Universidade Federal do Rio de Janeiro

Programa de pós-graduação em Ensino de Física
Campus Macaé

MNPEF
Mestrado Nacional Profissional em Ensino de Física

SBF
SOCIEDADE BRASILEIRA DE FÍSICA

MANUAL DA ACLIVEJS 0.10.

Material instrucional associado à dissertação de mestrado de Wallace Robert da Silva Nascimento, apresentada ao Programa de Pós-Graduação Campus UFRJ-Macaé no Curso de Mestrado Profissional de Ensino de Física (MNPEF), da Universidade Federal do Rio de Janeiro.

Macaé

Abril 2017

Sumário

1 INTRODUÇÃO

A AcliveJS é uma ferramenta computacional criada como um objeto educacional digital para o desenvolvimento de simulações de física a partir de linhas de códigos e com suporte orientado a objeto. Desse modo, o usuário possui à sua disposição uma biblioteca de comandos para a construção dos seus Objetos de Aprendizagem.

A AcliveJS oferece ao usuário uma liberdade maior de criação, mas exigindo um conhecimento técnico um pouco mais apurado. Além disso, também se propõe a ser uma alternativa ao professor que deseja criar as próprias simulações que atendam as características particulares de seus alunos, para que possam trabalhar os modelos ajustadas as suas necessidades.

Para obter uma cópia da AcliveJS, é necessário que o usuário visite o repositório de códigos *GitHub* no seguinte endereço eletrônico: *https://github.com/AcliveJS/Aclive*. Essa ação não exige cadastro e o download é feito clicando num único botão. O arquivo a ser baixado encontra-se com extensão no formato *zip* e, portanto, não possui instalador, ou seja, basta descompactá-lo.

Este manual tem como objetivo apresentar ao usuário a AcliveJS, explicar o seu princípio de funcionamento, avaliando suas funcionalidades e potencialidades, sem adentrar em questões técnicas do seu desenvolvimento. Para a elaboração da biblioteca AcliveJS foram levadas em considerações alguns critérios. O primeiro foi a gratuidade, ou seja, a ferramenta deveria estar disponível para todos. O segundo foi a possibilidade do usuário alterar a AcliveJS de acordo com suas necessidades. Para tanto, optou-se pelo código aberto. Por fim, a linguagem de programação necessitaria ser utilizada na web, isto é, em páginas de internet e, finalmente, teria que ter uma curva de aprendizagem rápida ou, em outras palavras, ser fácil de programar.

Os dois primeiros critérios foram atendidos através do uso de uma Interface de Programação de Aplicativos, cuja sigla é API, do inglês *Application Programming Interface,* de uma biblioteca gráfica e uma linguagem de marcação de hipertexto. Foram utilizados o *WebGL*, o *Threejs* e o *HTML5*, respectivamente. O *WebGL* é a sigla usada para *Web Graphics Library* e trata-se de uma API em *javascript*, disponível a partir do elemento *canvas* do *HTML5*, que oferece suporte para renderização de gráficos 2D e 3D. O *Threejs* é uma biblioteca gráfica desenvolvida a partir de *WebGL*. O HTML5 é sigla usada para *HyperText Markup Language*, na qual o número cinco faz referência à

quinta versão da linguagem de hipertexto *HTML* utilizada para a criação de páginas de internet. Elemento *canvas* do *HTML5* é uma janela gráfica definida pelo usuário e é através dela que será exibida a simulação. A programação é feita através da linguagem interpretada *javascript* cuja sintaxe é muito semelhante ao da linguagem *java*, utilizada no desenvolvimento de *applets*, discutido anteriormente. A AcliveJS é uma biblioteca que foi construída sobre o *Threejs* para permitir que o usuário seja capaz de criar uma simulação utilizando recursos gráficos avançados sem as complicações diretas oferecidas pelo próprio *Threejs*.

É importante compreender o significado da palavra 'biblioteca' utilizada no contexto de programação de computadores, ou o termo em inglês *Library.* Biblioteca é um conjunto de funções ou comandos que atendem a uma necessidade específica. Assim, o *Threejs* é uma biblioteca de funções escritas em *javascript* cuja finalidade é facilitar o uso do *WebGL* e a AcliveJS outra biblioteca, também escrita em *javascript*, com objetivo de facilitar o uso do *Threejs*.

A escolha do *HTML5*, *WebGL*, *Threejs* e *javascript* é justificada pelo fato de serem suportados por várias plataformas, permitindo obter um produto capaz de criar simulações de forma independente, de alto desempenho e que pode ser compartilhado via internet. Pode ser usado em diferentes sistemas operacionais e com ajustes futuros produzir aplicativos para o ensino de física que podem ser executados em *smarthphones* e *tablets*. Portanto, a AcliveJS exige do educador os conhecimentos de Física e de programação necessários no processo de se criar uma simulação, via linhas de código.

2 COMO A ACLIVEJS FUNCIONA

Após fazer o download da página do *GitHub* e descompactar o arquivo com extensão *zip* no computador, deverá aparecer um conjunto de pastas como mostrado na Figura 2.1.

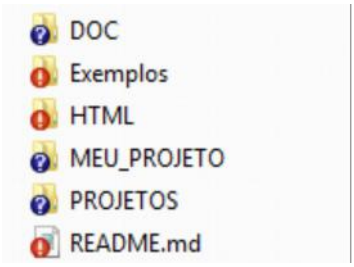


Figura 2.1: Pastas e arquivos da AcliveJS

A pasta **DOC** possui dois arquivos, *javascript* e *aclivejs,* no Formato Portátil de Documento ou PDF, do inglês *Portable Document Format*. O primeiro contém um breve manual de como utilizar a linguagem interpretada e o segundo explica quais as regras para trabalhar com a AcliveJS. A pasta **Exemplos** contém exemplos que mostram algumas das funcionalidades da biblioteca. A pasta **HTML** possui arquivos de ajuda no formato *HTML*. A pasta **MEU_PROJETO** possui um outro diretório chamado **js** e um arquivo *main.html*. A pasta **PROJETOS** é o local onde serão armazenados os projetos dos usuários. A diferença entre os diretórios **MEU_PROJETO** para **PROJETOS** está no fato de que no primeiro há um modelo para um projeto novo e o segundo serve como um repositório para os projetos desenvolvidos.

O arquivo **readme.md:** possui informações sobre a biblioteca. Ao visitar a página do projeto, no repositório *GitHub*, o *readme* estará disponível para leitura com dados sobre atualizações e novas funcionalidades.

Conhecido todo sistema de pastas e arquivos da AcliveJS, a Figura 2.2 exibe um fluxograma do seu princípio de funcionamento.

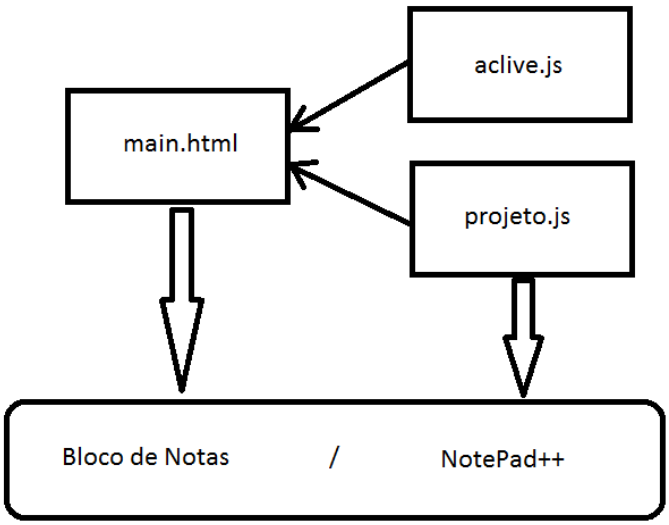


Figura 2.2: Esquema do princípio de funcionamento da AcliveJS

Os arquivos *aclive* e *projetos* possuem extensão *js*, de *javascript*, e ficam no diretório **js**, dentro da pasta **MEU_PROJETO**.

O arquivo *main.html* é uma página no formato *HTML5*, que pode ser aberta por um navegador de internet (*browser* em inglês), tanto no modo *offline*, com o arquivo salvo no computador, ou no modo *online*, hospedado na internet. Quando o *main* é aberto, automaticamente importa a biblioteca contendo todas as novas funcionalidades oferecidas através do arquivo *aclive*. *Main* também carrega a simulação desenvolvida pelo usuário, por intermédio do arquivo *projeto*, e em seguida exibe a simulação na janela de visualização, ou seja, no *canvas* do *HTML5*. Ambos os arquivos, *aclive* e *projeto*, possuem extensões '*js*', usadas para designar arquivos que possuem códigos escritos na linguagem interpretada *javascript*. Este processo está indicado na Figura 2.2 pelas setas que ligam os documentos *aclive* e *projeto* ao *main*. Todos os três arquivos são editáveis, entretanto, sendo o *aclive* o núcleo do sistema é conveniente alterá-lo apenas para inserir novas funções, buscando enriquecer a biblioteca. Qualquer usuário pode fazer isto e submeter estas alterações ao repositório do *GitHub*. Estas mudanças serão analisadas e testadas, e se aprovadas, passarão a fazer parte do corpo da biblioteca e os devidos créditos serão dados ao programador. Na maioria das vezes, não será necessário ao educador mexer no arquivo *aclive*, por isso a ausência da seta ligando-a

ao Ambiente de Desenvolvimento Integrado ou IDE, do inglês *Integrated Development Environment*, representadas na Figura 3.2 pelo Bloco de Notas e o *NotePad++*.

P*rojeto* é o arquivo de trabalho do usuário onde serão inseridos códigos da simulação. Portanto, na maior parte do tempo, o desenvolvedor estará com este documento aberto no IDE. Por padrão, o *main* está configurado para importar a simulação com o nome *projeto,* mas é possível modificá-lo. Se for este o caso, será preciso editar o arquivo *main* no trecho do código indicado abaixo em negrito:

<script src="js/Aclive.js"></script>

*<script src="js/**projeto.js**"></script>*

Caso o usuário necessite desenvolver uma simulação mais interativa que aceita entradas via teclado ou mouse por meio de botões (*buttons*), *Type Radio*, *CheckBox*, entre outros recursos oferecidos pelo *HTML5*, será preciso editar o arquivo *main*. O processo de edição deve ser realizado com muito cuidado, e nada deve ser alterado no código a fim de evitar que a AcliveJS deixe de funcionar. Para impedir contratempos, foram indicadas as regiões no código em *main* onde é possível inserir os *inputs* do *HTML5*. Isso determinará onde estas entradas aparecerão no aplicativo, que pode ser na parte superior ou inferir da janela de visualização. A seguir, o trecho de código *HTML* no arquivo *main*, que permite ao usuário reconhecer qual o espaço reservado para inserir entradas:

<!---Seu código HTML5 - Entradas de dados via formulários aqui (veja o manual) --->

(forms e inputs aqui)

<!--.....*-->*

Os arquivos *main*, *projeto* e *aclive*, juntos, fornecem o suporte necessário para a construção de uma simulação. Os comandos utilizados pelo usuário ao escrever os códigos são os nativos do *javascript* e os desenvolvidos através de funções (*function*), escritas no arquivo *aclive*, que visam atender necessidades específicas e facilitam o uso dos comandos do *Threejs*. Portanto, é imprescindível que usuário aprenda manusear a codificação necessária para criar os modelos em *projeto* e inserir dados de entrada em *main*. Estes arquivos são encontrados na pasta MEU_PROJETO. O bloco inferior da Figura 2.2 refere-se ao IDE, que é um programa externo. Pode ser utilizado Bloco de Notas, *NotePad++*, *Sublime*, *Context*, entre outros.

AcliveJS fornece uma estrutura conceitual simplificada, cujo intuito é permitir ao educador focar na construção do modelo da física a ser desenvolvido, evitando que este se perca nas nuances técnicas decorrentes de uma programação mais pesada, como ocorrem com o *WebGL* e o *Threejs*. No que se refere a novas funcionalidades, elas estão disponíveis no arquivo *aclive* e foram elaboradas para atender as necessidades desta pesquisa. Em termos de potencialidade, a AcliveJS herda todos os recursos do *WebGL* e do *HTML5*, tais como ambiente 3D, uso de figuras e sons, possibilidade de apresentações em vídeos e animações 3D.

2.1 A Estrutura de uma Simulação

Todo o programa desenvolvido utilizando a AcliveJS deve obedecer o modelo abaixo, escrito a partir do arquivo projeto.

```
// Declara as variáveis globais aqui
Function Ambiente()
{
        // Código aqui
}
Function Simulacao()
{
        // Código aqui
}
```

Function é uma palavra reservada do *javascript*, já *Ambiente* e *Simulacao,* sem cedilha e til, são palavras reservadas da AcliveJS. As duas barras (//) são linhas de comentários e tudo que vier escrito após as barras, será ignorado pelo interpretador. As variáveis globais devem ser declaradas antes da *function Ambiente* utilizando a palavra reservada *var.* Em seguida escreve-se a *function Ambiente()*, onde os códigos devem vir obrigatoriamente entre o par de chaves e é neste espaço que as variáveis receberão seus valores iniciais e também onde os objetos que fazem parte da cena serão criados. Esta *function* é executada apenas uma vez quando iniciada a simulação. O próximo passo está em escrever a *function Simulacao()*. Esta função é executada constantemente em repetições sucessivas, *loop* ou *looping,* termo em inglês utilizado por programadores. A

quantidade de repetições por segundo é denominada *fps, frames per second* ou quadros por segundo. Quanto mais rápido for o processador, maior também será a taxa de *fps*. É possível controlar o *fps* para executar a simulação numa taxa específica. É em *Simulacao()* que as variáveis serão atualizadas de acordo com o modelo e os objetos sofrerão as transformações correspondentes, como mudança de posição, rotação, desenho de vetores, rótulos (*labels)*, entre outros. Todas as atualizações, após os cálculos realizados pelo modelo, ocorrerão nesta função que serão executados em *looping* até que a simulação seja interrompida. Abaixo, segue um exemplo de uma simulação desenvolvida utilizando AcliveJS através da edição do arquivo *projeto*.

```
var CE=[];
var w, A, x, t;
function Ambiente()
{
        w=2; A=50; x=0; t=0;
        Janela3D('rgb(100,40,40)',0.75,0.95);
        InserirObservador(45,0.1,20000);
        PosicaoDoObservador(60.0, 60.0, 60.0);
        OlharPara(0.0, 0.0, 0.0);
        SistemaRetangular(50,false, false, false);
        GradeXY('rgb(255,255,0)', 'rgb(50,50,0)',100, 5, false)
         CamOrbita();
        Janela2D('rgb(40,40,100)',0.2,0.95);
        CE[1] = new CorpoExtenso(2,2,0.5,x,0,0,0,0,0,'rgb(255,0,0)',false,0);
}
function Simulacao()
{
        x=A*Math.cos(w*t);
        CE[1].CorpoExtenso.position.set(x,0,0);
        t+=0.01;
}
```

Não é preciso compreender este código por enquanto, mas somente sua estrutura principal a partir do que foi descrito no parágrafo anterior. Nas duas primeiras linhas do programa, antes da *function Ambiente,* foram declaradas cinco variáveis, CE, w, A, x e t, que correspondem as variáveis globais. Estas foram inicializadas dentro de *Ambiente()* de acordo com a condição inicial do modelo físico envolvido. As linhas seguintes possuem funções chamadas a partir da *aclive*, que incluem a inicialização e inserção de alguns objetos, como *Janela3D*, *InserirObservador*, entre outros. Em *Simulacao()* possui o modelo em si, neste caso específico trata-se do Movimento Harmônico Simples (MHS). A primeira linha atualiza a posição x de acordo com a equação do MHS, a segunda atribui este valor ao objeto CE[1], que é do tipo corpo extenso, e a terceira atualiza o tempo. Por enquanto, o interesse está apenas em explicar como um programa desenvolvido na AcliveJS deve ser estruturado e mostrar que, independente do modelo escolhido, as regras para construção de uma simulação são as mesmas e devem ser obedecidas. Mais adiante, o MHS será explorado em detalhes e um novo programa será criado, a fim de atender a algumas necessidades específica sobre o tema. Simulações computacionais costumam possuir as mesmas regras de base de códigos. Este paradigma considera a simulação constituída de três partes:

a. *Reserva de memória:* Ao declarar variáveis, o usuário está na verdade reservando espaço na memória do computador, cujo objetivo é guardar os valores que serão utilizados durante a simulação.

b. *Estado inicial do sistema:* Toda simulação consiste em atribuir condições iniciais a partir de variáveis criadas para este fim. Por isso, são caracterizadas como grandezas de estado e podem assumir determinados valores dentro do intervalo de validade do fenômeno simulado.

c. *Looping:* É caracterizado pelo fluxo contínuo de execução de uma série de comandos repetidamente. Todos os cálculos e atualizações do visual da simulação ocorrem dentro deste bloco do programa.

