## Bibliotecas de Jogos

Uma biblioteca de jogos ou game engine pode ser vista como uma caixa 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 pregos, martelos, parafusos, chaves de fenda e serras, mas as vantagens que as 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 precisa proporcionar, no mínimo, ferramentas para manipular sons, imagens (texto, imagens, etc), memória (dados) e controle (teclado, mouse, etc).

## Saga Game Library

Há duas finalidades que justificam o desenvolvimento de uma tecnologia que já existe: 1. O estudo da tecnologia, que por sua vez possibilita 2. O aperfeiçoamento da tecnologia. 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.

A engine está estruturada em 6 pacotes. São eles:

**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.

* 1. **-**  **Sgl:** arquivo base que contém todos os includes da Allegro que serão usados e as definições dos outros pacotes. Além disso, contém o tipo String, como definido abaixo.

typedef std::string String;

* 1. **– AllegroStarter:** classe responsável por inicializar a Allegro e seus componentes, e também por desalocar os recursos 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. É baseada no padrão Singleton, e portanto não é possível instanciá-la manualmente.
  2. **– Sgl\_exception:** é a classe que captura e trata as exceções que possam ocorrer durante a execução do programa. É uma especialização de std::exception. Exemplo:

if( !al\_init() ) {

throw sgl::Exception( "Failed to initialize ALLEGRO\_Lib." );

}

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 exceção. No console, teremos a mensagem: "terminate called after throwing an instance of 'sgl::Exception'

what(): Failed to initialize ALLEGRO\_Lib."

* 1. **– Color:** possui recursos de cores, que podem ser usados, 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 html ou o nome em inglês de uma cor, desde que ele já esteja pré-definido. Strings mal formatadas podem levar a resultados inesperados. Nomes válidos são:

aliceblue, antiquewhite, aqua, aquamarine, azure, beige, bisque, black, blanchedalmond, blue, blueviolet, brown, burlywood, cadetblue, chartreuse, chocolate, coral, cornflowerblue, cornsilk, crimson, cyan, darkblue, darkcyan, darkgoldenrod, darkgray, darkgreen, darkkhaki, darkmagenta, darkolivegreen, darkorange, darkorchid, darkred, darksalmon, darkseagreen, darkslateblue, darkslategray, darkturquoise, darkviolet, deeppink, deepskyblue, dimgray, dodgerblue, firebrick, floralwhite, forestgreen, fuchsia, gainsboro, ghostwhite, goldenrod, gold, gray, green, greenyellow, honeydew, hotpink, indianred, indigo, ivory, khaki, lavenderblush, lavender, lawngreen, lemonchiffon, lightblue, lightcoral, lightcyan, lightgoldenrodyellow, lightgreen, lightgrey, lightpink, lightsalmon, lightseagreen, lightskyblue, lightslategray, lightsteelblue, lightyellow, lime, limegreen, linen, magenta, maroon, mediumaquamarine, mediumblue, mediumorchid, mediumpurple, mediumseagreen, mediumslateblue, mediumspringgreen, mediumturquoise, mediumvioletred, midnightblue, mintcream, mistyrose, moccasin, avajowhite, navy, oldlace, olive, olivedrab, orange, orangered, orchid, palegoldenrod, palegreen, paleturquoise, palevioletred, papayawhip, peachpuff, peru, pink, plum, powderblue, purple, purwablue, red, rosybrown, royalblue, saddlebrown, salmon, sandybrown, seagreen, seashell, sienna, silver, skyblue, slateblue, slategray, snow, springgreen, steelblue, tan, teal, thistle, tomato, turquoise, violet, wheat, white, whitesmoke, yellow, yellowgreen.

A classe tem, ainda, métodos para conversão de formatos e para mudança de cor de um objeto. Exemplos:

Definição das cores vermelho e azul escuro usando o primeiro construtor.

Color vermelho(255,0,0);

Color azulEsc(0,0,139);

Definição das cores rosa passando um nome pré-definido e das cores verde escuro e coral no formato html. Note que o símbolo ‘#’ é opcional.

Color rosa("pink");

Color verdeEsc("#006400");

Color coral("FF7F50");

Convertendo formatos.

String cor;

cor = vermelho.getName(); // cor = red

cor = azulEsc.toHTML(); // cor = #00008b

Alterando a cor vermelho para vermelho escuro.

int r = 139, g = 0, b = 0;

vermelho.toRGB(r,g,b);

* 1. **– Util:** classe de métodos utilitários. Exemplos:

O trecho de código abaixo converte uma string para lower case e outra para upper case, respectivamente.

