e a utiliza para escrever textos na tela de jogo;
- audio: *package* destinado ao carregamento, manuseio e reprodução dos arquivos de áudio;

- input: inclui todas as funções relacionadas aos comandos recebidos pelos dispositivos de entradas do computador (teclado, *mouse* e *joystick*).

Na Figura 3.1 é possível visualizar o esquema da organização da *engine*. A seguir, serão abordados os principais recursos da biblioteca bem como os detalhes da suas respectivas implementações.

Figura 3.1: Esquema estrutural da SGL.



Fonte: Elaborada pelos autores.

3.1 SGL

É o pacote mais geral e que engloba todos os outros. Quando a funcionalidade de uma classe não é específica ou é usada como ferramenta auxiliar em outras classes, ela é colocada nesse pacote.

3.1.1 AllegroStarter

Cada um dos módulos da Allegro deve ser inicializado antes de seu uso. A classe AllegroStarter é responsável por alocar e também por desalocar os recursos

da Allegro quando o programa é fechado. Uma exceção é lançada caso algum dispositivo apresente problemas durante a inicialização. Também contém informações sobre a atual versão da Allegro.

3.1.2 SGLException

É a classe que captura e trata as exceções que possam ocorrer durante a execução do programa, como inicialização dos componentes da Allegro e carregamento de arquivos de imagem, fonte e áudio. A classe é uma especialização de `std::exception`. O código a seguir exemplifica o uso desta classe.

```
1
2 if( !al_init() ) {
3   throw sgl::Exception( "Failed to initialize ALLEGRO_Lib." );
4 }
```

Caso ocorra algum erro de inicialização da Allegro, a função *al_init()* retornará *false*, fazendo com que o programa execute o conteúdo do *if* e lance a exceção. No console, teremos a mensagem: "*terminate called after throwing an instance of 'sgl::Exception' what(): Failed to initialize ALLEGRO_Lib.*".

3.1.3 Color

A classe Color possui recursos para se trabalhar com diversos formatos de especificação de cores, como RGB e formato hexadecimal utilizado em HTML. Os objetos dessa classe podem ser usadas, por exemplo, para colorir a tela ou alterar a cor de uma determinada fonte de texto. Ela aceita dois construtores. Com o primeiro deles é possível definir cores no formato RGB. Para isso, o construtor recebe três parâmetros que variam de 0 a 255, um para vermelho, outro para verde e outro para azul, respectivamente. O segundo construtor aceita *strings* no formato hexadecimal utilizado em HTML ou o nome em inglês de uma cor, desde que ele já esteja predefinido. Alguns dos nomes válidos são: *cyan*, *lightgreen*, *green*,

entre outros. Caso o nome de uma cor inexistente seja passado como parâmetro, o construtor cria um objeto com as coordenadas RGB iguais a 0 (cor preta). A seguir temos alguns exemplos de uso da classe Color.

```
1  // Definicao das cores vermelho e azul escuro usando o primeiro
       construtor.
2
3      Color vermelho(255,0,0);
4      Color azulEscuro(0,0,139);
5
6  // Definicao das cores rosa passando um nome pre-definido
7  // e das cores verde escuro e coral no formato hexadecimal
8  // utilizado em html.
9  // Note que o simbolo # e opcional.
10
11     Color rosa("pink");
12     Color verdeEsc("#006400");
13     Color coral("FF7F50");
14
15 // Convertendo formatos.
16
17   std::string cor;
18   cor = vermelho.getName(); // cor = red
19   cor = azulEsc.toHTML(); // cor = #00008b
```

3.1.4 Video

A classe Video é responsável por gerenciar todos os recursos de vídeo da *engine*. Através dela, tem-se acesso a todas as rotinas pertinentes (de atualização de tela, posicionamento, e outros eventos) para o gerenciamento de vídeo. A classe também possibilita a escolha por parte do usuário de criar um *display* no formato de janela

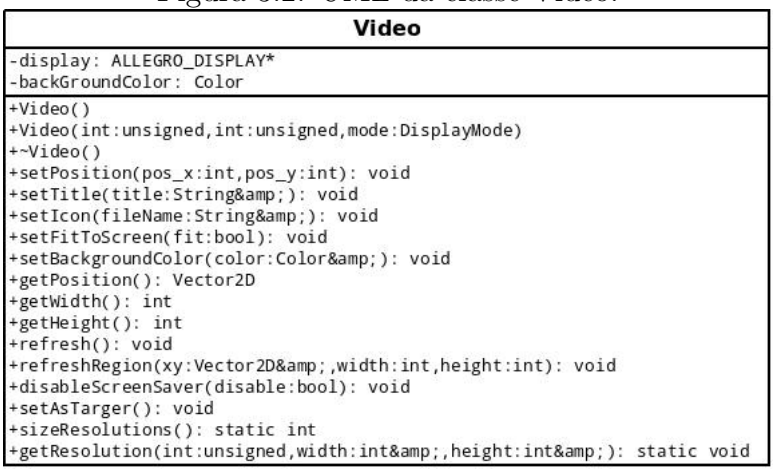
ou em tela cheia (*fullscreen*).

Quando criarmos um objeto da classe Video, devemos definir as dimensões do janela ou *display*, passando como parâmetro a largura e altura desejadas e o modo (WINDOWED, FULLSCREEN) no construtor da classe. Por padrão, a cor de fundo do vídeo é preta, podendo ser alterada. Também existe a possibilidade de adicionar um ícone e um título para a janela. A função *refresh()* é responsável por por transferir o conteúdo da memória de vídeo para o *display* e deve ser chamada sempre que seja necessário atualizar o *display*, caso contrário, nenhum das ações realizadas pelas rotinas de desenho serão visualizadas.

A principal dificuldade na implementação dessa classe foi aprender como a Allegro fazia uso da memória de vídeo e entender conceitos como o taxa de atualização do *display*. Outro fator preocupante diz respeito ao desempenho da renderização da *display*, que foi resolvido ajustando a Allegro para que ela armazenasse todas as imagens a serem desenhadas na memória de vídeo ao invés da memória RAM. Essa configuração apresentou resultados significativos no uso do processador uma vez que enquanto as imagens armazenadas na RAM utilizavam 25% da CPU, as mesmas imagens armazenadas na memória de vídeo não consumiam mais de 2% do processador durante o processo de desenho no *display*.

A Figura 3.2 mostra a classe Video em detalhes.

Figura 3.2: UML da classe Video.



Fonte: Elaborada pelos autores.

e a Figura 3.3, um exemplo de uso da mesma.

```
1
2  // Criamos uma instancia de video
3  Video video( 300, 100, DisplayMode::WINDOWED );
4
5  // Definimos o icone da janela do display
6  video.setIcon("sprite.jpg");
7
8  // Mudanca da cor de fundo da tela.
9  video.setBackgroundColor( Color("white") );
10
11 // Definindo um titulo para a janela.
12 video.setTitle( "Saga Game Library" );
13
14 // Atualizando o display.
15 video.refresh();
```

Figura 3.3: *Display* criado com o uso da classe Video.

Fonte: Elaborada pelos autores.