A Figura 2.3 exibe um fluxograma da estrutura dos programas construídos com a *AcliveJS.*

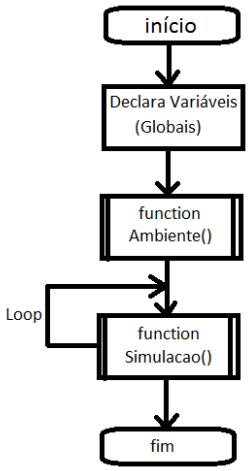


Figura 2.3: estrutura dos programas AcliveJS

2.2 Inserindo textos e imagens

Alguns programas utilizados para o desenvolvimento de simulações de Física, como o Modellus ou o EJS, oferecem opções para o usuário inserir uma descrição textual do objeto de estudo. Estes textos podem ser uma introdução, um roteiro ou um resumo teórico. É possível fazer o mesmo com a AcliveJS por meio da edição do arquivo *main*. Há um espaço reservado neste documento especificamente para este fim como ilustrado abaixo.

<!---Seu código HTML5 - Entradas de dados via formulários aqui (veja o manual) --->

(Descrição Textual ou Resumo Teórico aqui)

<!--.....-->

Entre as *tags* de comentário (<!-- e -->) o usuário poderá utilizar os recursos disponíveis da linguagem *HTML5*. A Figura 2.4 mostra um exemplo do uso de imagem e textos utilizando a AcliveJS. Com os recursos oferecidos pelo *HTML* tudo pode ser editado, deste o tamanho das imagens e sua posição até o tipo de fonte, tamanho e cor dos caracteres impressos na tela. A listagem encontra-se Apêndice 1.

Figura 2.4: Exemplo de exibição de textos e imagens

Outra opção, para quem não deseja ficar escrevendo códigos do *HTML*, refere-se ao uso de um editor externo, que será discutido no Capítulo 4.

2.3 Construindo Modelos com a AcliveJS

Nos tópicos 2.1 e 2.2, não foi discutida a Física envolvida no modelo utilizado como exemplo, já que o interesse era apenas mostrar o princípio básico de funcionamento da AcliveJS e as regras para se criar uma simulação utilizando-a. O intuito agora é mostrar como construir uma simulação completa, a partir de objetivos específicos delineados pelo que será discutido a respeito do fenômeno físico de interesse. Os rumos que a simulação tomará ao ser desenvolvida dependerá de algumas etapas:

(a) Escolher o assunto da Física a ser explorado e delinear com clareza os objetivos a serem atingidos com o uso da simulação;

(b) O tipo de atividade que deseja utilizar, se será exploratória ou expressiva;

(c) Se necessário, escrever uma descrição textual. Pode ser um roteiro, texto explicativo, entre outros;

(d) Programar o modelo utilizando as equações de estado, obtidas do estudo do tema, ajustando-as as necessidades do projeto. Algumas simulações utilizam métodos

numéricos, como o de Euler ou Runge-Kutta, para obtenção de resultados. O uso de um ou outro, dependerá da Física envolvida e, claro, da escolha do professor.

A compreensão das regras de como trabalhar com a AcliveJS e das etapas descritas neste tópico ajudarão na construção do Objeto de Aprendizagem que o professor desejará aplicar em sala de aula.

2.3.1 Escolha do Modelo: Movimento Harmônico Simples (MHS)

O assunto escolhido para mostrar como produzir uma simulação completa com a AcliveJS foi o estudo do MHS, sistema massa-mola ideal sem atrito. Isto significa que a mola é desprovida de massa e foram desprezadas as forças de contato entre a superfície e o bloco. Sobre este tema, o objetivo da simulação é facilitar o entendimento do que vem a ser amplitude do movimento, frequência angular, ângulo de fase e verificar o comportamento cinemático desse sistema a partir das forças envolvidas. Em seguida, avaliar o que ocorre com as oscilações a partir da mudança da massa do bloco e da constante elástica da mola. Será comparado o movimento de dois blocos de massas inicialmente iguais presos a molas de mesma constante elástica. Como os sistemas são idênticos, oscilarão da mesma forma quando abandonados da mesma posição. Alterando os valores das grandezas envolvidas em casa oscilador e por meio de comparações entre eles, será possível analisar o que acontece baseando-se na teoria da Física.

Para facilitar o processo, a simulação foi desenvolvida pensando numa aplicação utilizando atividade exploratória através de uma possível aula expositiva. Não serão elaborados elementos de entrada, através de *inputs* do HTML, portanto todas as alterações de valores das grandezas serão modificadas diretamente pelo código escrito no *projeto*. Isto é feito alterando as condições iniciais e, em seguida, reiniciando o navegador. A Figura 2.5 ilustra o layout escolhido para esta simulação:

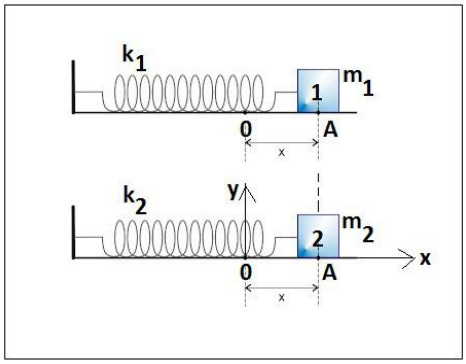


Figura 2.5: Layout da Simulação MHS

2.3.2 A Física do MHS

Considere o sistema massa-mola representado na Figura 2.6 abaixo.

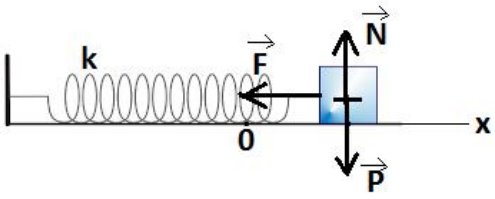


Figura 2.6: Sistema massa-mola sem atrito

Um corpo de massa m está conectado à extremidade móvel de uma mola de constante elástica k. O bloco é sustentado por uma força exercida para cima, a qual anula a força da gravidade que atua para baixo. Quando a peça é movimentada a partir de sua posição de equilíbrio estável, em que a mola tem seu comprimento relaxado, e depois liberado oscilará ao longo de uma linha horizontal. A Figura 2.6 mostra o corpo num instante em seu ciclo de oscilação quando acontece de estar à direita de sua posição de equilíbrio estável. Nestas circunstâncias, sua coordenada x da posição medida a partir do x=0, terá um valor positivo. A mola é estendida e, assim, exerce uma força horizontal F sobre o corpo. Esta força atua para a esquerda, portanto, seu valor é negativo. Quando o bloco encontra-se em posições negativas, então a mola estará comprimida e a força F atuará para a direita, sendo positiva. A força F é denominada força elástica e a distância x do ponto de equilíbrio é chamada elongação da mola. A elongação máxima atingida pela mola, em módulo, é conhecido como amplitude de oscilação e geralmente é representada pela letra A. A relação entre F e x é dada pela Lei de Hooke mostrada na Equação 2.3.1, onde k é um valor que depende da natureza do material e da geometria da mola. Denominada constante elástica da mola, k é sempre positivo. O sinal negativo na Equação 2.3.1 indica que o sentido da força F e do deslocamento em torno do ponto de equilíbrio estável são sempre contrários.

$$F = -kx \qquad (2.3.1)$$

Existem 3 forças atuando sobre o bloco, peso, normal e a força elástica. Peso e normal se cancelam mutuamente, restando como força resultante apenas a força elástica. Substituindo esta força na 2ª Lei de Newton e escrevendo a aceleração do bloco como uma derivada segunda do tempo, obtêm-se a Equação 2.3.2. Esta é uma equação diferencial ordinária de 2ª ordem.

$$\frac{d^2x}{dt^2} + \frac{k}{m}x = 0 \qquad (2.3.2)$$

Equação 2.3.2 admite duas soluções, ambas funções exponenciais de base *e* elevada a uma constante multiplicada pela variável tempo. Esta constante é positiva para uma solução e negativa para a outra. Para uma solução mais completa, admite-se a combinação linear dessas duas soluções. Outra possibilidade, está em escrever esta última resposta por meio de funções seno e cosseno, tornando-a mais elegante e fácil de interpretar. A manipulação deste último resultado permite obter uma solução baseada em cosseno com diferença de fase, que é mostrada na Equação 2.3.3. Esta será a equação de estado, ou evolução, utilizada para esta simulação.

$$x(t) = A.cos(\omega t + \phi) \qquad (2.3.3)$$

Sendo:

$$\omega = \sqrt{\frac{k}{m}} \qquad (2.3.4)$$

Segue as grandezas presentes nas Equações 2.3.3 e 2.3.4:

x – posição da extremidade da mola presa ao corpo;

A – elongação máxima da mola;

ω – frequência angular do movimento;

ϕ – ângulo de fase do movimento;

t – tempo;

k – constante elástica da mola;

m – massa do corpo preso a mola;

De acordo com a Equação 2.3.3, a coordenada *x* da posição do bloco é função da variável independente t. As grandezas A, ω e ϕ são constantes. Sendo a amplitude a distância máxima atingida pelo corpo a partir da coordenada x do equilíbrio estável do sistema. A frequência angular, ω, fornece o número de oscilações que o bloco executa num intervalo de tempo de 2π segundos. O ω depende da massa do bloco e da dureza

da mola, caracterizada pela constante elástica k, dada em Newtons por metro. Quanto maior a massa menor será o valor de ω, indicando que a frequência de oscilação reduzirá e o período, que é o tempo que o bloco leva para ir e voltar, será maior. Em contrapartida, quando mais rígida for a mola, isto é, quando maior for o valor de k maior será a frequência de oscilação e menor o período do movimento. Para obter a frequência em Hertz basta dividir ω por 2π. Um sistema massa-mola sem atrito pode ser descrito matematicamente através das projeções das grandezas cinemáticas posição, velocidade e aceleração sobre o eixo horizontal considerando um ponto, em P', que descreve um Movimento Circular Uniforme (MCU), como ilustrado na Figura 2.7.

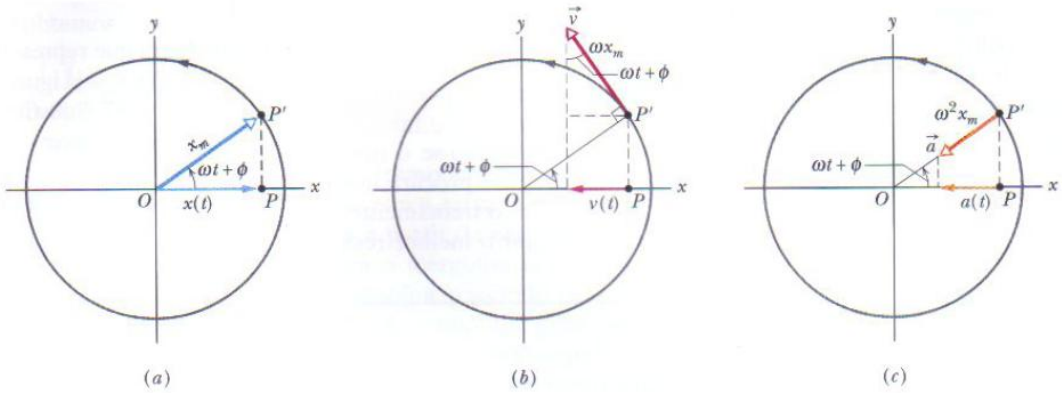


Figura 2.7: Estudo do MHS massa-mola como projeção do MCU.

O ângulo de fase ϕ determinará a posição inicial do corpo, sendo x=+A para ϕ=0, x=0 para ϕ=π/2 e x=-A para ϕ=π.

2.3.3 Construindo a cena em Ambiente()

A codificação foi realizada utilizando o IDE ConText (http://www.contexteditor.org). A Figura 2.8 ilustra o código base digitado no ConText.

```
//
function Ambiente()
{

}

function Simulacao()
{

}
```

Figura 2.8: Código base da AcliveJS

O próximo passo consiste em declarar as variáveis que correspondem às grandezas envolvidas no problema. A primeira menção da variável a configura na memória, ou seja, declarar uma variável é dizer ao programa para reservar um espaço da memória para guardar os valores que serão atribuídos a ela para que mais tarde seja possível fazer referência a esta variável no *script*. Variáveis devem ser declaradas antes de usá-las. Isto é feito através da palavra-chave *var*, do *javascript*, seguido do nome da variável. A Figura 2.9 ilustra esse processo.

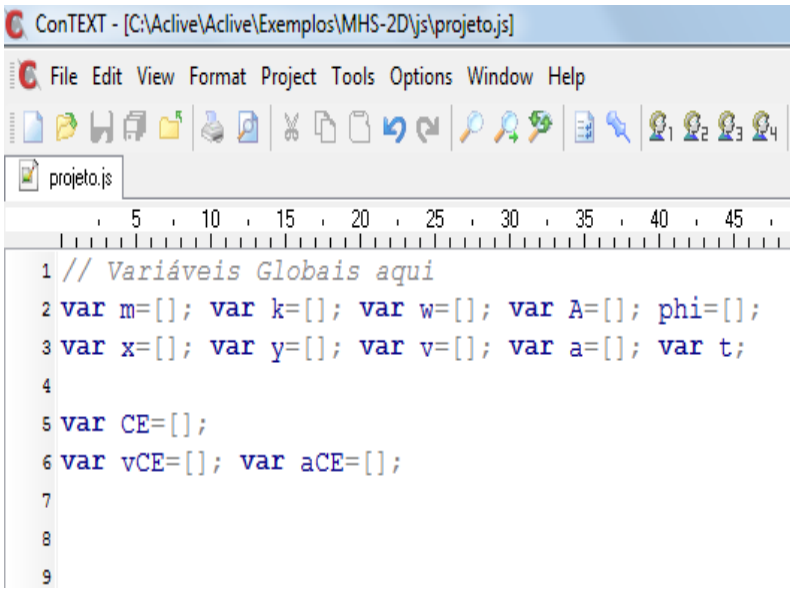


Figura 2.9: Declarando variáveis.

Na linha 2 foram declaradas as variáveis m, k, ω, A e ϕ, correspondendo a massa, constante elástica da mola, a frequência angular, amplitude do movimento e o ângulo de fase, respectivamente. Na linha 3, são declaradas as coordenadas x e y da posição do corpo, em seguida sua velocidade e a aceleração. Por fim, na mesma linha, foi declarada a variável t. Com exceção de t, todas possuem o sinal de igual seguido de colchetes. Isto indica que esta variável é do tipo *array*, ou seja, possuem subíndices. Variáveis *arrays* são chamadas por índices e torna-se muito útil quando a simulação apresenta mais de um objeto na cena. Por exemplo, x[1] e x[2] corresponderiam a coordenada x do corpo 1 e do corpo 2, respectivamente. Esta opção é útil já que o desejo é comparar dois corpos que oscilam e assim tratar as grandezas do primeiro bloco com o número 1 entre colchetes e do segundo bloco com o número 2 entre colchetes. CE é um mnemônico para Corpo Extenso. Basicamente é a variável que vai armazenar o objeto nativo da AcliveJS chamado *CorpoExtenso*. Também é do tipo

array. Usar *arrays* para simulações que possuem mais de um objeto não é uma regra. O usuário poderia optar por declarar uma variável para cada corpo, mas neste caso, seriam necessárias duas para cada grandeza física do problema, ou seja, nove variáveis a mais. Uma simulação consistindo de diversas entidades, como um sistema de partículas, declarar variáveis sem o uso de *array* torna o código confuso, além de trabalhoso e, portanto, o uso de *arrays* se faz necessária.

O próximo passo está em atribuir valores às variáveis, de acordo com as condições iniciais do problema. *Ambiente()* é executado apenas uma vez e é o local reservado para esta finalidade, bem como criar todos os objetos que estarão presentes na cena. A Figura 2.10 ilustra a atribuição dos valores iniciais das variáveis. O ponto e vírgula (;) é um separador de declarações e deve ser usado, caso contrário, acusará erro.

```
9
10 function Ambiente()
11 {
12          // Condições iniciais aqui
13          m[1]=1.0; k[1]=1.0; A[1]=50.0; phi[1]=(Math.PI/180)*0.0;
14          w[1]=Math.sqrt(k[1]/m[1]);
15          x[1]=A[1]*Math.cos(w[1]*t + phi[1]); y[1]=-1;
16          v[1]=(-1)*w[1]*A[1]*Math.sin(w[1]*t + phi[1]);
17          a[1]=(-1)*Math.pow(w[1],2)*y[1];
18
19          m[2]=1.0; k[2]=1.0; A[2]=50.0; phi[2]=(Math.PI/180)*90.0
20          w[2]=Math.sqrt(k[2]/m[2]);
21          x[2]=A[2]*Math.cos(w[1]*t + phi[2]); y[2]=4;
22          v[2]=(-1)*w[2]*A[2]*Math.sin(w[1]*t + phi[2]);
23          a[1]=(-1)*Math.pow(w[2],2)*y[2];
24          t = clock.start();
```

Figura 2.10: Atribuição dos valores iniciais das variáveis

Na linha 13 foram atribuídos valores para a massa m do bloco, a constante k da mola, a amplitude A do movimento e ao ângulo de fase ϕ (phi). O ângulo ϕ deve estar em radianos. Assim, basta substituir o valor 0.0 por um valor em graus que a multiplicação *Math.PI/180* garantirá um ângulo expresso em radianos. Na linha 14, o valor atribuído a ω vem da Equação 2.3.4, onde *Math.sqrt()* é o comando usado para extrair a raiz quadrada de um número colocado no argumento. Na linha 15, foram inicializadas as coordenadas x e y do bloco. O bloco não se move na direção y, logo possui um valor fixo. Entretanto, oscila na direção x onde foi atribuído o valor utilizando a Equação 2.3.3. A linha 16 calcula a velocidade inicial do bloco e a 17 a aceleração inicial. A equação usada na linha 16 é obtida derivando a Equação 2.3.3 em função do tempo. E a equação da linha 17 é a derivada da equação da linha 16, substituído o valor de x da Equação 2.3.3. O processo se repete para a corpo 2, a partir

da linha 19 até a 23. Na linha 24, foi iniciada a variável independente t a partir do relógio interno do computador.