String test;

test = Util::toLower("AlFa"); // test = alfa

test = Util::toUpper("beta"); // test = BETA

Função para decodificar dados que estejam em codificação Base64.

Util::decodeBase64("aGVsbG8gd29ybGQK", test); // test = hello world

Funções para arredondamento para menor e para maior inteiros, respectivamente.

float menorInt, maiorInt;

menorInt = Util::floorFloat(8.99); // menorInt = 8

maiorInt = Util::ceilFloat(8.99); // maiorInt =9

A classe conta também com funções para descompactar os formatos ZLIB e GZIP.

* 1. **– Vector2D:** tal qual o nome diz esta classe define um vetor bidimensional, iniciando-se no ponto (0,0) da tela e indo até o ponto (x,y) definido pelo construtor. A classe possui sobrecarga de operadores para o correto tratamento das operações vetoriais, além de outros métodos para a manipulação dos vetores. Exemplos:

Produto escalar entre a e b.

float escalar;

Vector2D a(3,5);

Vector2D b(3,9);

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

Normalização de c.

Vector2D c(4,3);

c = c.normalize(); // c = ( 0.8, 0.6 )

Arredondamento para maior inteiro.

c.ceil(); // saida = ( 1,1 )

Soma entre vetores, com o operador ‘+’ sobrecarregado.

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

* 1. **– Video:**  classe responsável por gerenciar todos os recursos de vídeo. A classe utiliza o padrão de projeto singleton, ou seja, permite apenas uma única instância de si mesmo. Através dela, tem-se acesso a todas as rotinas pertinentes (de atualização de tela, posicionamento, e outros eventos) para o gerenciamento de vídeo. Como exemplo foi criada um mini-janela. Definimos o tamanho, dado pelas coordenadas x e y conforme o desenho e o modo (WINDOWED, FULLSCREEN) no construtor da classe. Por padrão, a cor de fundo do vídeo é preta, podendo ser alterada. Também adicionamos um ícone e um título para nossa janela. A função refresh() é reponsável por atualizar a tela e deve ser chamada após alguma alteração. Caso contrário, nada de novo aparecerá. É importante saber também que a imagem a ser carregada, neste caso o ícone, deve estar dentro pasta de projeto do programa.

Video video( 300, 100, DisplayMode::WINDOWED );

video.setIcon("nice.jpg");

Mudança da cor de fundo da tela.

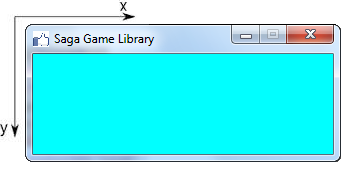
video.setBackgroundColor(Color("aqua"));

Definindo um título para a janela.

video.setTitle( "Saga Game Library" );

Atualizando a janela.

video.refresh();



* 1. **– Bounding Box:** cria um retângulo usando um Vector2D e um par de coordenadas.Possui rotinas para alterar a posição do retângulo e movê-lo. Verifica se há colisão com outro retângulo. É importante no jogo para saber se o personagem pode ou não passar em um determinado local. Por exemplo, um muro deve ser um objeto colidível, e ao aproximar-se dele a classe deve dizer que haverá colisão. Assim, deve ser feito um tratamento adequado para o que o personagem não possa atravessar o muro.
  2. **– Geometrics:** ela é usada para desenhar elementos geométricos básicos. Possui 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 de preenchimento. O construtor padrão inicializa a espessura com 1, a cor de preenchimento como branca e cor da linha como preta. Outro construtor dá ao usuário a liberdade de definir os valores como queira. Com ela é possível desenhar linhas, triângulos, retângulos, retângulos abaulados, elipses, círculos, arcos e ranhuras. Exemplos:

Declaração de vetores que serão usados

Vector2D a(200,150);

Vector2D b(280,150);

Vector2D c(350,150);

Vector2D d(200,100);

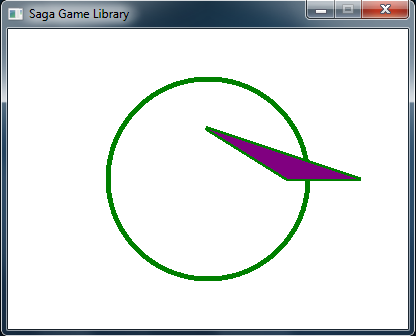
No construtor da classe, dizemos que a espessura da linha é 5 e que sua cor é verde. O terceiro parâmetro diz respeito à cor de preenchimento da forma geométrica a ser desenhada.