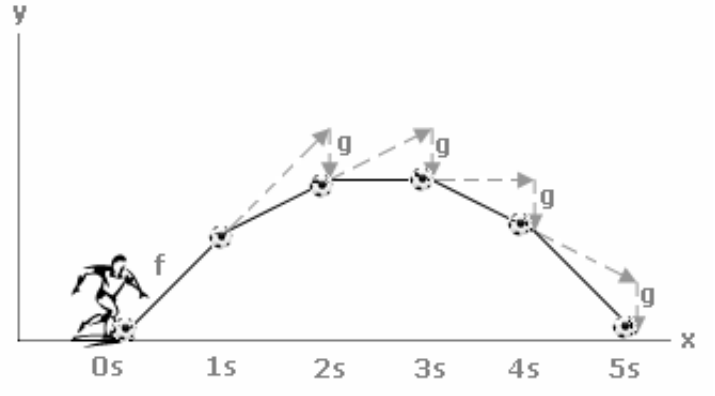
3.1.5 Vector2D

A classe Vector2D, tal qual o nome diz, define um vetor bidimensional, iniciando-se no ponto (0,0) da tela e indo até o ponto (x,y) definido pelo usuário, sendo responsável pelo posicionamento de qualquer entidade, seja ela um imagem ou um texto, no *display*. A escolha do uso de vetores ao invés de simples coordenadas deve-se a vantagens como simplicidade e padronização. Muitas operações matemáticas, como deslocamento, rotações e escalonamentos, são feitas utilizando vetores. Até mesmo efeitos de física, como gravidade e aceleração, são executados usando como base os conceitos da mecânica clássica. A vantagem do vetor sobre um par de coordenadas (x,y) deve-se à quantidade de informação que o primeiro carrega. Enquanto um ponto representa apenas uma localização no espaço, um vetor possui uma direção, sentido e magnitude.

Assim como ocorre na geometria, os vetores também possibilitam diversas

operações algébricas entre eles, através da sobrecarga de operadores. Soma e subtração de vetores e normalização são algumas das operações possíveis e que podem ser usadas para se obter interessantes resultados, como a simulação de gravidade, ilustrada na figura 3.4, onde temos um vetor representando o deslocamento do personagem e outro representando a força gravitacional.

Figura 3.4: Uso de vetores na simulação de gravidade.



Fonte: Imagem retirada de [GODOY 2014];

A seguir temos um exemplo com algumas das operações de Vector2D.

```
// Produto escalar entre a e b.

  Vector2D a(3,5);
  Vector2D b(3,9);

  float escalar = a.dotProduct(b); // escalar = 54

// Normalizacao de c.
  Vector2D c(4,3);
  c = c.normalize(); // c = ( 0.8, 0.6 )

// Soma entre vetores, com o operador + sobrecarregado.
```

```
13    c = a + b; // saida = ( 6,14 )
```

3.1.6 BoundingBox

Esta classe cria um retângulo usando dois objetos Vector2D. Ela possui rotinas para alterar a posição do retângulo e também verificar se houve colisão com outro retângulo. A BoundingBox é utilizada na biblioteca como um mecanismo para detectar se ocorreu colisão entre duas entidades, como por exemplo, um personagem e um objeto do cenário, como uma parede. Se ocorreu a colisão, o método responsável pela verificação retorna *true*, caso contrário ele retorna *false*. Vale ressaltar que a classe é responsável apenas por verificar se ocorreu ou não colisão, sendo de responsabilidade do usuário o tratamento adequado dessa colisão.

3.1.7 Geometrics

Geometrics é a classe usada para desenhar primitivas geométricas tendo como atributos um inteiro para armazenar a espessura da linha e duas variáveis da classe Color, uma para cor da linha e outra para cor do preenchimento. O construtor padrão inicializa a espessura com 1, a cor de preenchimento como branca e a cor da linha como preta. Outro construtor da ao usuário a liberdade de definir os valores como quiser. Através da classe Geometrics é possível desenhar linhas, triângulos, retângulos, retângulos com cantor arredondados, elipses, círculos e arcos.

Exemplos:

Declaração de vetores que serão usados:

```
1  Vector2D a(200,150);
2  Vector2D b(280,150);
3  Vector2D c(350,150);
4  Vector2D d(200,100);
```

No construtor da classe, dizemos que a espessura da linha é 5 e que sua cor é verde. O terceiro parâmetro diz respeito à cor de preenchimento da forma geométrica a ser desenhada.

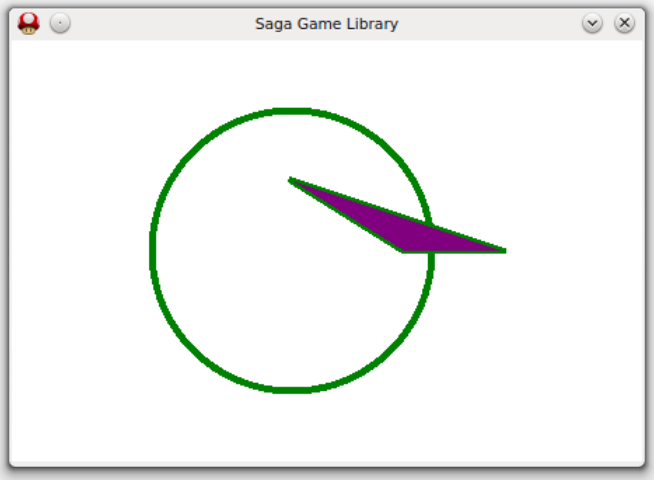
```
1  Geometrics geo(5, Color("green"), Color("purple"));
```

Em *drawCircle()*, o primeiro parâmetro deve ser a posição do círculo. O segundo é o raio e o terceiro é do tipo booleano. Se for *true*, o objeto será preenchido com a cor definida no construtor. Se for *false*, a forma não terá preenchimento algum. Note que parte do círculo não aparece porque está atrás do triângulo. Se o triângulo não tivesse cor de preenchimento, essa parte estaria visível.

```
1  geo.drawCircle(a, 100, false);
2  geo.drawTriangle(b, c, d, true);
```

O resultado pode ser visualizado na Figura 3.5.

Figura 3.5: Exemplo de uso da classe Geometrics.



Fonte: Elaborada pelos autores.

3.1.8 TimeHandler

TimeHandler é a classe que possui rotinas relacionadas à contagem de tempo, em milissegundos. A contagem de tempo é algo muito importante no desenvolvimento de *games*, pois ela pode ser usada para controlar a contagem de atualizações por segundo do *display* ou até mesmo executar ações do jogo que são dependentes do tempo. A Allegro possui suas próprias rotinas de contagem de tempo e a classe TimeHandler encapsula essas rotinas de modo a facilitar sua utilização, permitindo iniciar, pausar, retornar uma contagem de tempo e encerrá-la.

O trecho a seguir faz a contagem de tempo necessária para o computador percorrer a extensão de um tipo *unsigned int* em arquitetura 32 bits.

```
1  TimeHandler time;
2  time.start();
3  for (unsigned int i = 0; i<4294967295; i++){}
4  time.pause();
5  float t = time.getTicks();
```

A execução do código acima nos devolve a seguinte saída (em segundos):

```
1  t = 17.0142
```

3.1.9 Resource e ResourceManager

A classe Resource (Figura 3.6) representa um recurso (áudio, imagem ou fonte) a ser utilizada no jogo e é uma classe abstrata que serve de base para as classes ImageResource, usada para o carregamento de imagens, FontResource utilizada no carregamento de fontes *True Type*, AudioStreamResource e AudioSampleResource, que representam *samples* e *streams* de áudio. Quando um arquivo é carregado, seja ele fonte *True Type*, imagem ou áudio, o mesmo é armazenado em sua respectiva classe filha de Resource. Deste modo conseguimos separar de maneira clara cada tipo de recurso, sendo que cada classe filha possui métodos para alocação e desalo-