O passo seguinte consiste em criar a cena utilizando as funções que foram desenvolvidas para a realização deste projeto, ou seja, a biblioteca AcliveJS. Estes comandos não estão em inglês, como os nativos do *javascript*, mas sim em português. Isto representa uma facilidade de uso quando comparado ao *Threejs* e visa a atender, num primeiro momento, aos usuários brasileiros. A Figura 2.11 mostra o restante do código presente em *Ambiente()*.

```
24        t = clock.start();
25
26        Janela3D('rgb(220,240,240)', 0.75, 0.95);
27        InserirObservador(45,0.1,20000);
28        PosicaoDoObservador(0.0,0.0,80.0);
29        Ortografica(0,0,15,false,false,true,25);
30        SistemaRetangular(50,false,false,false);
31        GradeXY('rgb(0,0,255)','rgb(50,50,0)',100,5,false);
32
33        CE[1]=new CorpoExtenso(2,2,0.5,x[1],y[1],0,0,0,0,'rgb(255,0,0)',false,0);
34        CE[2]=new CorpoExtenso(2,2,0.5,x[2],y[2],0,0,0,0,'rgb(255,0,0)',false,0);
35
```

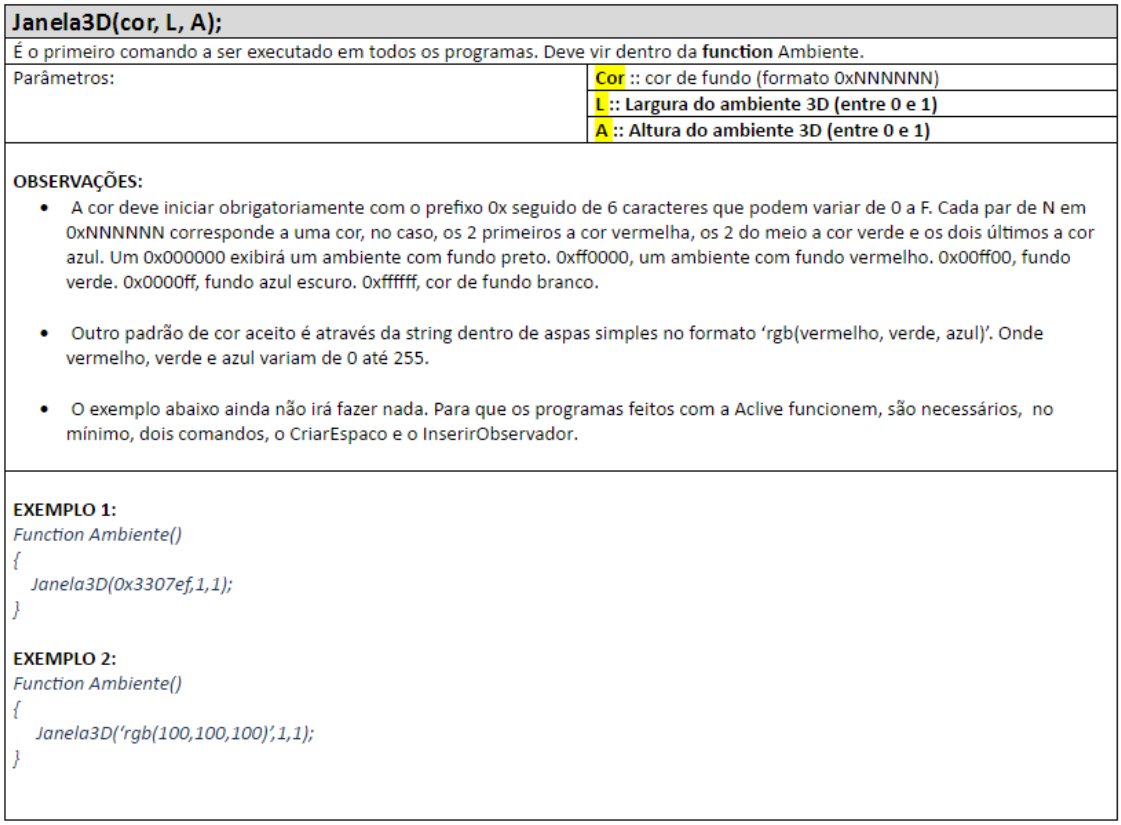
Figura 2.11: Codificação dentro da *function Ambiente()*.

Cada comando da AcliveJS é, na verdade, uma *function* que foi elaborada para cumprir requisitos específicos, resumindo um conjunto de comandos do *Threejs* num único com o nome da função na língua portuguesa, de acordo com a sua finalidade. Como exemplo, para habilitar a visualização 3D seriam necessárias várias linhas de códigos escritas em *javascript* utilizando as funções do *Threejs*, em inglês. A AcliveJS condensa todos estes comandos numa única função, em português, chamada *Janela3D*.

Retornando a Figura 2.11, a primeira função da AcliveJS a ser utilizada é a *Janela3D*, na linha 26. O objetivo é criar uma janela de visualização através do c*anvas* do *WebGL* e é onde será construída o cenário da simulação. *Janela3D* possui três parâmetros, são eles: a cor de fundo, a porcentagem horizontal e vertical do tamanho da janela. Na pasta *HTML,* há um arquivo denominado *comandos.html,* informando os parâmetros utilizados pela função com uma descrição sobre cada um deles. Este documento contém um resumo sobre a biblioteca e tabelas que apresentam cada função. Na mesma pasta é possível acessar estas tabelas, ou ficheiros, de forma isolada bastando

procurar pelo nome do comando nos arquivos com extensão *html* presentes na pasta *HTML*.

Na Figura 2.12 é exibido o ficheiro correspondente ao comando *Janela3D* que foi aberto a partir do arquivo *Janela3D.html*.



Janela3D(cor, L, A);

É o primeiro comando a ser executado em todos os programas. Deve vir dentro da **function** Ambiente.

Parâmetros:	**Cor** :: cor de fundo (formato 0xNNNNNN)
	L :: Largura do ambiente 3D (entre 0 e 1)
	A :: Altura do ambiente 3D (entre 0 e 1)

OBSERVAÇÕES:

- A cor deve iniciar obrigatoriamente com o prefixo 0x seguido de 6 caracteres que podem variar de 0 a F. Cada par de N em 0xNNNNNN corresponde a uma cor, no caso, os 2 primeiros a cor vermelha, os 2 do meio a cor verde e os dois últimos a cor azul. Um 0x000000 exibirá um ambiente com fundo preto. 0xff0000, um ambiente com fundo vermelho. 0x00ff00, fundo verde. 0x0000ff, fundo azul escuro. 0xffffff, cor de fundo branco.
- Outro padrão de cor aceito é através da string dentro de aspas simples no formato 'rgb(vermelho, verde, azul)'. Onde vermelho, verde e azul variam de 0 até 255.
- O exemplo abaixo ainda não irá fazer nada. Para que os programas feitos com a Aclive funcionem, são necessários, no mínimo, dois comandos, o CriarEspaco e o InserirObservador.

EXEMPLO 1:

```
Function Ambiente()
{
   Janela3D(0x3307ef,1,1);
}
```

EXEMPLO 2:

```
Function Ambiente()
{
   Janela3D('rgb(100,100,100)',1,1);
}
```

Figura 2.12: Ficheiro do comando *Janela3D*.

Todos os comandos da AcliveJS deste exemplo possui um ficheiro correspondente dentro da pasta *HTML*. O próximo passo consiste em inserir uma câmera no ambiente 3D. O nome "câmera" foi trocado por "observador". *Janela3D*, originalmente, era chamado *CriarEspaco*. Como é a primeira linha a ser digitada dentro de *Ambiente()*, o objetivo era fazer com que o educador trabalhasse de modo mais intuitivo, raciocinando da seguinte maneira: "estou 'criando' um espaço 3D, agora posso inserir o observador neste espaço". Com a evolução dos comandos, foi necessário desenvolver outras janelas de visualização, uma 2D para exibição de textos e informações, e outra 2D para gráficos. A partir disso, buscando seguir um padrão, o *CriarEspaco* foi renomeado para *Janela3D*. No entanto, a ideia inicial permanece: primeiro é preciso 'criar' o espaço para depois inserir coisas dentro dele. É natural

imaginar que após criar o espaço é necessário inserir o observador. *InserirObservador,* na linha 27*,* possui três parâmetros. São eles: ângulo de visão em graus, corte de perto e corte de longe. Para a maioria das aplicações um ângulo de 45 graus funcionará muito bem. É bom deixar o corte de perto entre 0.1 e 0.5, e corte de longe entre dez e vinte mil. Da mesma forma que *Janela3D*, *InserirObservador* pode ser consultado através dos ficheiros presentes na pasta *HTML*.

Na linha 28, *PosicaoDoObservador* permite posicionar o observador na cena. Foi colocado na posição cujas coordenadas cartesianas é x=0, y=0 e z=80.0. Quando usado em *Ambiente()*, fica estabelecido a posição inicial, mas se usado em *Simulacao()*, o observador poderá se mover durante as atualizações seguindo as regras programadas pelo usuário.

Ortografica, na linha 29, estabelece o modo de visualização da janela, neste caso, uma visão 2D no plano xy.

SistemaRetangular, na linha 30*,* é um comando que exibe eixos cartesianos na tela na posição x=0, y=0 e z=0. Este comando possui quatro parâmetros, são eles: tamanho dos eixos e três campos do tipo *booleano* que pode receber *true* (verdadeiro) ou *false* (falso) – mantido em inglês. Por padrão, todos são falsos. Se verdadeiro, o sistema exibirá o eixo negativo em pontilhado. O primeiro *booleano* refere-se ao eixo x, o segundo ao y e o terceiro ao z. Cada eixo é identificado por uma cor específica, vermelho para x, verde para y e azul para z. Finalmente, na linha 31, a função *GradeXY* desenha um plano segmentado.

O tamanho da grade foi acertado para o mesmo valor da amplitude do movimento dos blocos, com subdivisões de 5 em 5, totalizando 20 linhas no eixo x e 20 no eixo y. Os principais objetivos dos comandos *GradeXY* e *SistemaRetangular* é servir de elementos de apoio que facilitarão a observação das posições dos objetos animados.

Finalmente serão criados os blocos que oscilarão no ambiente. São dois paralelepípedos com tamanhos ajustados por meio de parâmetros repassados para o comando *CorpoExtenso,* da AcliveJS. Foi dada a denominação *CE* como um mnemônico para Corpo Extenso. Possui 12 parâmetros: 3 para as dimensões x, y e z; 3 para as coordenadas x, y e z da posição; 3 para rotações dos eixos x, y e z referente a orientação; 1 para cor; 1 para exibição em formato preenchido ou arame e 1 para o tipo de material. Foram adotadas as dimensões 2, 2 e 0.5 para comprimento, largura e altura, respectivamente. Cada objeto assume os valores iniciais x e y armazenados em *x[1]*, *y[1]*, *x[2]* e *y[2]*. *CE[1]* refere-se ao Corpo Extenso 1 e CE[2] ao Corpo Extenso 2. A

AcliveJS permite o usuário simular fenômenos com quantos objetos desejar, o limite está no poder de processamento da máquina. Os objetos são criados em *javascript* através do comando *new*.

A Figura 2.13 mostra a cena com todos os elementos descritos na listagem.

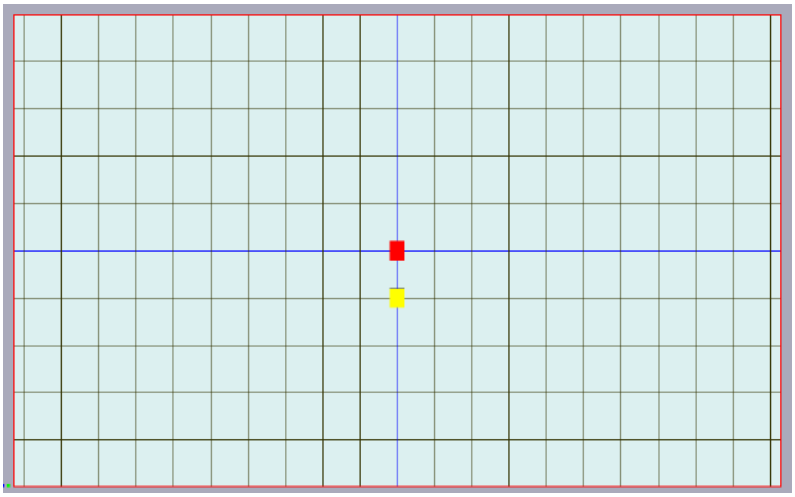


Figura 2.13: Primeira execução da simulação do MHS.

2.3.4 Equações de Evolução

A Equação 2.3.3 corresponde a equação de evolução do sistema físico a ser simulado. É a partir dela que o sistema mudará a posição do objeto de acordo com o número de quadros por segundo. Esta evolução é um processo iterativo e depende da variável tempo cuja taxa de atualização acompanha o relógio interno da máquina, atribuída na linha 24. As equações de evolução derivam da resolução dos modelos teóricos por métodos analíticos exatos, ou discretos via métodos numéricos aproximados, dadas por equações diferenciais ordinárias. Serão necessárias duas equações, uma para CE[1] e outra para CE[2] como mostra a Figura 2.14.

```
43 function Simulacao()
44 {
45         x[1]=A[1]*Math.cos(w[1]*t + phi[1]);
46         x[2]=A[2]*Math.cos(w[2]*t + phi[2]);

52         CE[1].CorpoExtenso.position.set(x[1],y[1],0);
53         CE[2].CorpoExtenso.position.set(x[2],y[2],0);
```

Figura 2.14: Equações de evolução do MHS.

As equações de evolução devem vir dentro da *function Simulacao()*. Os valores de *x[1]* e *x[2]* mudarão a cada quadro ou passo da simulação, lembrando que a cada passo todos os comandos presentes na função *Ambiente* serão executados. *Math* é um objeto predefinido do *javascript* que pode ser acessado sem a necessidade do uso do *new*. *Math.cos()* vai calcular o seno em radianos do argumento entre parêntesis. As equações são idênticas, o que indica que os dois corpos estão sujeitos às mesmas regras matemáticas provenientes das leis físicas que permitiram chegar a Equação 2.3.3. A diferença ficará a cargo dos valores de A[1], A[2], ω[1], ω[2], *phi[1]* e *phi[2]*, que é justamente o objetivo desta simulação, mostrar o que ocorre com o movimento do bloco quando alteramos algumas propriedades do sistema o que permitirá fazer comparações. É importante ficar atento ao uso das funções da AcliveJS. Se considerar a função *CorpoExtenso*, para criar um objeto que seja do tipo 'CorpoExtenso' é necessário o uso do *new* e dos argumentos que devem ser repassados ao objeto.

```
CE[1] = new CorpoExtenso(2,2,0.5,x[1],y[1],0,0,0,0,'rgb(255,0,0)',false,0);
```

A linha acima deve ser interpretada da seguinte forma pelo usuário: "vou criar um objeto chamado *CE[1]* que é do tipo *CorpoExtenso* e que possui dimensões (2,2,0.5), na posição (x[1],y[1],0), orientação espacial (0,0,0), na cor vermelho ('rgb(255,0,0)'), com visualização no formato 'arame' falso e com um material básico (0)."

A princípio pode parecer complicado, mas com o tempo acostuma-se a pensar desta maneira. Após o objeto ser criado no *Ambiente()*, é possível acessar suas propriedades e os seus métodos em *Simulacao()*. O javascript é uma linguagem orientada a objeto. Linguagens de Programação Orientada a Objeto (POO) foram criadas para que o programador ao desenvolver aplicativos pensasse nas soluções dos problemas como na vida real, ou seja, como se estivesse manipulando objetos. Todos os objetos possuem comportamentos (métodos) e atributos (propriedades). Um carro é um objeto, ele possui alguns atributos, como cor, tem rodas, volante, entre outros. Mas além dos atributos, possui comportamento, como acelerar, acender faróis ou virar o volante. Quando o objeto CE foi criado através da declaração *new*, seus atributos foram definidos quando a *function* presente no arquivo *aclive.js* foi criada. As *functions* desenvolvidas para este projeto funcionam como "gabaritos", ou *blueprints*, do termo em inglês, para a criação dos objetos. Os atributos são dados aos objetos no momento de

sua criação através da passagem de parâmetros, como sua cor. Compreendido a Programação Orientada a Objetos, POO, é possível interpretar as próximas linhas de *Simulacao()*:

CE[1].CorpoExtenso.position.set(x[1],y[1],0);

A linha de código acima deve ser interpretada como: "O objeto *CE[1]* do tipo *CorpoExtenso* terá sua posição ajustada para x=x[1], y=y[1] e z[1]=0". A posição de cada *CE* será atualizada através desta chamada. A Figura 2.15 mostra um instante da simulação com os valores iniciais adotados.