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 iria aparecer.

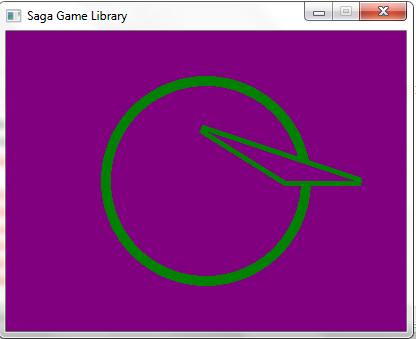
geo.drawCircle(a,100,false);

geo.drawTriangle(b,c,d,true);



É possível preencher o fundo da tela com a função fillBackground(). A tela será colorida com a cor de preenchimento definida no construtor.

geo.fillBackground();



* 1. – Scene: (game loop)
  2. **Time Handler:** é a classe que possui rotinas relacionadas a tempo. Com ela podemos iniciar, parar, pausar e retornar a uma contagem de tempo, entre outras. Exemplo:

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.

unsigned int i;

TimeHandler time;

time.start();

for (i = 0; i<4294967295; i++){}

time.pause();

float t = time.getTicks();

Saída: t = 17.0142 (segundos).

A instrução a seguir segura a execução do programa por 10 segundos.

time.sleep(10.0);

* 1. – **Resource e Ressource Manager:** a classe Ressource é a provedora de recursos. Quando um arquivo é carregado, seja texto, imagem ou áudio, ele é armazenado em uma instância de Ressource. Se o mesmo arquivo for usado novamente, é a Ressource quem o proverá. Assim ele não precisará ser carregado novamente. Ressouce Manager é quem faz o mapeamento dos recursos.

Esta estrutura permite o gerenciamento dos recursos de forma otimizada, fazendo que com 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. Isto é possível pelo fato de a Allegro ter rotinas escritas em Assembly para isso.

É muito importante ter esse tipo de gerenciamento. Especialmente para jogos, porque no geral tem-se uma imagem que se repete uma grande quantidade de vezes. Isso pode acontecer ao criar um cenário que tem muita grama, ou céu, ou mar, sendo que um mesmo sprite é usado para o desenho. Nesses casos a diferença entre ter ou não ter esse gerenciamento de recursos é grande e pode influenciar na performance do jogo. Exemplos serão vistos posteriormente.

1. **– Sgl::input:** pacote responsável por gerenciar as entradas de mouse e teclado.

**2.1 - key\_Code:** contém simplesmente dois tipos enumerados. Ambos representam os respectivos códigos das teclas do teclado e são inteiros não sinalizados. A diferença entre eles é que um guarda os códigos de 8 bits e o outro os de 16 bits. Os de 16 bits são modificadores, como shift, control, alt etc..

**2.2 - KeyboardManager**: usa a Key\_Code e faz o gerenciamento do teclado. Possui duas variáveis do tipo ALLEGRO\_KEYBOARD\_STATE. Uma é para guardar o último estado e outra para guardar o estado atual do teclado. O método update() é quem atualiza essas variáveis. Os outros métodos verificam se determinada tecla foi pressionada, continua pressionada ou se foi solta.

**2.3 – MouseManager:** trabalha de forma semelhante a KeyboardManager, fazendo o gerenciamento das teclas do mouse.

Essas classes trabalham em conjunto com sgl::event. Exemplos serão vistos mais adiante.

1. **– Sgl::event:** é a classe responsável pelo tratamento dos eventos que venham a acontecer, como por exemplo uma entrada do teclado ou um clique do mouse.
   1. **– Event:** classe mais genérica da qual irão derivar as outras. Além de construtor e destrutor, possui um atributo do tipo ALLEGRO\_EVENT e uma função unsigned int getType() que retorna o tipo do evento.
   2. **- Listener:** classe abstrata da qual irão derivar as classes key\_listener, display\_listener e mouse\_listener. Tem um método callMethods que recebe por referência um objeto de Event. A implementação deste fica a cargo das classes filhas.
   3. **– Key\_event:** especialização de Event. Verifica se algo foi digitado no teclado.