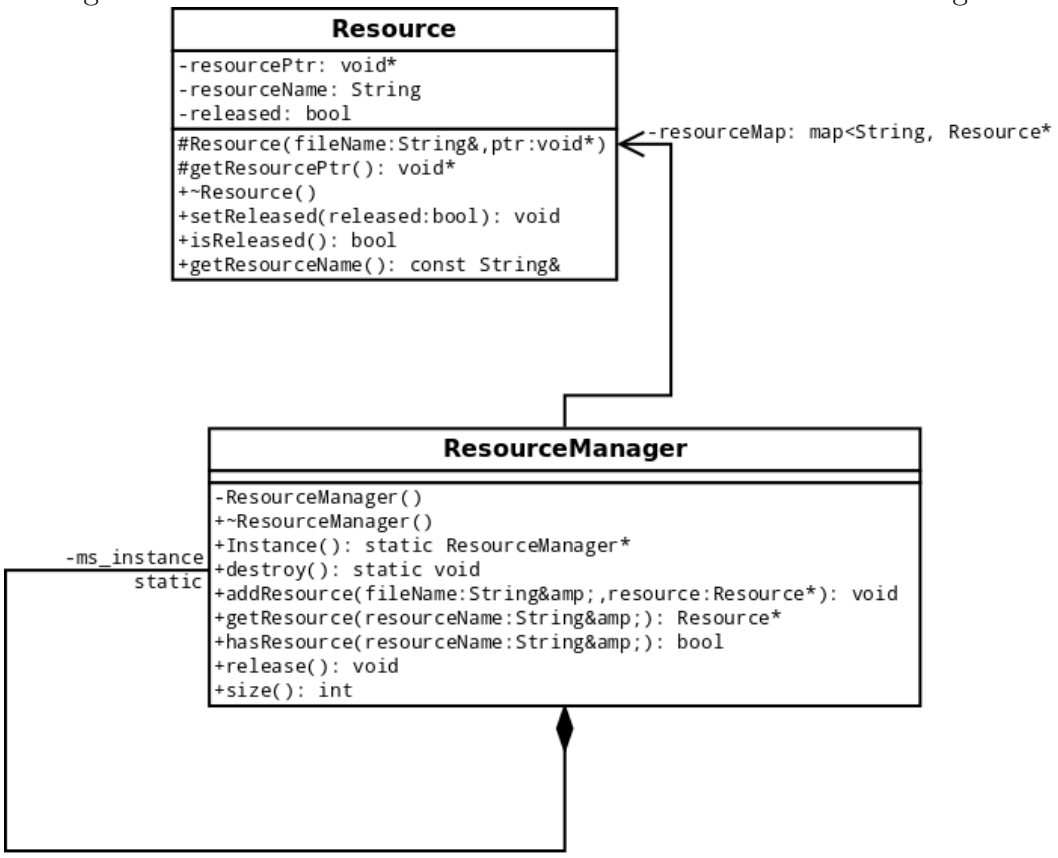
cação desse recursos pela Allegro além de métodos próprios para se trabalhar com o seu tipo de recurso.

Arquivos como imagens e áudio, podem ocupar tamanho razoável na memória (RAM e ROM) e apresentar um custo computacional considerável quando são carregados. No desenvolvimento de *softwares*, buscar o máximo em economia de memória e desempenho é um aspecto importante. Porém, quando o *software* em questão é um jogo eletrônico, esse aspecto passa a ser algo essencial. Jogos eletrônicos contam com rotinas para simulação de física, de inteligência artificial, de renderização e de carregamento de recursos são rotinas que demandam muito processamento. Além disso, tem-se os recursos que devem ser alocados, como arquivos de imagem e áudio, por exemplo. Com o objetivo de ter uma economia de memória e processamento, nota-se que o carregamento de um mesmo recurso várias vezes torna-se um inconveniente, conforme será explicado posteriormente. A classe ResourceManager foi criada para corrigir esse problema.

A ResourceManager (Figura 3.6) é a classe responsável por executar esse gerenciamento de recursos. Ela possui como principal atributo um objeto da classe *map* da biblioteca STL, chamado *resourceMap*, que usa como chave para armazenamento os *paths* dos recursos carregados.

Esta estrutura permite o gerenciamento dos recursos de forma otimizada, fazendo que haja economia de memória e processamento. É muito mais rápido para o programa apenas reproduzir um recurso já carregado do que ter de carregá-lo novamente. É muito importante ter esse tipo de gerenciamento, especialmente para jogos com imagens muito repetitivas. O algoritmo usado pelas classes que fazem uso do ResourceManager é o seguinte: procura-se pelo objeto (pertencente a uma das classes filhas de Resource), utilizando o nome do seu arquivo, no mapa de Resources na ResourceManager. Se o mesmo for encontrado, a ResourceManager retorna um ponteiro desse objeto, caso contrário criamos esse novo Resource e o inserimos no mapa de Resources. A seguir podemos verificar o uso desse algoritmo no método que faz o carregamento das imagens na biblioteca. O padrão *Single-*

Figura 3.6: UML detalhando as classes Resource e ResourceManager.



Fonte: Elaborada pelos autores.

ton foi utilizado devido a classe exercer uma papel de gerenciamento, ou seja, ter mais de uma classe gerenciando os recursos criados apenas tornaria mais custosa a utilização da mesma, uma vez que teríamos de procurar do resourceMap de cada ResourceManager pelo objeto Resource desejado.

```
1
2 ImageResource* ImageResource::loadImageResource( const String&
    fileName ) {
3
4   // Inciamos a string com a msg de carregamento
5   String str( "File " + fileName );
```

```
6   // Pegamos uma instancia do mapa
7   ResourceManager* rscMap = ResourceManager::Instance();
8   // Verificamos se o recurso ja foi carregado
9   ImageResource* rsc = static_cast<ImageResource*> ( rscMap->
        getResource( fileName ) ) ;
10
11  // Se for NULL devemos alocar um novo ImageResource e inserir no
        mapa
12  if( !rsc ) {
13
14    // Carregamos o bitmap
15    ALLEGRO_BITMAP* bitmap = al_load_bitmap( fileName.c_str() );
16
17    if( !bitmap ) {
18      throw sgl::Exception( "ERROR: Error to load ImageResource: " +
            fileName );
19      return nullptr;
20    }
21
22    // Criamos um novo recurso
23    rsc = new ImageResource( fileName, bitmap );
24    // Adicionamos o resource ao mapa
25    rscMap->addResource( fileName, rsc );
26    str += " loaded successfully!";
27  }
28  else {
29    str += " already exists!";
30  }
31
32  // Imprimimos o resultado da criacao da imagem
```

```
33   std::cout << str << std::endl;
34   return rsc;
35 }
```

Nesses casos a diferença entre ter ou não ter esse gerenciamento de recursos é grande e pode influenciar na performance do jogo, como podemos ver no exemplo a seguir.

De modo a verificar as vantagens do uso da ResourceManager, executamos o seguinte código e realizamos algumas medições:

```
1
2   // Vetor contendo a imagem para o nosso jogo
3   ALLEGRO_BITMAP* v_imagens[5000];
4
5   // Carregamos cada uma das ALLEGRO_BITMAP
6   for( int i=0; i < 5000; i++ )
7   {
8        v_imagens[i] = al_load_bitmap("sprite.png");
9   }
```

Esse código faz a Allegro carregar um conjunto de 5000 ALLEGRO_BITMAPs, todas com a mesma imagem. Isso consumiu cerca de 471 MB de memória e exigiu 25% do processador, levando 4,136 segundos para realizar carregamento das imagens.