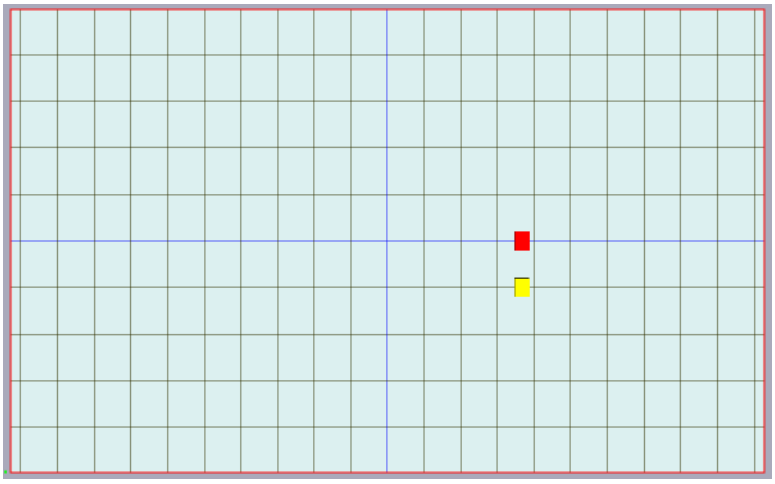


Figura 2.15: MHS. Frequências angulares iguais.

Alterando ω[2] para metade do valor de ω[1], quando o corpo 1 completar uma oscilação e corpo 2 ainda estará no meio do caminho. A Figura 2.16 um momento qualquer dessa simulação onde os corpos oscilam com frequências distintas.

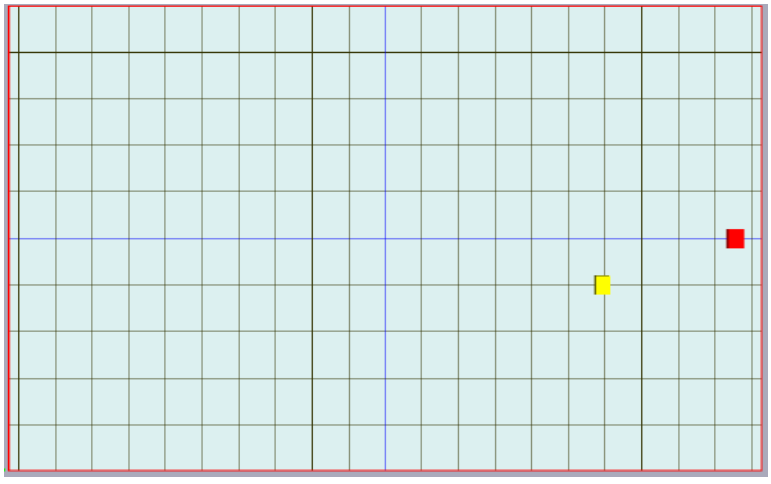


Figura 2.16: MHS. Frequências angulares distintas.

2.3.5 Objetos Complementares

Até aqui a simulação oferece um bom material para que o educador possa explicar o conceito de amplitude e frequência angular do movimento oscilatório alterando o valor da massa e da constante elástica da mola. Mas é possível explorar outros detalhes, como o comportamento do vetor velocidade e aceleração de ambos os blocos partindo da análise das forças envolvidas no sistema. Para isso, é necessário incluir alguns objetos complementares, como vetores. Ao derivar a Equação 2.3.3 encontra-se a velocidade do bloco representada através da Equação 3.4.5 abaixo:

$$v = -A\omega . sen(\omega t + \phi) \qquad (2.3.5)$$

Derivando a Equação 2.3.5 chega-se a Equação 2.3.6, que é a equação de evolução para a aceleração:

$$a = -\omega^2 x \qquad (2.3.6)$$

Com as Equações 2.3.5 e 2.3.6 disponíveis, o próximo passo é criar os objetos vetores que serão ligados a posição de cada bloco com objetivo de exibir as setas para as velocidades e acelerações. Nas linhas 36 e 37 são criados os vetores para a velocidade dos blocos 1 e 2, representados por *vCE[1]* e *vCE[2]*, mnemônicos para 'velocidade do corpo extenso'. As linhas 39 e 40 fazem o mesmo, só que para a aceleração, designados por *aCE[1]* e *aCE[2]*. A Figura 2.17 exibe o trecho do código em *Ambiente()*.

```
35
36          vCE[1]=new Vetor(x[1],y[1],0,0,v[1],0,0.3,'rgb(255,0,0)');
37          vCE[2]=new Vetor(x[2],y[2],0,0,v[2],0,0.3,'rgb(255,0,0)');
38
39          aCE[1]=new Vetor(x[1]-0.2,y[1],0,0,a[1],0,0.15,'rgb(0,0,255)');
40          aCE[2]=new Vetor(x[2]-0.2,y[2],0,0,a[2],0,0.15,'rgb(0,0,255)');
41  }
```

Figura 2.17: Atualizações dos vetores em *Simulacao()*.

vCE é a variável que armazenará o objeto vetor velocidade e *aCE*, o vetor aceleração, ambos criados dentro da *function Ambiente()*. Os parâmetros a serem repassados podem ser consultados na folha *vetor.html*. Basicamente os três primeiros parâmetros são as coordenadas especiais da posição da origem do vetor, os três seguintes referem-se à extremidade do vetor, o fator de escala e finalmente a cor do

vetor. A Figura 2.18 mostra a execução tomada num instante qualquer onde aparece o vetor velocidade, em vermelho, e aceleração, em azul, de ambos os blocos.

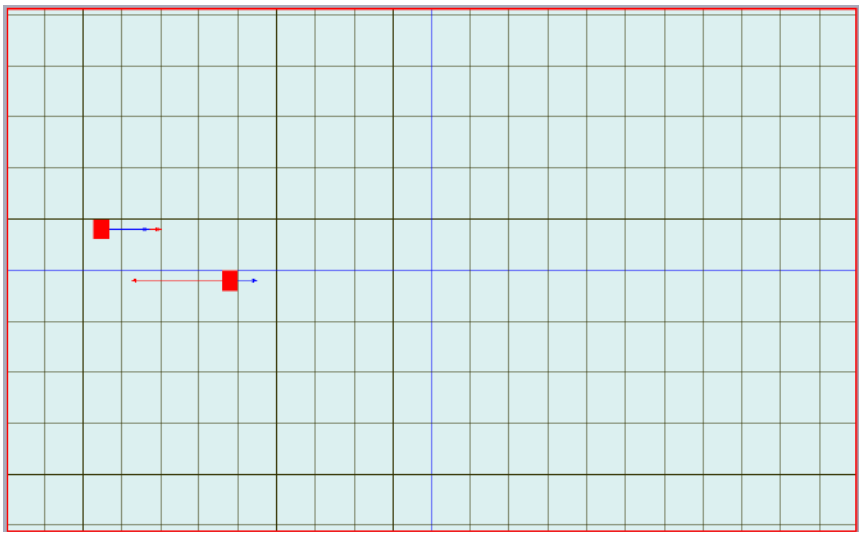


Figura 2.18: Vetor velocidade e aceleração dos blocos 1 e 2.

Para esta simulação foi utilizada apenas a janela de renderização do *WebGL* através do comando *Janela3D*. Não foi habilitado o *canvas* do *HTML5* para exibição de textos e dos valores das grandezas para análise. Tampouco foram exploradas entradas via teclado ou mouse utilizando os *inputs* do *HTML5* por meio de edição do arquivo *main.html*. Trabalhada desta forma a AcliveJS serve como uma ferramenta de apresentação através do método expressivo, geralmente por meio de uma aula expositiva. Uma simulação desta natureza, ajuda a quebrar a rotina do meio estático do quadro de escrever para fazer uso do movimento dos objetos na tela do computador. A AcliveJS permite a importação de figuras, tornado a simulação mais atraente e chamativa. Entretanto, este recurso não foi explorado neste modelo. Como exemplo, seria possível criar um plano com a figura de uma mola com fundo transparente e fazer este plano aumentar e diminuir em comprimento de acordo com a posição do corpo, simulando uma mola esticando e contraindo obedecendo a Lei de Hooke. No Apêndice 2 encontra-se a listagem completa do o exemplo explorado neste capítulo.

Convém recordar que a mesma simulação, se feita utilizando as funções do *Threejs*, teriam uma quantidade de linhas superiores ao mostrado no Apêndice 2. Essa é uma das vantagens do uso da AcliveJS em detrimento do uso direto do *WebGL* e do *Threejs*. Uma das desvantagens que o educador encontrará ao utilizar este método é que sempre terá que retornar ao código para modificar os parâmetros do modelo caso deseje

mostrar uma nova situação aos seus estudantes. Não é necessário fechar o navegador, a alteração pode ser feita diretamente no IDE e após salvar basta atualizar o *browser* que a simulação vai reiniciar. Mesmo assim, este procedimento pode se tornar uma boa prática para mostrar os estudantes como criar uma simulação computacional utilizando a AcliveJS e, inclusive, servir como atividade extraclasse estimulando os estudantes a desenvolverem suas próprias simulações a partir de um modelo físico proposto, ocasionando em uma atividade exploratória e tornando a simulação muito mais atraente.

3 VISITANDO A PÁGINA DO PROJETO

Para fazer o download da AcliveJS 0.10 é necessário acessar o seguinte endereço eletrônico: https://github.com/AcliveJS/Aclive

Outra opção consiste em ingressar na página do GitHub (https://github.com/) e no campo '*Search GitHub*' ao lado do botão '*Sign In*' digitar *Aclive*. A Figura 3.1 mostra uma captura de tela da página do GitHub.

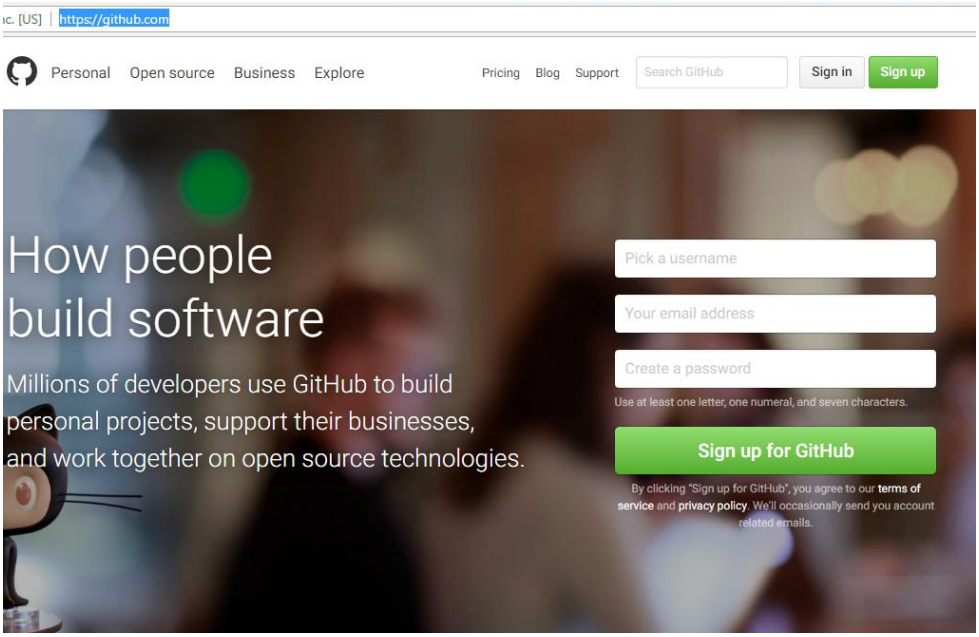


Figura 3.1: Página do GitHub.

A Figura 3.2 exibe o repositório onde a AcliveJS pode ser baixada.

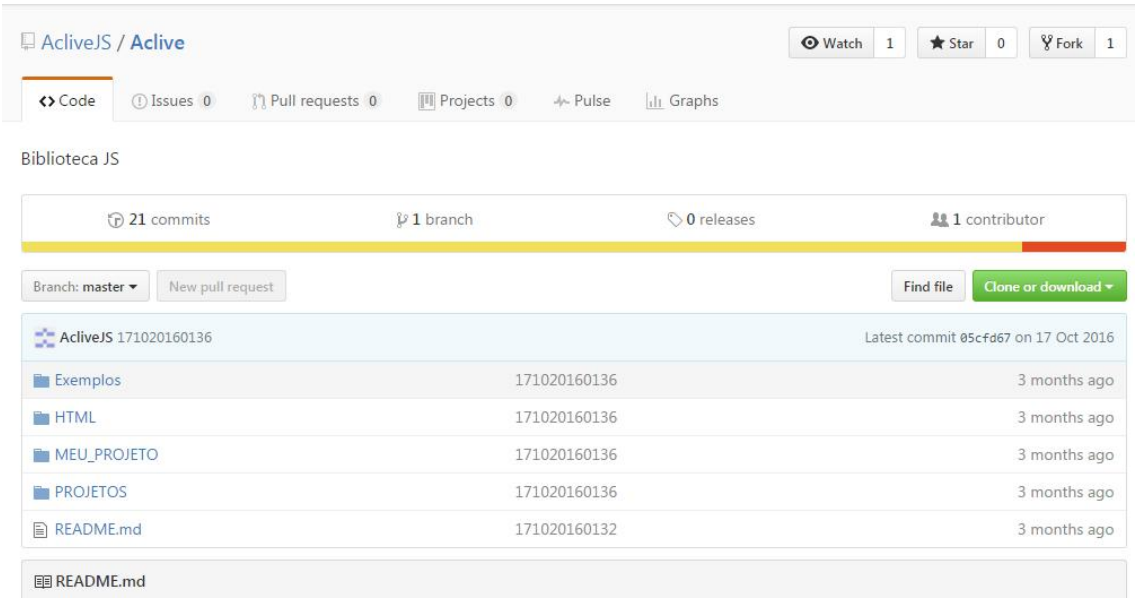


Figura 3.2: Página da AcliveJS no GitHub.

Para uma melhor compreensão do princípio de funcionamento da AcliveJS é importante executar os exemplos, contidos no diretório EXEMPLOS, na seguinte ordem:

- Janela3D
- Janela2D
- Sistema de coordenadas
- CamOrbita
- Grade xy-xz-yz
- Particulas
- Texto2D
- Corpo Extenso
- Vetores
- MHS
- Componentes
- Referencial Local
- Demo
- MR1D (Movimento Relativo em 1 dimensão)*
- MR2D (Movimento Relativo em 2 dimensões)*

3.1 Arquivo LEIA-ME PRIMEIRO.txt

Após baixar a AcliveJS é importante compreender os passos necessários para iniciar uma simulação. Daí a importância do arquivo LEIA-ME PRIMEIRO. A Figura 3.3 exibe um *screenshot* retirado desse arquivo.

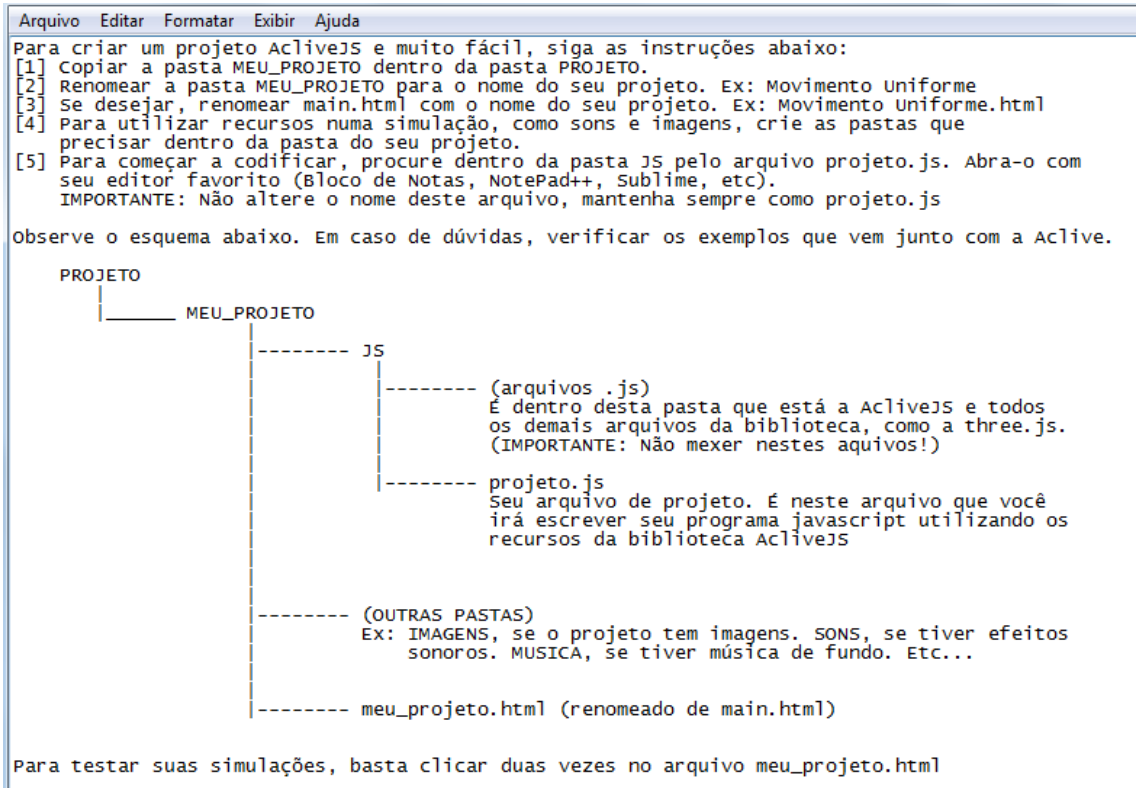


Figura 3.3: Conteúdo do arquivo LEIA-ME PRIMEIRO.txt.