* 1. –**Key\_listener:** especialização de Listener. Implementa callMethods, que verifica o tipo de evento e chama a função adequada (keydown(), keyRelease(), ou keyChar()).
  2. Mouse\_event, mouse\_listener, display\_event, display\_listener: mesma estrutura .

**sgl::font**

O pacote sgl::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 um texto na tela. É constituído de duas classes: a classe *FontResource*, especialização da classe *Resource* e a classe *Font*.

A *FontResource* é usada dentro do método load da classe *Font* e funciona da seguinte maneira: recebe um arquivo TTF e um tamanho e, caso ainda não tenha sido colocado, esse arquivo é adicionado em uma instância da classe *ResourceManager*. É possível existir dois arquivos de fonte iguais mas com tamanhos diferentes em *ResourceManager*.

*Font* 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. Abaixo um exemplo de uso da classe:

**Color\* c1 = new Color( 200, 0, 0 );**

Instância de um objeto c1 ponteiro de *Color*, variando somente o parâmetro referente à cor vermelha do sistema RGB.

**Vector2D\* v1 = new Vector2D( 320, 120 );**

Instância de um objeto v1 ponteiro de *Vector2D*, passando as coordenadas horizontal e vertical, respectivamente.

**Font\* f1 = new Font( "alger.ttf", 50 );**

Instância de um objeto f1 ponteiro de *Font*, que recebe o nome do arquivo de fonte e o seu tamanho.

**f1->setColorFont( \*c1 );**

**f1->setPosition( \*v1 );**

**f1->setText( "Fonte: Alger" );**

**f1->setAlignment( FontAlignment::CENTRE );**

Faz com que f1 esteja na cor dada por c1 e na posição dada por v1. Altera também o texto e o alinhamento, que pode ser dos tipos LEFT, RIGHT, CENTRE e INTEGER.

**f1->drawText();**

Desenha o texto na tela.

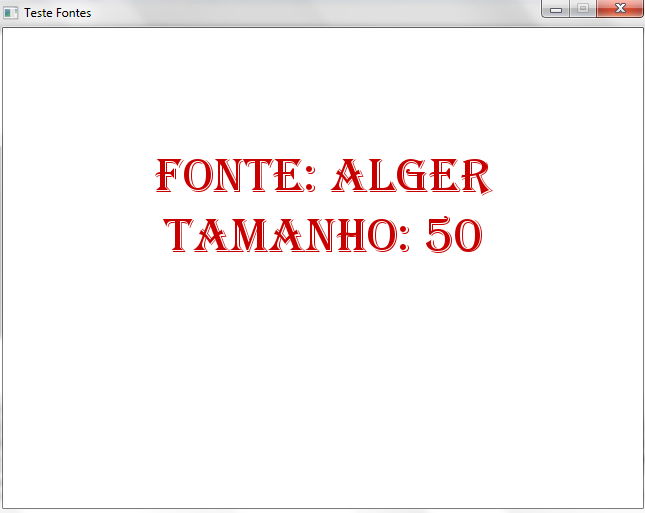
**f1->setText( "Tamanho: 50" );**

**v1->setCoordenate( 320, 180 );**

**f1->setPosition( \*v1 );**

**f1->drawText();**

Altera o texto e a posição na tela e desenha novamente. O trecho de código produz uma saída como na imagem abaixo.



**sgl::audio**

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,* e 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. É importante saber que o método para alterar a velocidade de um arquivo de áudio também altera sua frequência. Seu uso inadequado pode tornar o áudio incompreensível. Não existe suporte ao formato mp3. Exemplo:

**AudioSample\* sample = new AudioSample( "palmas.wav" );**

Cria-se um objeto ponteiro para *AudioSample.*

**sample->setGain( 0.8 );**

**sample->setSpeed( 0.9 );**

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

**sample->setLoopingMode( AudioPlayMode::PLAY\_LOOP );**

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

**sample->play();**

Finalmente, coloca o arquivo para tocar.

**AudioStream\* stream = new AudioStream( "interface.ogg", 4, 1024 );**

Carrega um arquivo para ser reproduzido como trilha sonora e recebe parâmetros para alterar o buffer (em bytes) e a quantidade de amostras.

**stream->setBegin( 14 );**

**stream->play();**

Faz o arquivo tocar a partir dos 14 segundos.