O código a seguir executa o mesmo algoritmo de antes, porem realiza a alocação de 5000 objetos do tipo ImageResource (carregando o mesmo arquivo de imagem). A classe ImageResource é a classe filha de Resource responsável pelo carregamento e desalocação dos arquivos de imagens.

```
1
2   // Vetor contendo a imagem para o nosso jogo
3   ImageResource* v_imagens[5000];
```

```
4
5   // Carregamos cada uma das ImageResource
6   for( int i=0; i < 5000; i++ )
7   {
8       v_imagens[i] = ImageResource::loadImageResource("sprite.png");
9   }
```

Após a execução do algoritmo, constatamos que a memória ocupada foi aproximadamente 1,912 KB (0,39% da memória alocada anteriormente) e que a rotina consumiu apenas 2% de processamento, levando aproximadamente 0,0264 segundos (0,638% do tempo anterior) para realizar o carregamento. Com esse teste, ficam claras as vantagens de uso da *SAGA Game Library*, ao invés de utilizar somente a ALLEGRO. Esses valores podem ser visualizados na Tabela 3.1.

Tabela 3.1: Comparação de desempenho entre Allegro e SAGA Game Library.

Parâmetro Analisado	**Allegro**	**SAGA Game Library**
Memória utilizada	471 MB	1,912 KB
Uso do processador	25%	2%
Tempo de carregamento	4,136 seg.	0,0264 seg.

Fonte: Dados obtidos em simulações realizadas pelos alunos.

A maior dificuldade encontrada na implementação da ResourceManager foi implementar a desalocação dos *resources* criados. Inicialmente implementamos um contador de referências, de forma que, quando um objeto Resource não fosse mais utilizado por nenhum outro objeto, ele seria removido do mapa de *resources*. Essa abordagem mostrou-se pouco produtiva e confusa em sua implementação, por isso decidimos simplificá-la e deixar a responsabilidade de desalocar os Resource com a classe ResourceMananger ao final da execução do programa. O usuário pode remover diretamente o Resource do mapa de Resources da classe ResourceManager, mas caso ele não o faça, a própria classe se encarrega de desalocar a memória utilizada nos *resource* que ela possui ao final da execução do programa. Essa abor-

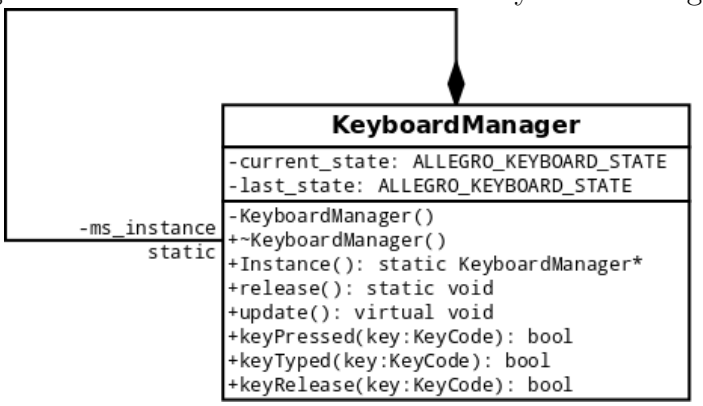
dagem oferece ao usuário um maior controle dos recursos alocados sem, no entanto, correr o risco de que memória alocada não seja liberada ao final da execução do programa.

3.2 SGL Input

Pacote que possui as classes responsáveis pelo gerenciamento das entradas de *mouse* e teclado. Esse pacote é constituído por duas classes: KeyboardManager e MouseManager. Ambas seguem o padrão *Singleton*, pois a Allegro não fornece suporte a mais de um teclado ou *mouse*.

3.2.1 KeyboardManager

Figura 3.7: UML detalhando a classe KeyboardManager.



Fonte: Elaborada pelos autores.

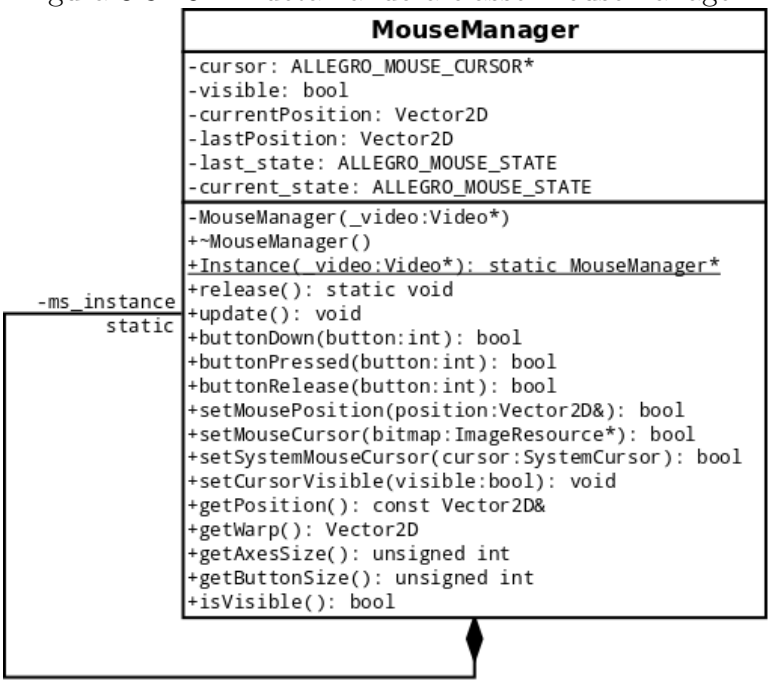
A classe KeyboardManager realiza o gerenciamento do teclado. A classe (Figura 3.7) possui duas variáveis do tipo ALLEGRO_KEYBOARD_STATE. Uma para guardar o último estado e outra para guardar o estado atual do teclado, de modo a conseguir verificar se determinada tecla foi pressionada, está sendo pressionada ou foi solta. Como o tipo ALLEGRO_KEYBOARD_STATE sugere, uma variável desse tipo armazena o estado (pressionada ou solta) de todas as teclas do

teclado no momento em que o método *update()* é chamado. Há outros três métodos que verificam se determinada tecla foi pressionada, continua pressionada ou se foi solta. Os métodos recebem como parâmetro o *keycode* da tecla e retornam um tipo booleano. A seguir temos um exemplo de uso da classe KeyboardManager.

```
1
2   // Inicializamos os dispositivos de entrada
3   KeyboardManager* keyboard = KeyboardManager::Instance();
4
5   // Atualiza o estado das teclas
6   keyboard->update();
7
8   // Verifica se a seta direita esta pressionada
9   if( keyboard->keyPressed( ALLEGRO_KEY_RIGHT ) ) {}
10
11  // Verifica se a tecla ESC foi solta
12  if( keyboard->keyRelease( ALLEGRO_KEY_ESCAPE ) ) {}
13
14  // Verifica se a tecla SPACE foi pressionda
15  if( keyboard->keyTyped( ALLEGRO_KEY_SPACE ) ) {}
```

3.2.2 MouseManager

Figura 3.8: UML detalhando a classe MouseManager.



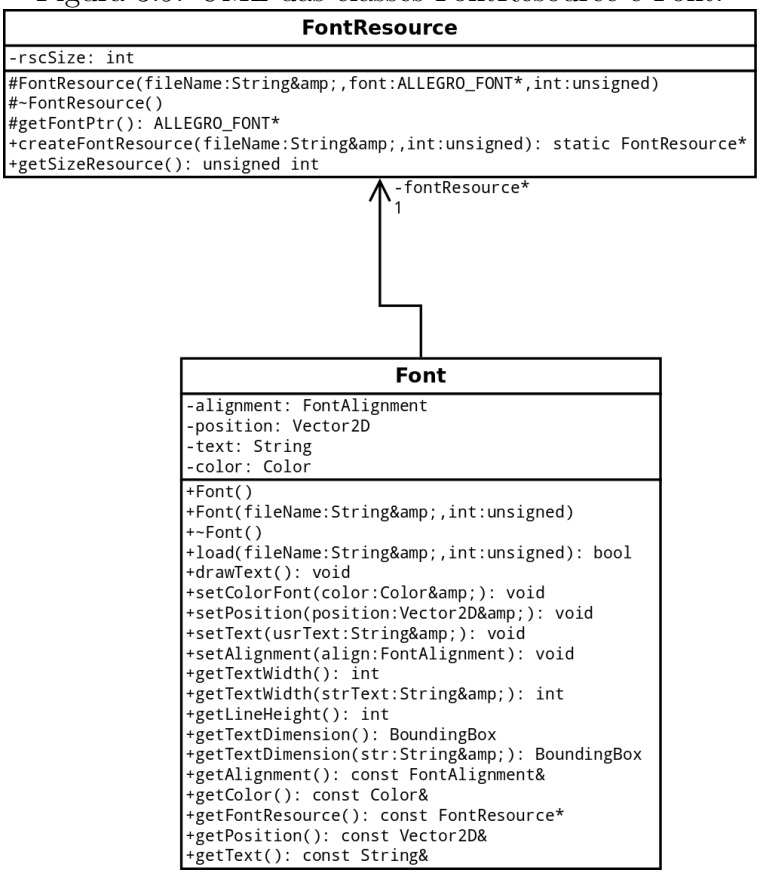
Fonte: Elaborada pelos autores.