Apesar de não explicitado no LEIA-ME PRIMEIRO.txt é possível ter mais de uma simulação dentro do mesmo diretório ou projeto. Entretanto será necessário modificar o nome do arquivo *projeto.js* para o nome da(s) simulação(ões) presentes na pasta '*js*'. Para uma melhor compreensão, imagine que o educador deseje uma sequência de três simulações sobre o Movimento Harmônico Simples, cada uma delas dando ênfase a um determinado tópico sobre o assunto. Após copiar o diretório MEU_PROJETO para a pasta PROJETO, ele renomeará MEU_PROJETO para MHS. Dentro de MHS possui a pasta '*js*' onde existe o arquivo de trabalho *projeto.js*. O educador pode renomear este arquivo para *MHS1.js* e criar outros dois, por exemplo, *MHS2.js* e *MHS3.js*. Entretanto, do jeito que está ainda não vai funcionar, pois a execução da simulação é realizada através do arquivo *main.html*, por intermédio de um arquivo denominado *projeto.js* e não *MHS1.js*. O usuário deverá editar o arquivo *main.html* alterando o nome *projeto.js* para *MHS1.js*.

A Figura 3.4 mostra as chamadas feitas por padrão pelo arquivo *main.html.*

```
<!-- Meus arquivos .js -->
<script src="js/Aclive.js"></script>
<script src="js/projeto.js"></script>
```

Figura 3.4: Chamada padrão da biblioteca Aclive e do arquivo projeto.

A Figura 3.5 mostra como devem ser realizadas as alterações no arquivo *main.html* caso o usuário deseje executar mais de uma simulação para o mesmo arquivo '*main*'.

```
<!-- Meus arquivos .js -->
<script src="js/Aclive.js"></script>
<script src="js/MHS1.js"></script>
<script src="js/MHS2.js"></script>
<script src="js/MHS3.js"></script>
```

Figura 3.5: Alterações para o *main.html* importar mais de uma simulação.

O arquivo *main.html* pode ser renomeado sem problemas, por exemplo, *EstudoMHS.html*.

4 O AMBIENTE DE DESENVOLVIMENTO INTEGRADO (IDE)

Um Ambiente de Desenvolvimento Integrado, ou IDE, do inglês *Integrated Development Enviromment*. é um software que tem a finalidade de facilitar o trabalho dos programadores em suas tarefas de escrita e edição de códigos. Qualquer IDE que seja capaz de operar os arquivos com extensão do javascript (*js*) servirá de ambiente de trabalho para o desenvolvimento de simulações utilizando a AcliveJS. O usuário poderá utilizar o Bloco de Notas do Windows para criar suas simulações, entretanto, existem IDEs disponíveis na internet para esta finalidade e com muito mais recursos, como por exemplo, o NotePad++, o Sublime e o ConText.

4.1 NotePad++

O NotePad++ (*https://notepad-plusplus.org/*) é um IDE gratuito e que possui suporte a diversas linguagens de programação, incluindo *javascript*. A Figura 4.1 mostra a captura de tela de um fragmento de um código AcliveJS escrito no NotePad++.

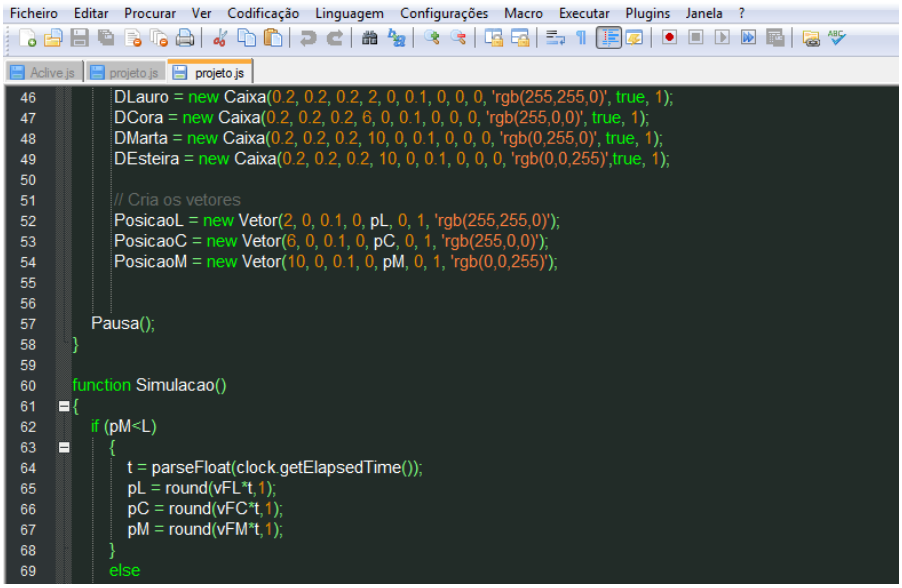


Figura 4.1: Visão do IDE NotePad++

Suas principais características são: *Syntax Hightlight*, *Syntax Hightlight* definida pelo usuário, autocompletar, multi-documentos através de tarjas (*tab*), *multi-view*, *zoom in* e *zoom out*, ambiente com suporte a várias linguagens de programação.

4.2 *Sublime*

A Sublime (*https://www.sublimetext.com/*) é um IDE pago, entretanto, sua versão gratuita oferece os recursos necessários para a construção dos projetos com a AcliveJS, através da programação javascript. A Figura 4.2 mostra a captura de tela de um fragmento de código AcliveJS digitado na Sublime.

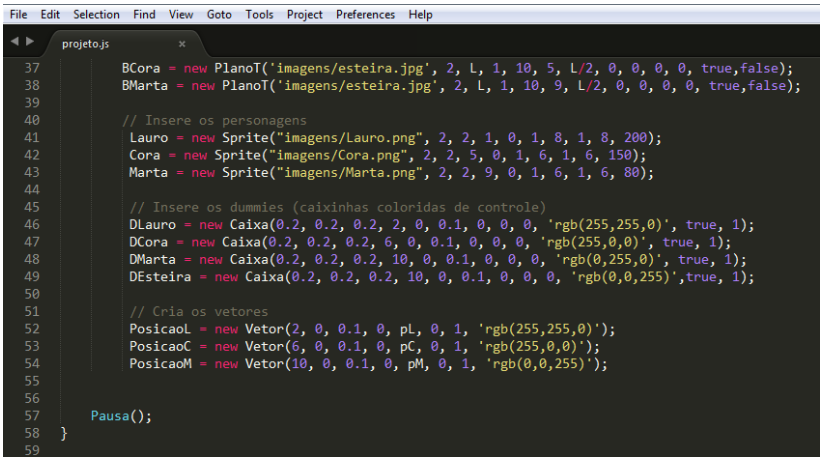


Figura 4.2: Visão do IDE Sublime.

As principais características da sublime são: possibilidade de uso de múltiplas seleções, sugestões de comandos, *Syntax Hightlight*, múltiplos documentos abertos por meio de tarjas, *zoom in* e *zoom out*, ambiente com suporte a várias linguagens de programação.

4.3 *ConText*

O editor ConText (*http://www.contexteditor.org/index.php*) representa outra opção gratuita. A Figura 4.3 mostra a captura de tela de um fragmento de código da AcliveJS digitada no ConText.

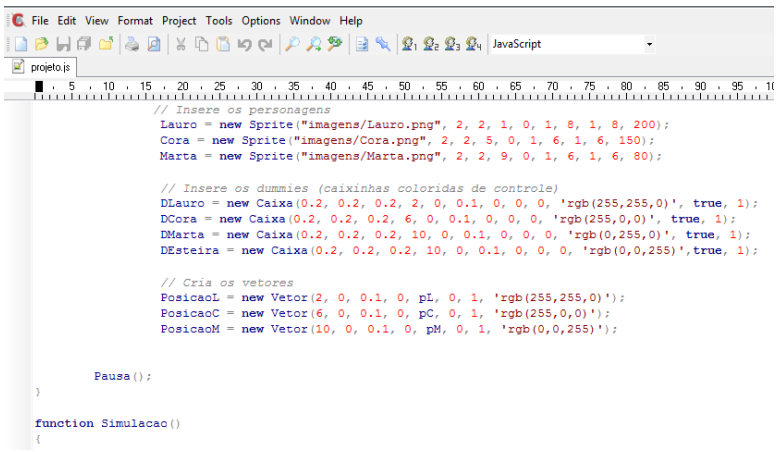


Figura 3.3: Visão do IDE ConText.

As principais características do ConText são: múltiplos arquivos abertos via tarja, arquivos de tamanho ilimitado, *Syntax Hightlight* para diversos tipos de linguagens, suporte a multi-linguagem (incluindo português do Brasil), entre outros.

5 INTRODUÇÃO AO JAVASCRIPT

Neste tópico será feita uma abordagem dos principais comandos do *javascript* tendo em vista que o desenvolvimento de uma simulação será realizada utilizando esta linguagem. Todos os modelos matemáticos desenvolvidos em *projeto.js* devem ser escritos utilizando os comandos da linguagem *javascript* junto com as funções da AcliveJS. As funções básicas para todo projeto com a Aclive são a *Ambiente(),* onde será construido o cenário, e *Simulacao(),* onde as propriedades dos objetos serão alteradas durante a simulação.

```
// variáveis globais
Function Ambiente()
{
    // Código aqui
}
Function Simulacao()
{
    // Código aqui
}
```

O programa deve ser salvo com o nome *projeto.js* dentro da pasta '*js*'. Para executar a simulação basta abri-lo clicando duas vezes no arquivo *main.html*. Não é preciso fechar o arquivo *main.html* para editar o programa. É possível fazer alterações no código com o arquivo aberto no navegador. Após realizar as modificações no código, basta atualizar a página. Caso a simulação não apareça no navegador, é provável que haja algum erro de digitação. Para descobrir o erro é possível utilizar o IDE do próprio navegador.

Javascript é uma linguagem utilizada para o auxílio no desenvolvimento de páginas de internet. Produz páginas interativas ou dinâmicas, com recursos de entrada e controle, como formulários, botões, imagens, exibição de calendários, entre outros recursos. Todo código javascript fica entre as *tags* <*script*> e </*script*>. A menos que você se torne um colaborador da AcliveJS, não vai se preocupar com isso. O código base acima, das funções *Ambiente()* e *Simulação()*, já garante que tudo que for digitado já estão entre as *tags* <*script*>.

5.1 Comentários

Comentários são feitos utilizando duas barras //. Comentários em blocos devem vir entra /* e */.

// Este comentário ocupa uma única linha
/ Este comentário*
Pode ocupar
*Várias linhas */*

5.2 Variáveis

Uma variável contém um valor. Quando o programador utiliza uma variável, está fazendo uso dos dados que ela contém. Variáveis são utilizadas para armazenar, recuperar e manipular dados presentes no código. Todas as variáveis necessitam ser declaradas, ou seja, o programador deve reservar um espaço na memória para armazenar o conteúdo que deseja. Para declarar uma variável é usado a palavra *var.*

Var a;
Var b;

É possível declarar uma variável sem usar a palavra chave *var* na declaração e atribuir um valor a ela. Trata-se de uma declaração implícita:

a= 100;
b = 2.5;

Não é possível utilizar uma variável que não tenha sido declarada. Podemos classificar as variáveis de acordo com a Tabela 5.1:

GLOBAIS	Declaradas (criadas) fora de uma função. As variáveis globais podem ser acessadas em qualquer parte do programa. O *var* é opcional nas variáveis globais, mas obrigatório nas locais.
LOCAIS	Declaradas (criadas) dentro de uma função. Só podem ser utilizadas dentro da função onde foram criadas e precisa ser definida com a instrução *var.*

Tabela 5.1: Tipos de variáveis.

Uma variável importante na simulação de Fenômenos Físicos são as do tipo *array*. Um *array* no javascript é um objeto com um construtor único, com uma sintaxe literal e com um conjunto adicional de propriedades e métodos herdados de um protótipo de Array.

```
Var a=[];
a[0]=”A”;
a[1]=”C”;
a.push(“L”);
a.push(“I”);
```

O código acima mostra o poder do uso de *array* em javascript. Em primeiro lugar, é declarada uma variável *a* que é do tipo *array*, mas esta *array* ainda está vazia. Na linha seguinte a[0]=”A” atribui o caractere “A” a posição 0 (zero) do *array*. Em outras palavras, foi adicionado um dado. A[1]=”C” faz a mesma coisa, só que em outra localidade da memória, indicada pela posição 1 entre os colchetes. Outra forma de incluir dados num *array* é através do método *push*, que vai adicionar um dado sempre no “final da pilha”, ou seja, no local do *array* ainda não preenchido com informações. Assim, o *a.push(“L”)* vai colocar o caractere “L” na posição a[2] do *array*.

5.3 Operadores

A Tabela 5.2 mostra os operadores de atribuição:

Operador	Significado
=	Atribuir
+=	Ex: x+=5 (mesmo que x=x+5)
-=	Ex: x-=5 (mesmo que x=x-5)
*=	Ex: x*=5 (mesmo que x=x*5)
/=	Ex: x/=5 (mesmo que x=x/5)
%	Resto

Tabela 5.2: Operadores de Atribuição.

A Tabela 5.3 mostra os operadores relacionais:

Operador	Significado
<	Menor que
>	Maior que
==	Igual
!=	Diferente
>=	Maior ou igual a
<=	Menor ou igual a

Tabela 5.3: Operadores relacionais.

A Tabela 5.4 mostra os operadores lógicos:

Operador	Significado
&&	E lógico
\|\| | OU lógico |

Tabela 5.4: Operadores Lógicos.

5.4 Expressões Condicionais

São comandos que permitem mudança no fluxo de execução do programa. Geralmente presente no núcleo da maioria das linguagens de programação, diferenciando muito pouco entre uma e outra.

5.4.1 If / Else/ If Else (encadeado)

comando	Exemplo de uso
IF	**If (b==6)** **{** **// Código aqui** **}**
IF / ELSE	**If (b==6)** **{** **// código aqui** **}** **else** **{** **// código aqui** **}**
IF / ELSE (encadeado)	**if (b==6)** **{** **// código aqui** **}** **else if (b==10)** **{** **// código aqui** **}** **else** **{** **// código aqui** **}**

Tabela 5.5: Expressões condicionais.

5.4.2 Switch

Este comando substitui o IF / ELSE encadeado. Observe os dois exemplos mostrados na Tabela 5.6.

Exemplo 1	Exemplo 2
f=”amarelo”; **switch (f)** **{** **Case “vermelho”:** **// código aqui** **Break;** **Case “amarelo”:** **// código aqui** **Break;** **Case “verde”:** **// código aqui** **Break;** **Default:** **// código aqui** **}**	**Letra = “e”;** **Switch(letra)** **{** **Case “a”:** **Case “e”:** **Case “i”:** **Case “o”:** **Case “u”:** **// código aqui** **Break;** **Default:** **// código aqui** **}**

Tabela 5.6: Switch

5.5 Estruturas de Repetição

São também conhecidos como Estruturas de Interação ou Loop, esses comandos mantêm a execução até que o seu argumento seja falso. Ou seja, permite ao programador executar um determinado bloco de código um determinado número de vezes. Existem duas maneiras de interromper uma estrutura de repetição, a primeiro é quando a condição de execução da estrutura é alcançada. Neste caso, a execução natural da estrutura é suficiente para que isto aconteça. No segundo caso, é possível interromper através da instrução break, o que finaliza o laço imediatamente (verifique o exemplo presente no tópico 5.4.2). É possível também “escapar” de um bloco de códigos dentro da estrutura de repetição através do comando *continue*.

5.5.1 For / While / Do While

FOR	Sintaxe: for(inicio; condição; incremento){ } Ex: **Var A = 2;** **Var i** **For(i=0; i<2;i++)** **{** **A=i;** **}**
WHILE	**Numero = 0;** **While(numero<10)** **{** **Numero++** **}**
DO WHILE	**Numero = 0;** **Do** **{** **Numero++** **}** **While(numero<18)** **(...continuação do código)**

Tabela 5.7: Estruturas de Repetição.

5.6 With

Quando é preciso manipular propriedades ou métodos de um mesmo objeto repetidas vezes, o nome do objeto deverá ser digitado sempre que estas propriedades ou métodos forem referenciados. O with evita exatamente que isto ocorra simplificando o processo de digitação.

SINTAXE	Exemplo
With(<objeto>) **{** **// código aqui** **}**	**With(Math)** **{** **A=PI;** **B=ABS(x);** **C=e;** **}** O código acima sem o uso do With: A = Math.PI; B = Math. ABS(x); C = Math.e;

Tabela 5.8: With

5.7 Functions

Uma função é um procedimento em javascript, ou seja, um conjunto de instruções que executa uma tarefa ou calcula um valor. Para utilizar uma função é preciso defini-la em algum lugar, seja no próprio bloco do programa que está desenvolvendo ou num arquivo *javascript* separado. Definir uma função é o mesmo que declará-la. Para declarar uma função é preciso a palavra-chave *function*, a lista de argumentos entre parênteses e separados por vírgulas e as declarações javascript que definem a função entre o par de chaves.