A classe MouseManager realiza o gerenciamento dos eventos gerados pelo *mouse*. A classe (Figura 3.8) trabalha usando lógica semelhante a Keyboard-Manager, só que aplicada ao mouse. Ela Possibilita também deixar ou não visível o cursor do mouse na aplicação e alterar o estilo do cursor.

3.3 SGL Font

É o pacote responsável pela parte textual da biblioteca. Ele carrega uma fonte no formato TTF ("True Type Font") padrão do Windows e a utiliza para escrever um texto na tela.

Figura 3.9: UML das classes FontResource e Font.



Fonte: Elaborada pelos autores.

3.3.1 FontResource

A classe é uma especialização de Resource. Ela é usada dentro do método *load()* da classe Font e funciona da seguinte maneira: recebe um arquivo TTF e o tamanho da fonte desejada, e em seguida esse arquivo é adicionado a uma instância da classe ResourceManager, caso ainda não tenha sido alocado. É possível existir dois arquivos de fonte iguais mas com tamanhos diferentes em ResourceManager, isso possibilita que utilizemos fontes iguais, mas com tamanho diferentes, uma medida necessária, uma vez que a Allegro não permite que alteremos o tamanho da fonte posteriormente.

3.3.2 Font

A classe Font (Figura 3.9) possui métodos para escrever na tela e mudar as configurações do texto, tais como cor, alinhamento e posição. O pacote utiliza também as funcionalidades das classes Vector2D, Color e BoundingBox. Com os métodos *gets* temos informações sobre as configurações atuais do objeto.

A seguir temos alguns exemplos de uso da classe. Para instanciar um objeto da classe Font utilizamos seu construtor, cujo primeiro parâmetro é o *path* do arquivo TTF. A barra '/' indica hierarquia de diretórios a partir da pasta do projeto. Ou seja, dentro da pasta de projeto existe uma pasta chamada Resource e o arquivo alger.ttf se encontra nela. O segundo parâmetro é o tamanho da fonte.

```
1  // Criamos uma font
2  Font alger("Resource/alger.ttf", 30);
```

Alterando cor, alinhamento e posição. O tipo do alinhamento pode ser RIGHT, LEFT, CENTRE ou INTEGER.

```
1  // Definimos a cor da fonte
2  alger.setColorFont(Color ("lightskyblue"));
3  // Definimos seu alinhamento
4  alger.setAlignment(FontAlignment::CENTRE);
5  // Definimos sua posicao
6  alger.setPosition(Vector2D(200, 100));
```

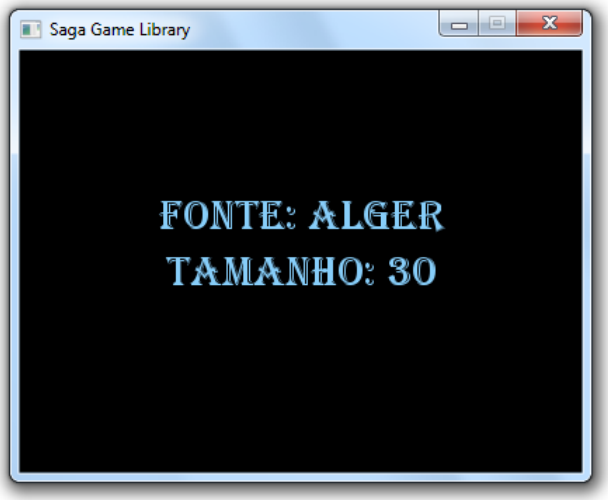
Definimos o texto que será exibido na tela e o escrevemos no *display*:

```
1  // Definimos o texto a ser escrito no display
2  alger.setText("Fonte: Alger");
3  // Desenhamos o texto
4  alger.drawText();
5  // Definimos um novo texto para ser escrito
6  alger.setText("Tamanho: 30");
```

```
7   // Ajustamos a posicao do segundo texto
8   alger.setPosition(Vector2D(200, 140));
9   // Desenhamos o segundo texto na display
10  alger.drawText();
11  // Atualizamos o display
12  video.refresh();
```

Após a execução do código acima, temos o seguinte texto no *display* (Figura 3.10).

Figura 3.10: Exemplo de uso da classe Font.



Fonte: Elaborada pelos autores.

3.4 SGL Áudio

É o pacote que realiza as funcionalidades referentes à parte sonora. É estruturado da seguinte maneira: de um lado a classe AudioResource, especialização de Resource, e da qual derivam AudioStreamResource e AudioSampleResource. Do outro lado, temos a classe abstrata Audio, da qual derivam AudioStream e AudioSample. As duas últimas relacionam-se com as respectivas especializações de AudioResource. Tal estrutura nos possibilita criar um objeto de AudioSample

quando desejamos manipular um efeito sonoro, como um soco ou o barulho de uma explosão, ou um objeto de AudioStream quando precisamos usar uma trilha sonora.

AudioSample e AudioStream possuem em comum métodos para carregar, tocar, alterar ganho, balanço, velocidade e modo de repetição. Além desses, cada classe possui outros métodos com funções específicas. Entre os vários formatos de arquivos de áudio suportados, os mais usados são .wav e .ogg. Infelizmente não existe suporte ao formato MP3, por o mesmo se tratar de um formato de áudio proprietário (e também pouco usado em jogos).

Exemplos:

Cria-se um objeto de AudioSample, indicando o caminho onde o arquivo de áudio encontra-se.

```
1  // Criamos um sample
2  AudioSample sample("Resource/audio/palmas.wav");
```

Altera o ganho para 80% do valor original e a velocidade para 90% do valor original.

```
1  // Ajustamos o valor do ganho
2  sample.setGain(0.8);
3  // Ajustamos a velocidade de execucao do sample
4  sample.setSpeed(0.9);
```

Faz com que o arquivo continue tocando indefinidamente após chamar o método *play()*.

```
1  // Ajustamos para reproducao em loop
2  sample.setLoopingMode(AudioPlayMode::PLAY_LOOP);
```

Finalmente, colocamos o arquivo para ser reproduzido.

```
1  // Iniciamos a reproducao do sample
```

```
2   sample.play();
```

O código a seguir carrega um arquivo para ser reproduzido como trilha sonora e recebe parâmetros para alterar o *buffer* (em *bytes*) e a taxa de amostragem.

```
1   // Carregamos um audiostream
2   AudioStream stream("Resource/audio/interface.ogg", 4, 1024);
```

Inciamos a reprodução do o arquivo a partir dos 14 segundos.

```
1   // Definimos o tempo de inicio
2   stream.setBegin(14);
3   // Iniciamos a reproducao do audio stream
4   stream.play();
```

3.5 SGL Image

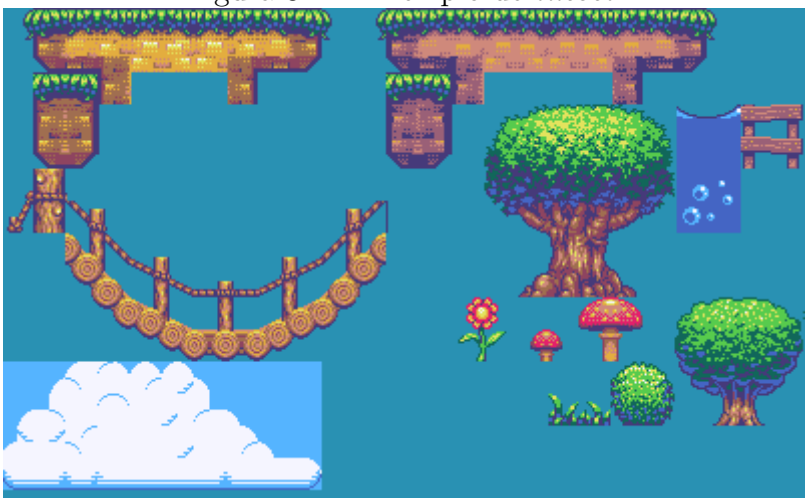
Trata-se de um dos principais pacotes da *SAGA Game Library*, contendo todas as classe referentes à manipulação de recursos de imagem. Apesar do pacote possuir um grande volume de classes, nos limitaremos às suas duas classes mais importantes: AnimatedSprite e TMXTileMap.

3.5.1 TMXTileMap

A grande maioria dos jogos eletrônicos 2D apresenta, além das imagens, também chamadas de *sprites*, dos personagens e itens, uma imagem representando o cenário do jogo. Dependendo da natureza do jogo, esses cenários podem possuir grandes dimensões, o que torna custoso em termos computacionais armazenar essa imagem em memória (RAM e disco) e desenhá-la no *display*. Entretanto, quando analisamos a imagem que representa o cenário, podemos verificar que o mesmo é formado por pequenas partes que se repetem com muita frequência. A definição formal do termo *sprite* será mostrado na subseção sobre a classe AnimatedSprite.

Assim, aproveitando dessa característica, com o objetivo de diminuir o consumo de memória e o desempenho na renderização e carregamento das imagens de resolução elevada, foi desenvolvida uma técnica conhecida como *TileMap*, que consiste no uso de uma imagem, chamada *tileset* (Figura 3.11), contendo pequenos pedaços de imagens, conhecidos como *tiles*, que são as imagens que se repetem em grande quantidade no cenário do jogo. Esses *tiles* são usados para criar uma imagem composta denominada *tiled layer* [NOVAK 2010].

Figura 3.11: Exemplo de *tileset*.



Fonte: *Generic tileset* de autoria de Surt(OpenGameArt).

O cenário final do jogo (Figura 3.12) pode ser constituído de um único *tiled layer* ou ser resultante da combinação de dois ou mais *tiled layers*. Através da técnica de *Tilemap*, torna-se possível construir inúmeros cenários, com variadas dimensões, usando como base o mesmo *tilesets*, aumentando a economia de memória e não reduzindo o desempenho no carregamento de imagens.

Figura 3.12: Exemplo de um cenário construído com *tileset*.

Fonte: Elaborada pelos autores.

A *SAGA Game Library* também faz uso da técnica para criação de cenários e *sprites* animados através do *Tiled*. O Tiled nos fornece um arquivo XML personalizado chamado TMX, contendo todos os dados relevantes para a construção do cenário do jogo, e para ter acesso a esses dados torna-se necessário o uso de um *parser* XML, para realizar a leitura do arquivo XML. O *parser* escolhido foi a TinyXML, por sua baixa curva de aprendizado e velocidade de leitura. A seguir podemos visualizar um exemplo simples de um arquivo .tmx.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <map version="1.0" orientation="orthogonal" width="25" height="16"
     tilewidth="32" tileheight="32">
3  <tileset firstgid="1" name="030_collision" tilewidth="32"
      tileheight="32">
4   <image source="030_collision.png" width="64" height="32"/>
5  </tileset>
```

```
6   <layer name="nuvem" width="25" height="16">
7    <data encoding="base64" compression="zlib">
8     eJy7xMDAcBmIrzDgBheA+CIQXyJCLTZwD4jvA/
          EDPGruAPFdItVS6h5iACnuGQXEAWzxdpXKdmCLt4dkuo3aYDRNjYLhBgDMpiiN