Sintaxe:

```
Function Nome_da_função([parâmetro1],...,[parâmetroN])
{
        // código aqui
        [return(valor_de_retorno)]
}
```

A chamada da função será da seguinte forma:

```
Nome_da_função([parâmetros])
```

A AcliveJS é constituída por um conjunto de funções que foram desenvolvidas, cada uma delas, com um propósito específico. Quando o usuário utiliza um 'comando' da AcliveJS está na verdade fazendo uma chamada de função que foi escrita no arquivo *aclive.js*.

5.8 Criando Objetos

Trabalhar com objetos é a única forma de manipular *arrays*. O próprio *array*, como foi visto, é um objeto no *javascript*. O tópico 5.7 mostra como declarar uma função. É possível utilizar o que foi aprendido para criar um objeto. Imagine que seja preciso fazer uma lista de clientes. O objeto seria definido a partir de uma função como segue:

```
Function Cliente(nome, endereco, telegone,renda)
{
        This.nome = nome;
        This.endereco = endereco;
        This.telefone = telefone;
        This.renda = renda;
}
```

A propriedade 'this' especifica o objeto atual como sendo fonte dos valores passados para a função. Agora, basta criar o objeto:

```
Maria = new Cliente("Maria", "Rua Tal", "123", "456");
```

Para acessar as propriedades do objeto Maria, basta usar:

Maria.nome – retorna "Maria"

Maria.endereco – retorna "Rua Tal"

Maria.telefone – retorna "123"

Maria.renda – retorna "456"

5.9 Hierarquia do Objeto

O objeto Maria que foi criado no tópico 5.8 possui apenas propriedades. Entretanto, objetos podem possuir propriedades e métodos. O exemplo a seguir mostra a diferença entre uma propriedade e um método de um objeto:

```
Function Cliente(nome, endereço, telefone, renda)
{
        // this.propriedade = "isto é uma propriedade"
        This.nome = nome;
        This.endereco = endereco;
        This.telefone = telefone;
        This.renda = renda;

        /* this.metodo = function ()
          {
                Return "isto é um método";
         } */

        This.despesa = function (renda)
        {
                Return (0.60*renda);
        }
}
```

Propriedades é uma característica do objeto, como o nome que o objeto possui. Método é um comportamento do objeto e geralmente retorna algum resultado ou realiza alguma tarefa quando chamado. Na listagem acima, o método 'despesa' calcula a despesa mensal do objeto de acordo com sua renda.

6 COMANDOS DA ACLIVEJS

A melhor forma de aprender sobre a AcliveJS é através de exemplos. A versão 0.10, dispõe de 15 programas prontos que ilustram seu uso e que podem ser acessados no diretório EXEMPLOS. O objetivo desde capítulo é permitir que o usuário inicie os estudos das funções da AcliveJS e reforce os comandos do *javascript* aprendidos no capítulo anterior. Foram explorados alguns aspectos fundamentais das simulações em tempo de execução e que servem de base para a compreensão, planejamento e concepção de modelos futuros permitindo criar simulações interativas com a AcliveJS. O arquivo *readme.md* reforça a importância de se seguir uma sequência ao executar os exemplos e estudar os códigos, pois trazem uma ordem crescente de complexidade o que facilita os estudos. Este documento explorará alguns destes exemplos obedecendo a ordem sugerida pelo *readme*.

6.1 Janela 3D

Este programa explica como criar uma aplicação básica com a AcliveJS. As *functions Ambiente()* e *Simulacao()* são obrigatórias em todos os aplicativos desenvolvidos com a AcliveJS. Os programas devem ter em *Ambiente()* pelo menos dois comandos: *Janela3D* e *InserirObservador*, caso contrário não funcionará.

A sintaxe para o comando *Janela3D* é: *Janela3D (cor, %largura, %altura)*

Janela3D vai criar uma janela no navegador com uma cor de fundo definido pelo parâmetro cor, com uma porcentagem relativa ao tamanho da janela do *browser* tanto na horizontal como na vertical.

A Figura 6.1 mostra o código do Exemplo *Janela3D*. Observe que este código está recheado de comentários que explica praticamente cada linha do programa o que ajuda num primeiro contato com a AcliveJS.

```
 8 function Ambiente()
 9 {
10         // CondiÃ§Ãµes iniciais aqui
11         Janela3D('rgb(40,40,100)',0.95,0.95);
12         InserirObservador(45,0.1,20000);
13
14
15         // (Códigos de montagem da simulação)
16 }
17
18 function Simulacao()
19 {
20         // Códigos da simulação aqui
21 }
```

Figura 6.1: código do Exemplo *Janela3D*.

Se todos os comentários forem retirados, este programa irá se resumir em dois comandos apenas, a exceção das duas *functions*, que são obrigatórias.

InserirObservador é o comando na AcliveJS que cria uma câmera 3D na cena e que mostra a região que esteja dentro do campo de visão. Os parâmetros são abertura do campo de visão, corte para perto e corte para longe. Para a maioria das aplicações, 45º de abertura é o suficiente. Valores elevados criam o efeito conhecido como "olho de peixe". Para corte de perto, 0.1 é um bom valor e para corte de longe, valores elevados, como 10000 ou 20000 serão suficientes. Neste exemplo, utilizamos os valores citados neste parágrafo. É importante notar para que uma simulação funcione, são necessários no mínimo estes dois comandos, *Janela3D* e *InserirObservador*. A concepção por trás disso está na ideia de que o programador vai criar um 'universo' onde desenvolverá um modelo para reproduzir um fenômeno. A primeira coisa a ser feita é 'criar o espaço' em si, e isto ocorre através do comando *Janela3D* (Nas versões anteriores, o nome deste comando era *CriarEspaco*), mas um espaço sem um observador não faz muito sentido. Por isso precisamos da figura do observador na cena, o que no universo da programação 3D é o mesmo que inserir um objeto câmera.

A Figura 6.2 mostra o programa em execução:

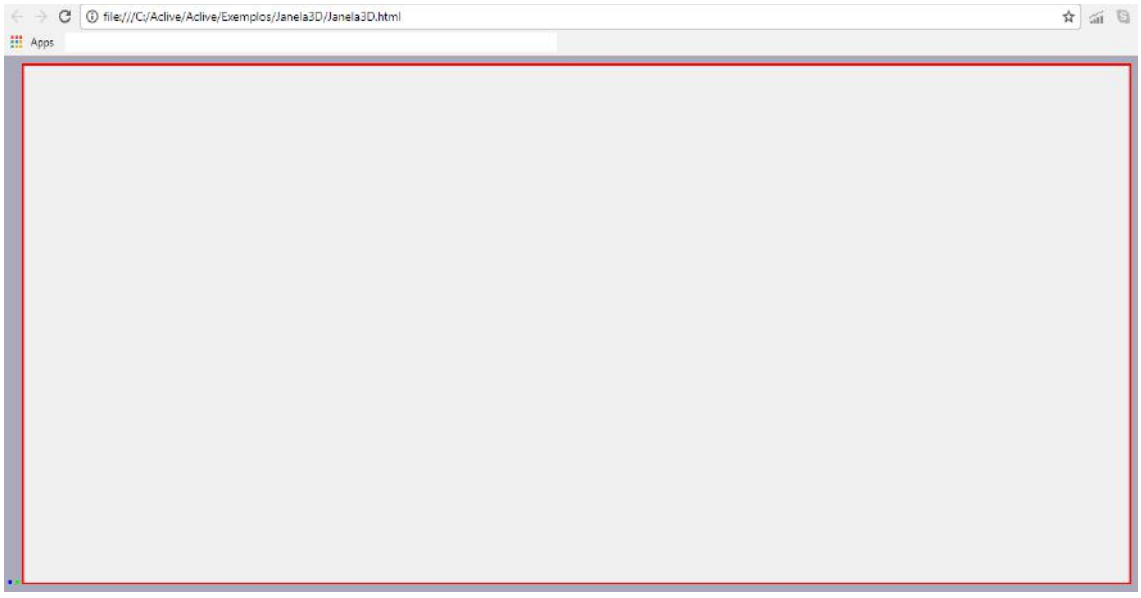


Figura 6.2: Exemplo *Janela3D*.

Este programa não tem muita graça, pois ele apenas cria um ambiente com um fundo cinza.

6.2 Janela 2D

Este exemplo é bastante parecido com o anterior. Agora além da janela 3D foi criada uma janela 2D.

A sintaxe para o comando *Janela3D* é: *Janela2D (cor, %largura, %altura)*

Esta janela terá borda azul, por se referir ao primeiro ponto. A Figura 6.3 mostra o código do exemplo *Janela2D*.

```
8  function Ambiente()
9  {
10         // CondiÃ§Ãµes iniciais aqui
11         Janela3D('rgb(100,40,40)',0.75,0.95);
12         InserirObservador(45,0.1,20000);
13
14         Janela2D('rgb(40,40,100)',0.2,0.95);
15
16 }
17
18 function Simulacao()
19 {
20         // CÃ³digos da simulaÃ§Ã£o aqui
21 }
```

Figura 6.3: Código do exemplo *Janela2D*.

Note que foi reduzida a largura em porcentagem da janela 3D para que a 2D coubesse ao lado. No exemplo *Janela3D* este valor era de 0.95 (com máximo em 1.0) e agora passou para 0.75, ou seja, a janela 3D (de borda vermelha) ocupa 75% do tamanho total enquanto a janela 2D (de borda azul) ocupa apenas 20%. Se os valores ultrapassagem os 100% da largura total do navegador uma janela ficará abaixo da outra e o navegador dará acesso à rolagem de tela no lado direito. A Figura 6.4 mostra o programa em execução:

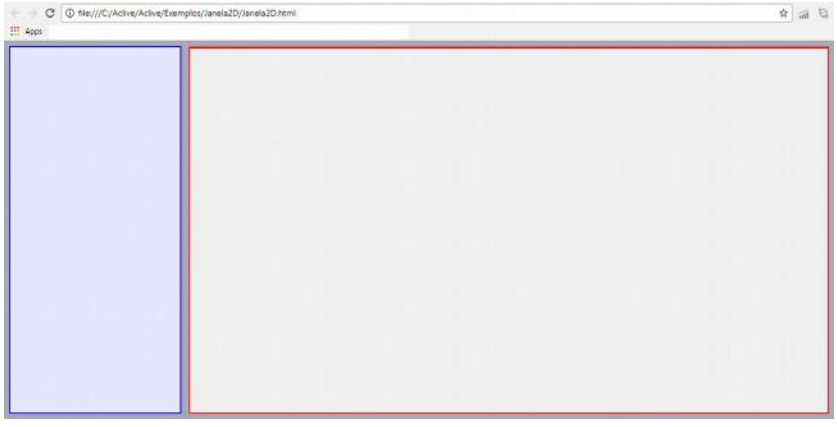


Figura 6.4: Exemplo *Janela2D* em execução no Google Chrome.

6.3 *Sistema de Coordenadas 3D*

Este exemplo segue a mesma linha dos anteriores, a diferença é que será inserido um elemento na cena: um sistema de coordenadas cartesiano.

A sintaxe para este objeto é*: SistemaRetangular(tamanho, boolX, boolY, boolZ)*

Este comando criará três eixos ortogonais entre si, um vermelho, outro verde e um azul. O vermelho corresponde ao eixo X, o verde ao Y e o azul ao Z. Os três últimos parâmetros se ajustado para *true* (verdadeiro) exibirá a parte negativa do eixo em pontilhado.

Outros dois comandos presentes neste exemplo são *PosicaoDoObservador* e *OlharPara*. O primeiro posiciona o observador na cena e o segundo faz o observador olhar para um ponto específico do ambiente. A Figura 6.5 mostra o código do exemplo Sistema de Coordenadas.

```
2  function Ambiente()
3  {
4          // CondiÃ§Ãµes iniciais aqui
5          Janela3D('rgb(100,40,40)',0.75,0.95);
6          InserirObservador(45,0.1,20000);
7          PosicaoDoObservador(60.0, 60.0, 60.0);
8          OlharPara(0.0, 0.0, 0.0);
9          SistemaRetangular(50,false, false, false);
10
11         Janela2D('rgb(40,40,100)',0.2,0.95);
12
13 }
14
15 function Simulacao()
16 {
17         // CÃ³digos da simulaÃ§Ã£o aqui
18 }
```

Figura 6.5: Código do exemplo Sistema de Coordenadas.

Pela listagem é fácil ver que o observador foi posicionado nas coordenadas x=60.0, y=60.0 e z=60.0, mas está ‘olhando’ para o centro do sistema de coordenadas, isto é x=0, y=0 e z=0.

A Figura 6.6 mostra o programa em execução:

Figura 6.6: Exemplo Sistema de Coordenadas em execução no Google Chrome.

6.4 *CamOrbita*

A Figura 6.7 mostra o código deste exemplo:

```
2  function Ambiente()
3  {
4          // CondiÃ§Ãµes iniciais aqui
5          Janela3D('rgb(100,40,40)',0.75,0.95);
6          InserirObservador(45,0.1,20000);
7          PosicaoDoObservador(60.0, 60.0, 60.0);
8          OlharPara(0.0, 0.0, 0.0);
9          SistemaRetangular(50,false, false, false);
10         CamOrbita();
11
12         Janela2D('rgb(40,40,100)',0.2,0.95);
13
14 }
15
16 function Simulacao()
17 {
18         // CÃ³digos da simulaÃ§Ã£o aqui
19 }
```

Figura 6.7: Código do exemplo *CamOrbita*.

Esta listagem é praticamente igual ao do exemplo anterior. A diferença está no comando *CamOrbita().* Este comando oferece ao usuário a possibilidade de movimentar a câmera utilizando o mouse. Com o botão esquerdo do mouse é possível girar a câmera em torno do ponto central da tela. Com o direito é possível fazer um '*pan*', movendo a

câmera para cima, baixo, esquerda e direita. Girando a roda do mouse acontece o *Zoom In* e *Zoom Out*.

6.5 Grade xy-xz-yz

Este exemplo é parecido com os anteriores. *GradeXY*, *GradeXZ* e *GradeYZ* são objetos da AcliveJS que desenham grades nos planos correspondentes. Ajudam na visualização e servem como escalas.

A sintaxe é: *GradeXY (CorM, CorG, tamanho, divisão, bool)*

Onde *CorM* é a cor da linha central da grade. *CorG* é a cor da grade. Tamanho é o tamanho da grade em uA (unidades Aclive). Divisão seciona a grade pelo número em divisão. *Bool*, se *true*, a grade começará das coordenadas (0,0). Por padrão, é *false*. A Figura 6.8 mostra o código deste exemplo:

```
2  function Ambiente()
3  {
4          // CondiÃ§Ãµes iniciais aqui
5          Janela3D('rgb(100,40,40)',0.75,0.95);
6          InserirObservador(45,0.1,20000);
7          PosicaoDoObservador(60.0, 60.0, 60.0);
8          OlharPara(0.0, 0.0, 0.0);
9          SistemaRetangular(50,false, false, false);
10         GradeXY('rgb(255,255,0)', 'rgb(50,50,0)',100, 5, false)
11         CamOrbita();
12
13         Janela2D('rgb(40,40,100)',0.2,0.95);
14
15 }
16
17 function Simulacao()
18 {
19         // CÃ³digos da simulaÃ§Ã£o aqui
20 }
```

Figura 6.8: Código do exemplo *GradeXY*.

A Figura 6.9 mostra o programa em execução:

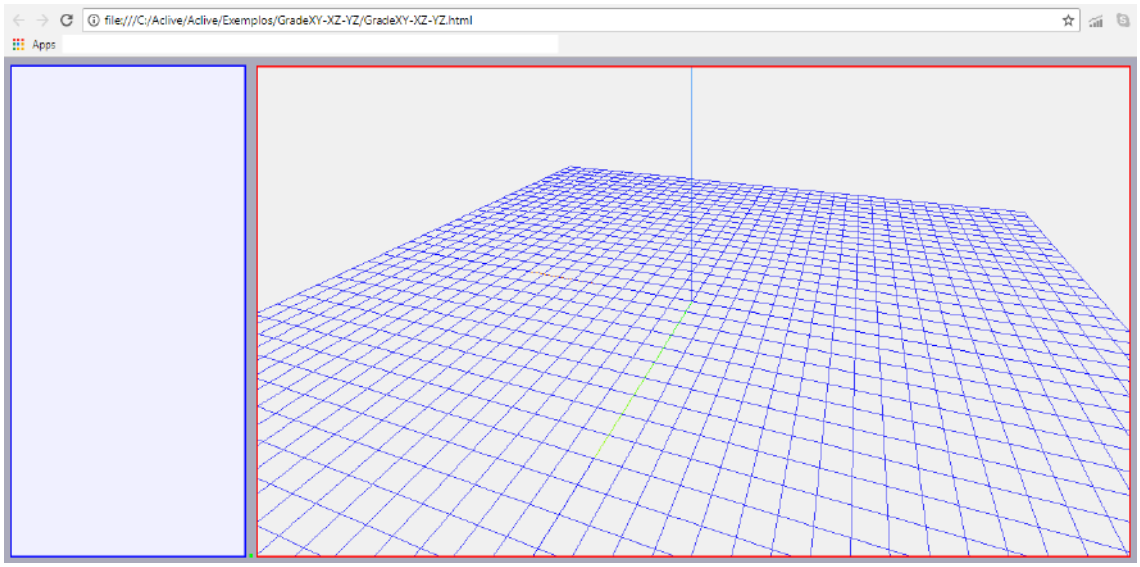


Figura 6.9: *GradeXY* em execução no Google Chrome.