9    </data>
10  </layer>
11 </map>
```

Como pode ser observado, o arquivo .tmx possui todos os detalhes de cada *tileset* usado na construção do cenário e os dados referentes a cada *layer* criado. Uma das dificuldades de implementação do suporte ao Tiled foi desenvolver as rotinas de leitura desses dados, de modo a torná-las rápidas e de fácil uso. O resultado desse trabalho foi o desenvolvimento de classes responsáveis pela leitura de cada elemento do arquivo, de forma que cada um desses dados fosse armazenados em atributos para posteriormente serem usados pelo usuário. Assim foram implementadas as classe TMXTileSet e TMXLayer entre outras. Cada uma dessas classes é responsável por realizar o *parsering* dos seus respectivos elementos e atributos (do arquivo .tmx). Por exemplo, a classe TMXLayer é responsável pela leitura dos elementos *layer* e de seus respectivos atributos, *name, width* e *height* e também é responsável pela leitura, compressão (GZIP ou ZLIB) e decodificação (base64) usadas no elemento *data*.

Quando observa-se o elemento *data* do arquivo .tmx, verifica-se que ele possui dois atributos: *encoding* e *compression*. Esses atributos nos indicam se foi usada algum tipo de codificação (base64) e se algum algoritmo de compressão (GZIP ou ZLIB) foi aplicado. Para a descompressão foi utilizada a biblioteca ZLIB, que fornece métodos para se trabalhar com compressão usando GZIP e ZLIB, e para a decodificação foi usada o algoritmo desenvolvido por René Nyffenegger [1], que fornece um modo muito prático de decodificação. Ambos os algoritmos,

[1] http://www.adp-gmbh.ch/cpp/common/base64.html

de compressão e descompressão funcionam da mesma maneira, é passada uma *string* contendo os dados codificados/comprimidos e como retorno temos a *string* decodificada/descomprimida.

Através dessa *string*, nós criamos um vetor de inteiros (através da conversão de tipos) que contem os índices de cada *tile* do *tileset*. Feito isso, podemos desenhá-los, fazendo com que o nosso cenário criado seja visualizado no *display*. Finalmente, a classe TMXTileMap contem os métodos para carregar e manipular os dados obtidos pela leitura do arquivo tmx, também provendo rotinas de verificação de colisão entre *sprites*.

Um cenário completo construído usando o *software* pode ser visualizado na Figura 3.13 e na Tabela 3.2 é possível verificar a redução do tamanho em disco de um arquivo .tmx e do tempo de carregamento desse em relação ao tipo de codificação/compressão utilizada (quando utilizada). Através desses dados pode-se observar que o uso de codificação/compressão trouxe redução no tempo de carregamento do cenário.

Tabela 3.2: Comparação de tamanho em disco e tempo de carregamento do arquivo .tmx compactado/codificado.

Codificação/Compressão	**Tamanho em disco**	**Tempo de carregamento**
Apenas XML	31,3 KB	0,10129 seg
Apenas Base 64	9,6 KB	0,07337 seg
Base64 + ZLIB	2,1 KB	0,07102 seg
Base64 + GZIP	2,2 KB	0,07122 seg
CSV	4,9 KB	0,07661 seg

Fonte: Dados obtidos em simulações realizadas pelos alunos.

Como pode-se observar, o Tiled fornece grande comodidade no projeto de cenários. Pensando nisso, também foi adaptado o uso do Tiled para criarmos *sprites* animados. As classes envolvidas são as mesmas usadas para se trabalhar com os cenários (com exceção da classe TMXTileMap).

Figura 3.13: Cenário construído usando o Tiled e a classe TMXTileMap.

Fonte: Elaborada pelos autores.

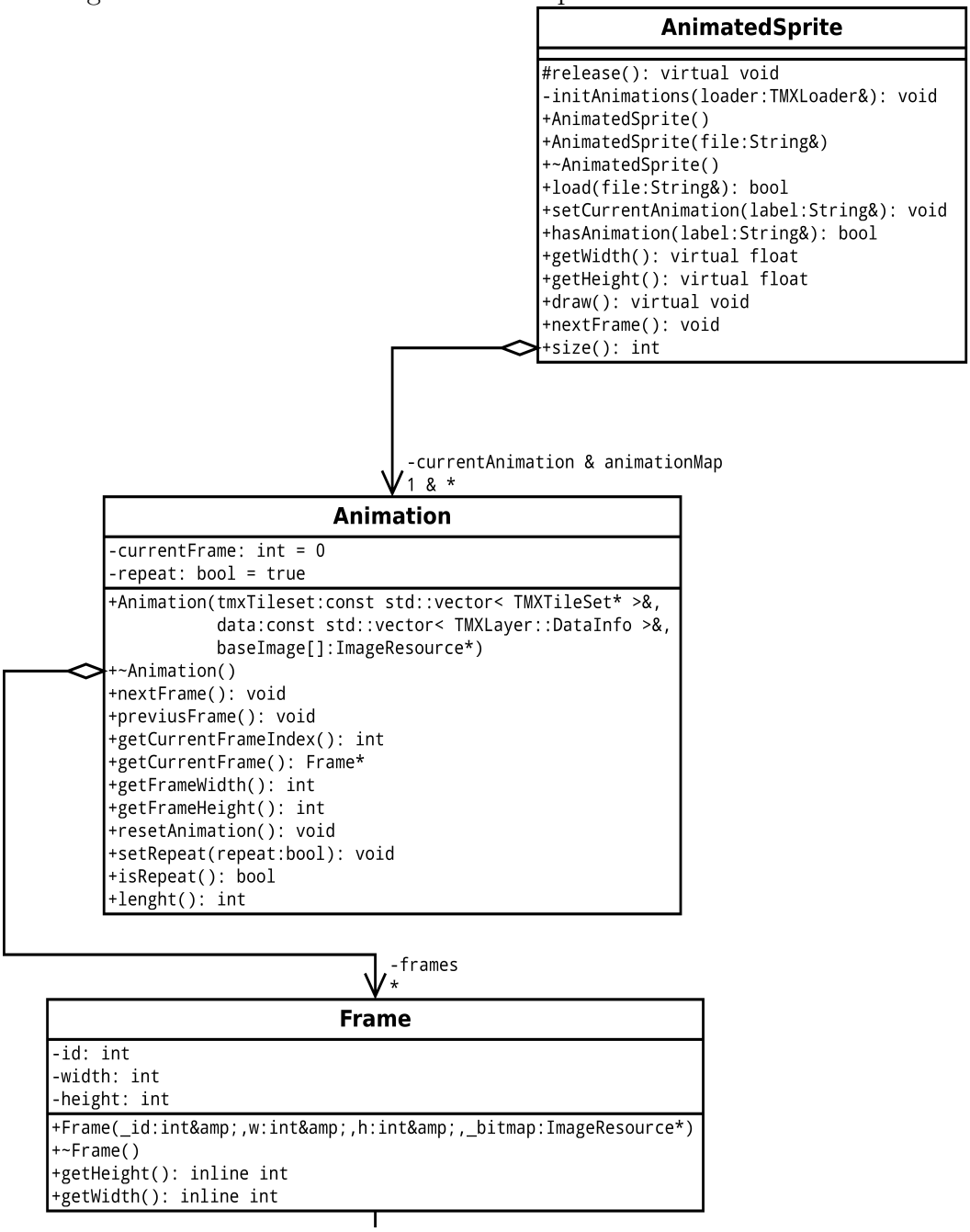
3.5.2 AnimatedSprite

Sprites são estruturas que permitem a criação de animações e permitem livre posicionamento na tela. De um modo geral, todas as imagens de um jogo podem ser denominadas *sprites*. Um *sprite* sempre possui um quadro fixo definido e pode ter vários conjuntos de animações configurados [PERUCIA 2007]. Os *sprites* que possuem apenas uma quadro, ou seja, que não possuem animação são representados pela classe StaticSprite. Esta classe recebe um objeto ImageResource e o manipula, definindo sua posição no *display*, realizando a renderização e verificando se ocorreu colisão desse com outro *sprite*. Quanto à classe AnimatedSprite, ela apresenta funções muito mais complexas e interessantes, as quais serão abordadas a seguir.

A classe AnimatedSprite (Figura 3.14) é usada quando se deseja ter um *sprite* com suporte a animações. Essas animações são criadas através do Tiled, e car-

regadas posteriormente pelo método *load()* da classe. Um objeto da classe AnimatedSprite é constituído de um vetor de objetos do Animation, que por sua vez possui uma coleção de objetos da classe Frame. Essas classes, como seus respectivos nomes sugerem, são responsáveis por armazenar os dados de cada animação criada para o *sprite* animado.

Figura 3.14: UML da classe AnimatedSprite e de suas classes auxiliares.



Fonte: Elaborada pelos autores.

A criação das animações faz uso de um *tileset* chamado *spritesheet* (Figura

3.15) que contem todas as imagens referentes às animações que o *sprite* pode possuir. Cada animação possui um número de *frames*, que são os quadros com o desenho do personagem em uma determinada posição [GEDIGames 2010].

Figura 3.15: Exemplo de *spritesheet*.

Fonte: *Spritesheet* de autoria de [SITHJESTER 2014].

As animações são identificadas por *labels* que são definidos durante a criação das animações com o Tiled e posteriormente podem ser acessadas com o método *setCurrentAnimation()* da classe AnimatedSprite. O uso da AnimatedSprite proporciona grandes vantagens ao desenvolvedor ao oferecer suporte ao Tiled, tornando a criação de animações uma tarefa visual e mais atraente, fazendo uso do gerenciado de recursos ResourceManager para carregamento dos *spritesheets* e possibilitando uso prático das animações. Vale ressaltar que a Allegro não possui suporte nativo à criação de *sprites* animados, deixando o desenvolvedor responsável por implementar esse recurso. Felizmente, esse detalhe é contornado com o uso da AnimatedSprite, deixando o desenvolvedor livre para se focar em outras áreas mais importantes de seu projeto.

Para criarmos um *sprite* animado usando a somente a Allegro, começamos definindo a estrutura de cada *frame*.

```
1  /* Constantes do sprite */
2  const int MAX_FRAMES = 12;
3  const int NUM_COLUNAS = 5;
4  const int NUM_LINHAS = 3;
5
6  /* Definimos as propriedades do sprite */
7  typedef struct Sprite
8  {
9      // Recebe o bitmap do sprite
10     ALLEGRO_BITMAP* vetorSprites[ MAX_FRAMES ];
11     // Guarda posicao dos sprites
12     float pos_x, pos_y;
13 } Sprite;
```

Em seguida criamos um *sprite* que possui apenas uma animação. Essa animação funciona em *loop*, ou seja, quando o último *frame* da animação for desenhado, a animação volta a se repetir desde o primeiro *frame*.

```
1    /* Declaramos a imagem de fundo e o sprite sheet */
2    ALLEGRO_BITMAP* sprite_sheet = al_load_bitmap( "lutador.png" );
3    /* Criando o personagem */
4    Sprite personagem;
5    /* Recebe as dimensoes do sprite */
6    int frame_w = al_get_bitmap_width (sprite_sheet)/NUM_COLUNAS;
7    int frame_h = al_get_bitmap_height(sprite_sheet)/NUM_LINHAS;
8    /* Ajustamos as posicoes do personagem */
9    personagem.pos_x = 0;
10   personagem.pos_y = 0;
11   /* Variavel auxiliar para receber a posicao de cada frame dentro
         do sprite sheet */
12   int frame_x, frame_y;
```

```
13
14      /* Incializamos o vetor do animacoes com os bitmaps */
15      for( int i = 0; i < MAX_FRAMES; i++ ){
16            frame_x = ( i % NUM_COLUNAS ) * frame_w;
17            frame_y = ( int ) ( i / NUM_COLUNAS ) * frame_h;
18
19            personagem.vetorSprites[ i ] = al_create_sub_bitmap(
                  sprite_sheet, frame_x, frame_y, frame_w, frame_h );
20      }
21      /* Indica qual o frame atual do sprite */
22      int frame_atual = 0;
23
24      while( true )
25      {
26            /* Desenhamos o sprite na tela */
27            al_draw_bitmap( personagem.vetorSprites[ frame_atual ],
                  personagem.pos_x , personagem.pos_y, 0 );
28            /* Atualiza a tela */
29            al_flip_display();
30            al_clear_to_color( al_map_rgb( 0, 0, 0 ) );
31            /* Incrementamos o frame e calculamos o novo valor. Assim,
                   sempre teremos valores entre 0 e MAX_FRAMES-1*/
32            frame_atual++;
33            /* Se for o ultimo frame, voltamos ao primeiro */
34            frame_atual = frame_atual % MAX_FRAMES;
35            /* Aguarda alguns milesegundos */
36            al_rest( 0.06 );
37      }
```

A seguir, temos a criação de um *sprite* animado usando a classe AnimatedS-

prite, considerando que já temos o arquivo .tmx com todas as animações prontas.

```
1  // Declaramos o sprite
2  AnimatedSprite spr;
3  // Carregamos o arquivo tmx
4  spr.load( "personagem.tmx" );
5  // Ajustamos posicao do sprite
6  spr.setPosition( Vector2D( 100.0f, 100.0f ) );
7  // Colocamos o sprite como visible
8  spr.setVisible( true );
9
10 while( true )
11 {
12     // Desenhamos o sprite
13     spr.draw();
14     // Avancamos para o proximo frame
15     spr.nextFrame();
16     // Atualizamos o display
17     video.refresh();
18     // Aguarda alguns milesegundos
19     al_rest( 0.06 );
20 }
```

Com o código acima, fica evidente a vantagem de usar a AnimatedSprite para criar *sprites* animados ao invés de utilizar a apenas a Allegro.

3.6 Testes

A fase de testes consistiu simplesmente no uso das classes implementadas buscando erros de execução e *bugs*. Não foi utilizada nenhuma ferramenta específica para testes de *software*, devido à pouca experiência da equipe com essas ferra-

mentas, ao tempo disponível para terminar essa versão do projeto e também à constatação das vantagens da *SAGA Game Library*, não só na performance, mas também na usabilidade. Muitos dos resultados dos testes foram descritos nesse relatório, como as medições de consumo de memória e o tempo de carregamento da classe ResourceManager.

4 Conclusão

Através deste projeto, nós concluímos que o desenvolvimento de jogos eletrônicos, em toda a sua extensão, envolve várias habilidades importantes para qualquer profissão da área da tecnologia, como a criatividade e a lógica matemática, especialmente para a tecnologia da informação e a engenharia da computação, como as técnicas de programação e a engenharia de software, o que confirma a viabilidade do uso dessa atividade de uma forma didática.

Ainda mais considerando a tendência da inclusão de disciplinas de programação na grade do ensino fundamental , em qual contexto essa atividade seria um grande incentivo, senão no mínimo um meio de divulgação da programação, que por sua vez colaboraria com a educação de uma forma geral. Por mais utópica que pareça, essa expectativa está de acordo com a nossa realidade – o Brasil é o quarto maior consumidor de games do mundo .

O projeto nos proporcionou um rico aprendizado sobre a área de desenvolvimento de jogos eletrônicos, bem como das técnicas utilizadas no desenvolvimento e na programação dos mesmos. Por meio de pesquisa e dedicação, conseguimos implementar um *software* que foi ao encontro de todos os requisitos estipulados no inicio do projeto, ou seja, uma *game engine* simples de usar, até mesmo para iniciantes na programação, incluindo características como portabilidade do código e uma ferramenta para facilitar o gerenciamento dos recursos dos jogos, que somam opções e qualidades sem no entanto diminuir a usabilidade da mesma. Usabilidade pode ser averiguada quando se observa os exemplos de uso da *SAGA Game Library*

disponíveis neste trabalho.

Uma vez que a *SAGA Game Library* esteja finalizada e totalmente testada, pretendemos torná-la um projeto de *software* livre, de modo que tanto estudantes e escolas quanto entusiastas possam utilizá-la para projetos pessoais, educacionais ou até mesmo comerciais.

Referências Bibliográficas

ALLEGRODOC. *ALLEGRO - a game programming library docs*. jun. 2014. Disponível em: <http://alleg.sourceforge.net/docs.html>.

GEDIGAMES. Relatorio final - mapeamento da industria brasileira e global de jogos digitais. Nucleo de Politica e Gestao Tecnologica - PGT, USP, 2010.

GODOY, V. *O Uso de Vetores em Jogos*. jun. 2014. Disponível em: <http://www.pontov.com.br/site/arquitetura/54-matematica-e-fisica/132-o-uso-de-vetores-nos-jogos>.

LARA, G. *Jornal da Tarde - quarto maior consumidor mundial de games*. jun. 2014. Disponível em: <http://blogs.estadao.com.br/jt-seu-bolso/brasil-e-4o-maior-consumidor-mundial-de-games/>.

MIZRAHI, V. V. *Treinamento em Linguagem C++ - Modulo 2*. [S.l.]: Pearson Prentice Hall, 2006.

NOVAK, J. *Game Development Essencials*. [S.l.]: CENGAGE Learning, 2010.

PEREIRA, L. *OLHAR DIGITAL*. jun. 2014. Disponível em: <http://olhardigital.uol.com.br/noticia/escolas-defendem-ensino-de-programacao-a-criancas-e-adolescentes/35075>.

PERUCIA, A. S. *Desenvolvimento de Jogos Eletronicos - Teoria e Pratica*. [S.l.]: Novatec Editora., 2007.

RODRIGUES, R. Notas sobre tinyxml. jun. 2014. Disponível em: <http://paginas.fe.up.pt/ aas/pub/Aulas/LAS/Ano10_11/Slides/TinyXML.pdf>.

SDL. *SDL - Simple DirectMedia Layer*. jun. 2014. Disponível em: <http://wiki.libsdl.org/FrontPage>.

SITHJESTER. *Sithjester RMXP Resources*. jun. 2014. Disponível em: <http://untamed.wild-refuge.net/rmxpresources.php?characters>.

THORBJORN, L. *TILED: Editor de mapas*. jun. 2014. Disponível em: <http://www.mapeditor.org/>.

XML. *Extensible Markup Language (XML)*. jun. 2014. Disponível em: <http://www.w3.org/TR/REC-xml/#sec-intro>.