6.6 Partícula

Este exemplo explica como criar uma partícula na AcliveJS. Este é o primeiro exemplo em que variáveis são declaradas fora das *functions Ambiente()* e *Simulacao()*. Estas variáveis armazenarão os objetos *Particula*. A Figura 6.10 mostra o código deste exemplo:

```
var p1, p2, p3, p4;

function Ambiente()
{
        // CondiÃ§Ãµes iniciais aqui
        Janela3D('rgb(100,40,40)',0.75,0.95);
        InserirObservador(45,0.1,20000);
        PosicaoDoObservador(60.0, 60.0, 60.0);
        OlharPara(0.0, 0.0, 0.0);
        SistemaRetangular(50,false, false, false);
        GradeXY('rgb(255,255,0)', 'rgb(50,50,0)',100, 5, false)
        CamOrbita();

        // Cria as partÃ-culas e coloca no ambiente 3D nas posiÃ§Â
        p1 = new Particula(5,5,10);
        p2 = new Particula(15,5,10);
        p3 = new Particula(5,15,10);
        p4 = new Particula(15,15,10);

        Janela2D('rgb(40,40,100)',0.2,0.95);

}
```

Figura 6.10: Código do exemplo *Particula*.

Os números entre parêntesis após a palavra *Particula* são as coordenadas onde ela será criada. A Figura 6.11 mostra o programa em execução:

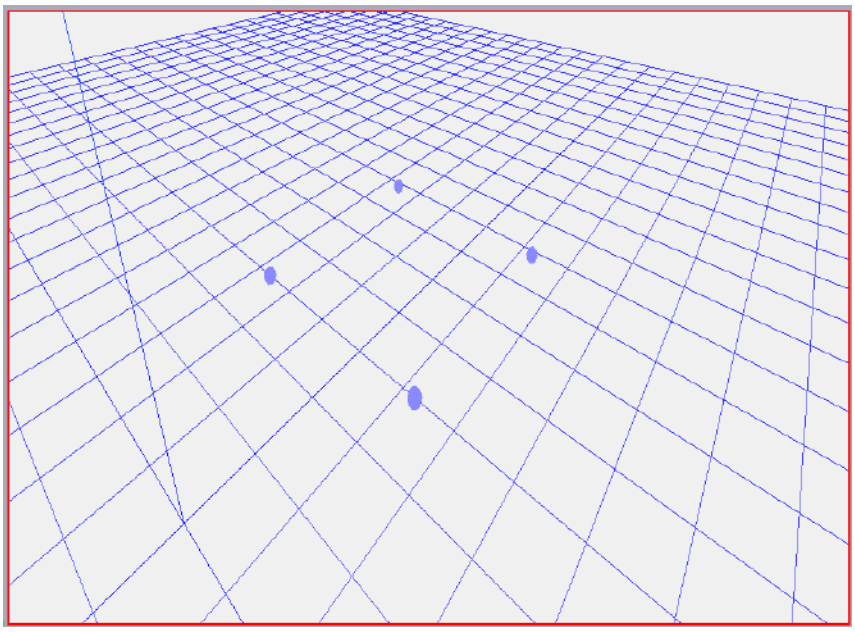


Figura 6.11: *Particula* em execução no Google Chrome.

Dentro do diretório *Particula* há dois exemplos: *ParticulaEx1.html* e *ParticulaEx2.html*. O código acima se refere ao primeiro exemplo onde foi declarado quatro variáveis para quatro partículas. Imagine uma simulação que possua cem partículas, neste caso, torna-se impraticável declarar cem variáveis. Desta forma, torna-se conveniente utilizar *array*, declarando uma variável como uma *array* vazia para armazenar as partículas. A Figura 6.12 mostra parte do código do exemplo dois. Foram omitidos os comandos repetidos nos exemplos anteriores.

```
2  var p=[];
3  var i
4  var r=100
5  var NumPart=1000;
6
7  function Ambiente()
8  {
.
.
.
18         // vamos criar 1000 partÃ-culas aleatoriamente em torno do ponto (0,0,0)
19         for(i=0;i<=NumPart;i++)
20         {
21                 p[i] = new Particula(Math.random()*r-r/2, Math.random()*r-r/2, Math.random()*r-r/2);
22         }
23         // Math.random() Ã© uma funÃ§Ã£o que gera nÃºmero aleatÃ³rios
24         // Ex: Math.random()*10 (vai gerar nÃºmeros aleatÃ³rios entre 0 e 10)
```

Figura 6.12: Código do exemplo *Particula para exibição de 1000 partículas.*

O código acima criará mil partículas e distribuí-las pelo ambiente de forma aleatória em torno da origem. A Figura 6.13 mostra o programa após sua execução:

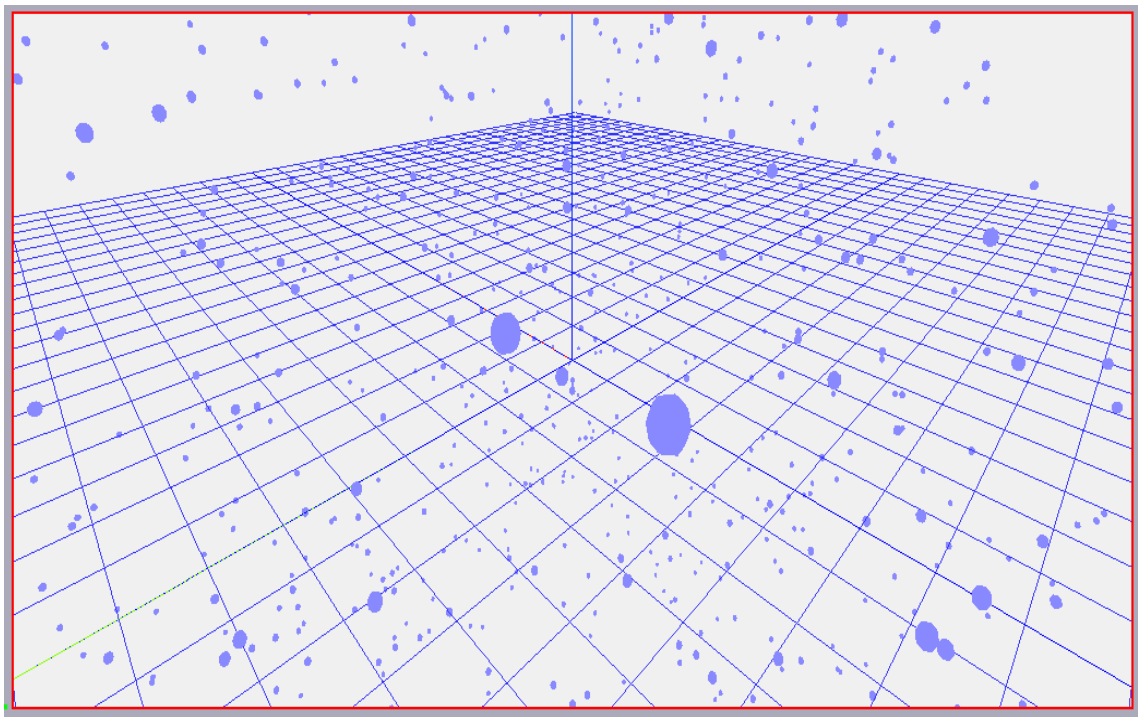


Figura 6.13: *Particula* em execução. Existem 1000 partículas nesta imagem.

6.7 *CorpoExtenso*

Este exemplo é semelhante ao do item 6.6 e explica como criar um objeto *CorpoExtenso* na AcliveJS. Foi considerado para este projeto o 'corpo extenso' como sendo um paralelepípedo com dimensões determinadas pelo usuário. A Figura 6.14 exibe o código:

```
2  var CE;
3
4  function Ambiente()
5  {
6          // CondiÃ§Ãµes iniciais aqui
7          Janela3D('rgb(100,40,40)',0.75,0.95);
8          InserirObservador(45,0.1,20000);
9          PosicaoDoObservador(20.0, 20.0, 20.0);
10         OlharPara(0.0, 0.0, 0.0);
11         GradeXY('rgb(255,255,0)', 'rgb(50,50,0)',100, 5, false);
12         SistemaRetangular(50,false, false, false);
13
14         CamOrbita();
15
16         CE = new CorpoExtenso(2,2,0.5,0,0,0,0,0,0,'rgb(255,0,0)',false,0);
17
18         Janela2D('rgb(40,40,100)',0.2,0.95);
19
20 }
```

Figura 6.14: Código do exemplo *CorpoExtenso*.

Foi criada uma variável denominada *CE, de Corpo Extenso. O comando CorpoExtenso* possui 12 parâmetros: 3 para as dimensões x, y e z; 3 para as coordenadas x, y e z da posição; 3 para rotações dos eixos x, y e z referente a orientação; 1 para cor; 1 *booleana* de exibição em formato cheio ou de arame e 1 para o tipo de material. Foram adotadas as dimensões 2, 2 e 0.5 para comprimento, largura e altura, respectivamente. O objeto encontra-se na origem do sistema de coordenadas e *não está 'girado'*.

6.8 Texto 2D

Este exemplo mostra como inserir objetos na janela 2D. Basicamente um texto com o tradicional "Olá mundo!". É derivado do exemplo anterior, logo possui a mesma listagem, sendo adicionados apenas as linhas que exibirão os textos na *function Ambiente()*. A Figura 6.15 mostra o fragmento de código adicionado.

```
26          Janela2D('rgb(40,40,100)',0.2,0.95);
27          Texto("Ola mundo!", 10, 20, 'rgb(255,255,255)', 14);
28          TextoV("Ola mundo!", 10, 50, 'rgb(255,255,0)',18,1);
```

Figura 6.15: Fragmento de código do exemplo *Texto2D*.

As funções *Texto* e *TextoV* são semelhantes quanto a sintaxe, a diferença está no fato que *TextoV* exibirá as letras vazadas.

A sintaxe é: *Texto (string, x, y, cor, tamanho);*

A Figura 6.16 mostra o programa após a execução.

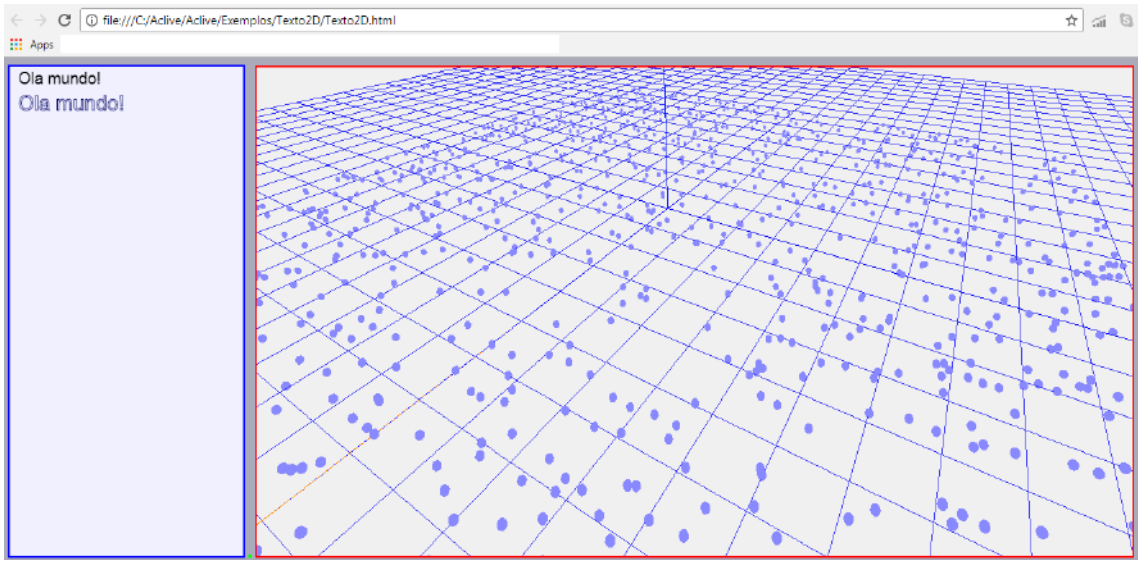


Figura 6.16: Exemplo *Texto2D* "Olá mundo!" em execução.

A captura de tela exibida na Figura 6.16 mostra as mil partículas espalhadas apenas no plano xy. Isto porque todas foram criadas adotando a posição z=0.

6.9 Vetores

O Vetor é um objeto utilizado em simulações que envolvam algum tipo de análise vetorial, com as exibições das setas. No diretório Vetores há três arquivos, *vetores.html*, *vetores2.html* e *vetores3.html*. O primeiro exemplo mostra como criar vetores através do comando *Vetor*. O segundo exemplo faz referência ao vetor posição de um corpo que se move no plano xy tomando como referência a origem do sistema de coordenadas. O último exemplo, mostra um corpo que descreve um MHS onde os vetores posição e velocidade são exibidos durante a simulação. Os dois últimos exemplos, como possui um objeto que se move na cena, foi feito uso de atualizações através da *function Simulacao()*.

A Figura 6.17 mostra um fragmento da listagem do programa do primeiro exemplo:

```
2  var p1, p2, p3, p4, vp1, vp2;
3
4  function Ambiente()
5  {
⋮
15         // Cria as partÃ-culas e coloca no ambiente 3D nas posiÃ§Ãµes x, y, z.
16         p1 = new Particula(5,5,10);
17         p2 = new Particula(15,5,10);
18         p3 = new Particula(5,15,10);
19         p4 = new Particula(15,15,10);
20
21         // Vetor posiÃ§Ãío da origem do sistema de coordenadas atÃ© a particula 1
22         vp1 = new Vetor(0,0,0,5,5,10,1,'rgb(255,255,255)');
23         // Vetor posiÃ§cao da partÃ-cula 2 relativa a particula 1
24         vp2 = new Vetor(5, 5, 10, 15-5, 5-5, 10-10, 1,'rgb(255,0,255)');
25
26         Janela2D('rgb(40,40,100)',0.2,0.95);
27         Texto("Vetores", 10, 20, 'rgb(255,255,255)', 14);
28
29 }
```

Figura 6.17: Código da *function Ambiente()* do exemplo Vetores.

A Sintaxe para vetor é: *Vetor (xo, yo, zo, Ex, Ey, Ez, cor);*

Sendo xo, yo e zo, as coordenadas da origem do vetor. E Ex, Ey e Ez, as coordenadas da extremidade do vetor. O último parâmetro é a cor da seta.

As variáveis *vp1* e *vp2* armazenam os vetores. Note que os valores das coordenadas da extremidade do vetor *vp2* foram colocados como uma diferença do valor final pelo valor inicial. A Figura 6.18 mostra o programa após a execução.

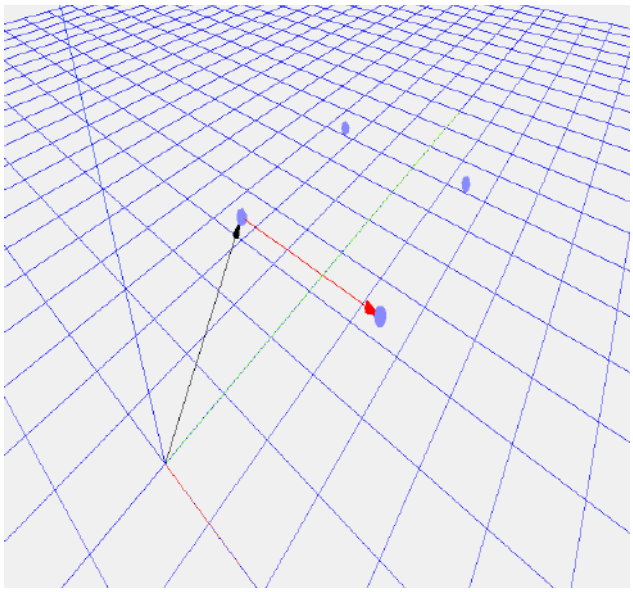


Figura 6.18: Exemplo Vetores em execução no Google Chrome.

A Figura 6.19 mostra um fragmento da listagem do segundo exemplo.

```
2  var CE=[];
3  var w, A, x, y, t, vp;
4
5  function Ambiente()
6  {
7          // CondiÃ§Ãµes iniciais aqui
8          w=2; A=50; x=0; y=0, t=0;
⋮
19         CE[1] = new CorpoExtenso(2,2,0.5,x,0,0,0,0,0,'rgb(255,0,0)',true,0);
20         vp = new Vetor(0,0,0, x,y,0,'rgb(255,255,255)');
21
22         Texto("Vetores", 10, 20, 'rgb(255,255,255)', 15);
23 }

25 function Simulacao()
26 {
27         x=A*Math.sin(w*t);
28         y=(A/2)*Math.sin(2*w*t)
29         CE[1].CorpoExtenso.position.set(x,y,0);
30         t+=0.01;
31
32         //Atualiza o vetor
33         UpdateVetor(vp); vp = new Vetor(0,0,0, x,y,0, 1,'rgb(255,255,255)');
34 }
```

Figura 6.20: Listagem do segundo exemplo de vetores.

Este é o primeiro exemplo que faz uso da *function Simulacao()*. O modelo usado para a movimentação do objeto *CorpoExtenso* é o do MHS em duas dimensões, dadas pelas equações x e y nas duas primeiras linhas do corpo de *Simulacao()*. O interesse é mostrar o vetor posição do corpo. *UpdateVetor* elimina o vetor para em seguida ser criado novamente na nova posição. O terceiro exemplo segue a mesma linha de raciocínio.

6.10 Componentes

Componentes é um objeto da AcliveJS que exibe as linhas pontilhadas referente as componentes ortogonais de um vetor em relação ao sistema de referência. A Figura 6.20 mostra um treco do código que evidencia o uso de componentes.

```
⋮
19          CE[1] = new CorpoExtenso(2,2,0.5,x,0,0,0,0,0,'rgb(255,0,0)',true,0);
20          vp = new Vetor(0,0,0, x,y,0,'rgb(255,255,255)');
21          cvp = new Componentes(x,y,0);
22
23          Texto("Vetores", 10, 20, 'rgb(255,255,255)', 15);
24  }
25
26  function Simulacao()
27  {
28          x=A*Math.sin(w*t);
29          y=(A/2)*Math.sin(2*w*t)
30          CE[1].CorpoExtenso.position.set(x,y,0);
31          t+=0.01;
32
33          //Atualiza o vetor
34          UpdateVetor(vp); vp = new Vetor(0,0,0, x,y,0, 1,'rgb(255,255,255)');
35          UpdateComponentes(cvp); cvp = new Componentes(x,y,0);
36
37  }
```

Figura 6.20: Listagem do programa Componentes.

UpdateComponentes faz a mesma coisa que *UpdateVetor*, ou seja, a AcliveJS trabalha atualmente como um processo de atualização de destruição / construção de entidades. *UpdateComponentes* destrói as componentes e logo em seguida, na mesma linha, ela é recriada utilizando o mesmo método usado em *Ambiente()*.

6.11 Referencial Local

O exemplo 6.3 mostrou como exibir o sistema de referência da AcliveJS, tomado como ‘fixo’, situado nas coordenadas (0,0,0) do universo 3D da ferramenta. Deve ficar evidente que a AcliveJS herda o sistema de referência do WebGL e, para a maioria das aplicações, considerar este sistema como sendo fixo não causará problemas. Entretanto, é possível adotar um referencial local situado num ponto qualquer do espaço, que pode estar em repouso ou em movimento e também ligá-lo a algum outro objeto existente na cena. Na Figura 6.21 é exibida a codificação que mostra o uso desse sistema de referência local.

```
2  var x, y, z;
3  var cubo, SLcubo;

5  function Ambiente()
6  {
7          // CondiÃ§Ãµes iniciais aqui
8          x=0; y=0; z=0;
9
⋮                  ⋮
17         Janela2D('rgb(40,40,100)',0.2,0.95);
18         Texto("Referencial Local", 10, 20, 'rgb(255,255,255)', 13);
19
20         cubo = new CorpoExtenso(4,4,4,0,0,0,0,0,0,0xff0000,true,1);
21         SLcubo = new SistemaRetangularL(10);
22         //cubo.CorpoExtenso.add(SLcubo.SistemaRetangularL);
23         Ligar(cubo.CorpoExtenso,SLcubo.SistemaRetangularL);
24
25 }

27 function Simulacao()
28 {
29         cubo.CorpoExtenso.rotation.x += 0.02;
30         cubo.CorpoExtenso.rotation.z += 0.002;
31 }
```

Figura 6.21: Listagem do programa Referencial Local.

Neste exemplo, foi criado um objeto cubo com renderização em *wireframe* (modo arame). *SLcubo* é o objeto *SistemaRetangularL*, de Sistema Retangular Local. O Comando *Ligar* conecta o *SLcubo* ao cubo, deste modo todas as transformações a qual o objeto cubo estiver sujeita o objeto *SLcubo* também estará. É importante ressaltar que o *Ligar* não serve apenas para conectar um sistema de referência a um objeto. Trata-se de um comando genérico, onde é possível conectar qualquer objeto a outro. *Ligar* possui dois parâmetros, o primeiro é o objeto pai e o segundo o objeto filho.

7 FICHEIROS HTML

Ao fazer o download da AcliveJS no conjunto de pastas há disponível uma chamada HTML. Neste diretório existem arquivos de ajuda, incluindo o manual javascript presente neste documento. Possui também um arquivo que apresenta os comandos da AcliveJS desenvolvidos para a versão 0.10 no formato de ficheiros, o *comandos.html*. Para consultas mais rápidas, cada ficheiro foi salvo separadamente como um arquivo HTML. A Figura 7.1 mostra o conteúdo do diretório HTML, o arquivo *comandos.html* e os demais arquivos com nomes das funções AcliveJS.

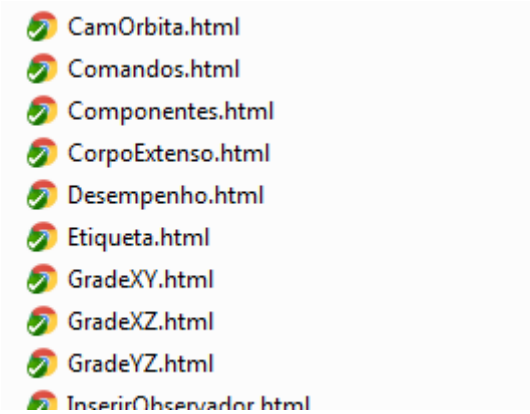


Figura 7.1: Arquivos de ajuda no formato html.

Cada ficheiro serve como um guia para uso de um comando AcliveJS. Nele está contida uma série de informações sobre a função, incluindo os parâmetros que devem ser repassados e exemplo de uso.

Como um guia para consulta, segue os ficheiros com os comandos da AcliveJS em ordem alfabética:

CamOrbita();	
Habilita o controle da câmera para observação do ambiente através do mouse.	
Parâmetros:	
OBSERVAÇÕES: • Pode ser utilizando em Ambiente() e Simulacao() • Com o botão direito pressionado mudamos a posição da câmera. Com o botão esquerdo giramos a câmera e com o botão do meio aproximamos ou afastamos a câmera. • Não irá funcionar direito se o comando OlharPara estiver habilitado.	
EXEMPLO: *CamOrbita();*	

Componentes(x,y,z);	
Exibe as linhas de componentes cartesianas nos eixos x, y e z a partir da posição x,y e z.	
Parâmetros:	X :: componente x
	Y :: componente y
	Z :: componente z
OBSERVAÇÕES: • Pode vir em Ambiente() e em Simulacao();	
EXEMPLO: *Componente (x,y,z);*	

CorpoExtenso(tx, ty, tz, x, y, z, rx, ry, rz, cor, wireframe, material);	
Cria uma particula.	
Parâmetros:	Tx :: dimensão x
	Ty :: dimensão y
	Tz :: dimensão z
	X :: coordenada x da posição do corpo extenso
	Y :: coordenada y da posição do corpo extenso
	Z :: coordenada z da posição do corpo extenso
	Rx :: coordenada X da orientação do corpo no espaço 3D
	Ry :: coordenada Y da orientação do corpo no espaço 3D
	Rz :: coordenada Z da orientação do corpo no espaço 3D
	Cor :: cor do objeto (formato 0xNNNNNN)
	Wireframe :: formato arame (true / false)
	Material :: Tipo de material (1-Básico / 2-Lambert)
OBSERVAÇÕES: • Consideramos que todo o corpo extenso será formado por um cubo ou paralelepípedo. É utilizado para estudo de caso geral, para corpos com formatos diferentes são necessárias outras funções; • Em Rx, Ry e Rz os valores são dados em radianos. Usar Math.PI para PI (π); • Chamada: **Nome = new CorpoExtenso(tx, ty, tz, x,y,z, rx, ry, rz, cor, wireframe, material);**	
EXEMPLO: *Var CE=[];* *CE[1] = new CorpoExtenso(10,10,10, 0, 0, 0, Math.PI*0.5, 0, 0, 0xFF0000, false, 1);*	

Desempenho(fps, MS, mb);	
Este comando permite visualizar algumas estatísticas relacionadas a simulação, como FPS (frame por segundo), o tempo em milisegundos entre cada quadro e a memória utilizada, em Mb, pelo aplicativo.	
Parâmetros:	Fps :: Se true exibe o quadro relativo ao FPS;
	MS :: Se true exibe o quadro relativo ao tempo em milisegundos;
	MB :: Se true exibe a memória utilizada pelo aplicativo;
OBSERVAÇÕES: • Mesmo que habilite apenas um, como FPS por exemplo, no momento da execução se clicar sobre o quadro ele irá alternar entre visualização FPS, MS e MB.	
EXEMPLO: *Desempenho(true, false, false);*	

Etiqueta(mensagem, x, y, z, EL, EA, parâmetros);	
Cria uma etiqueta com texto 2D no ambiente.	
Parâmetros:	mensagem :: texto a ser exibido
	(x,y,z) :: coordenadas da posição da etiqueta
	(EL,EA) :: escala – Largura e Altura
	Parâmetros: **Fontface** – tipo de fonte. Ex.: "Arial" **Fontsize** – tamanho da fonte. **borderThickness** – tamanho da borda **borderColor** – cor da borda {r: xx, g: xx, b: xx, a: xx} **fillColor** – cor de preenchimento {r: xx, g: xx, b: xx, a: xx} **textColor** – cor do texto {r: xx, g: xx, b: xx, a: xx} **radius** - raio **vAlign** – alinhamento vertical. Ex: "Center" **hAlign** – alinhamento horizontal. Ex: "Center"

OBSERVAÇÕES:

- Pode estar em Simulacao() ou Ambiente();
- Os parâmetros devem vir entre chaves {} e separados por virtulas.
  Ex.: {fontface: "Arial", fontsize: 48, borderColor: {r:255, g: 0, b: 0, a: 1}, borderThickness: 4, fillColor: {r:128, g: 128, b: 128, a: 0.5}, textColor: {r: 255, g: 255, b: 255, a: 1}, radius: 0, vAlign: "Center", hAlign: "Center"}) ;
- Parâmetros de cor, como observado no item anterior, devem também vir entre chaves {}. As letras "r", "g" e "b" são de "red" (vermelho), "Green" (verde) e "blue" (azul) e a faixa varia de 0 até 255. A letra "a" vem de "alpha" (transparência) e a faixa vai de "0" (transparente) até "1" (opaco).
- Caso utilize uma etiqueta para exibir variáveis dentro de Simulacao(), então é preciso recriar a etiqueta através do UpdateTexto2D(nome) e em seguida copiando logo abaixo o código da etiqueta criada.

EXEMPLO:

```
UpdateTexto2D(posicaoL.Texto2D);
posicaoL = new Etiqueta( "pL="+round((pL/10.0),1)+"m", 5, pL, 15, 130, 60, {fontface:"Arial", fontsize:48,  borderColor: {r:255, g:0, b:0, a:1.0},  borderThickness:4, fillColor: {r:255, g:0, b:0, a:0.3}, textColor:{r:255, g:255, b:255, a:0.9}, radius:0, Align:"center", hAlign:"center" });
```

GradeXY(CorMeio, CorGrade, tamanho, divisão, bool);	
Este comando exibe uma grade quadrada no plano XY. Útil para visualização das projeções neste plano.	
Parâmetros:	CorMeio :: Cor das linhas centrais da grade
	CorGrade :: Cor da grade.
	Tamanho :: tamanho da grade
	Divisão :: número de subdivisões da grade
	Bool :: Se true a posição (0,0,0) estará no canto da grade. Se false estará no meio da grade.

OBSERVAÇÕES:

- Pode ser utilizado em Ambiente() e Simulacao()
- Formato da cor 'rgb(vermelho, verde, azul)'. Faixa para as cores 0-255.

EXEMPLO:

```
GradeXY('rgb(200,200,200)', 'rgb(255,255,255)', 60, 5, true);
```

GradeXZ(CorMeio, CorGrade, tamanho, divisão, bool);	
Este comando exibe uma grade quadrada no plano XZ. Útil para visualização das projeções neste plano.	
Parâmetros:	CorMeio :: Cor das linhas centrais da grade
	CorGrade :: Cor da grade.
	Tamanho :: tamanho da grade
	Divisão :: número de subdivisões da grade
	Bool :: Se true a posição (0,0,0) estará no canto da grade. Se false estará no meio da grade.

OBSERVAÇÕES:

- Pode ser utilizado em Ambiente() e Simulacao()
- Formato da cor 'rgb(vermelho, verde, azul)'. Faixa para as cores 0-255.

EXEMPLO:

GradeXZ('rgb(200,200,200)', 'rgb(255,255,255)', 60, 5, true);

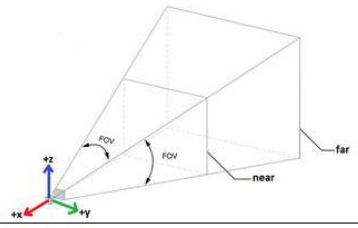
GradeYZ(CorMeio, CorGrade, tamanho, divisão, bool);	
Este comando exibe uma grade quadrada no plano YZ. Útil para visualização das projeções neste plano.	
Parâmetros:	CorMeio :: Cor das linhas centrais da grade
	CorGrade :: Cor da grade.
	Tamanho :: tamanho da grade
	Divisão :: número de subdivisões da grade
	Bool :: Se true a posição (0,0,0) estará no canto da grade. Se false estará no meio da grade.

OBSERVAÇÕES:

- Pode ser utilizado em Ambiente() e Simulacao()
- Formato da cor 'rgb(vermelho, verde, azul)'. Faixa para as cores 0-255.

EXEMPLO:

GradeYZ('rgb(200,200,200)', 'rgb(255,255,255)', 60, 5, true);

InserirObservador(fov,near,far);	
Este comando deve vir depois do comando CriarEspaco. Ele insere o "observador" na cena. Deve virdentro da function Ambiente.	
Parâmetros: 	**Fov** :: ângulo de visão do observador
	Near :: corte de visão próximo
	Far :: corte de visão distante

OBSERVAÇÕES:

- Ângulos de visão superiores a 50 fazem com que a imagem pareça distorcida (efeito lente).

EXEMPLO:

```
Function Ambiente()
{
   CriarEspaco(0x3307ef);
   InserirObservador(45, 0.1, 20000);
}
```

Janela2D(cor, L, A);	
Define uma janela 2D. Deve vir dentro da **function** Ambiente.	
Parâmetros:	**Cor** :: cor de fundo ('rgb(vermelho, verde, azul)
	L :: Largura da janela (entre 0 e 1)
	A :: Altura da janela (entre 0 e 1)

OBSERVAÇÕES:

- Padrão RGB no formato 'rgb(vermelho, verde, azul)' – deve vir entre aspas simples. Os valores para vermelho, verde e azul variam de 0 à 255.
- O exemplo abaixo ainda não irá fazer nada. Para que os programas feitos com a AcliveJS funcionem, são necessários, no mínimo, dois comandos, o Janela3D e o InserirObservador.

EXEMPLO 1:

```
Function Ambiente()
{
  Janela2D('rgb(220,100,50)',0.3, 0.9);
}
```

Janela3D(cor, L, A);	
É o primeiro comando a ser executado em todos os programas. Deve vir dentro da **function** Ambiente.	
Parâmetros:	**Cor** :: cor de fundo (formato 0xNNNNNN)
	L :: Largura do ambiente 3D (entre 0 e 1)
	A :: Altura do ambiente 3D (entre 0 e 1)

OBSERVAÇÕES:

- A cor deve iniciar obrigatoriamente com o prefixo 0x seguido de 6 caracteres que podem variar de 0 a F. Cada par de N em 0xNNNNNN corresponde a uma cor, no caso, os 2 primeiros a cor vermelha, os 2 do meio a cor verde e os dois últimos a cor azul. Um 0x000000 exibirá um ambiente com fundo preto. 0xff0000, um ambiente com fundo vermelho. 0x00ff00, fundo verde. 0x0000ff, fundo azul escuro. 0xffffff, cor de fundo branco.
- Outro padrão de cor aceito é através da string dentro de aspas simples no formato 'rgb(vermelho, verde, azul)'. Onde vermelho, verde e azul variam de 0 até 255.
- O exemplo abaixo ainda não irá fazer nada. Para que os programas feitos com a Aclive funcionem, são necessários, no mínimo, dois comandos, o CriarEspaco e o InserirObservador.

EXEMPLO 1:

```
Function Ambiente()
{
  Janela3D(0x3307ef,1,1);
}
```

EXEMPLO 2:

```
Function Ambiente()
{
  Janela3D('rgb(100,100,100)',1,1);
}
```

Ligar(ObjetoPai, ObjetoFilho);	
Este comando liga um objeto a outro.	
Parâmetros:	ObjetoPai :: objeto principal
	ObjetoFilho :: objeto que estará ligado ao objeto principal

OBSERVAÇÕES:

- Muito útil para ligarmos um sistema de coordenadas local a uma partícula ou corpo extenso.
- Todo o comportamento de translação e rotação executado pelo objeto pai será transmitido ao objeto filho, em outras palavras, o objeto filho está ligado ao objeto pai como que por um "cabo rígido invisível".

EXEMPLO:

```
P1 = new Particula (x,y,z);
S1 = SistemaRetangularL(5);
Ligar(P1.Particula, S1.SistemaRetangularL);
```

Linha2D (x1, y1, x2, y2, espessura, cor, borda);	
Cria uma linha 2D na janela 2D.	
Parâmetros:	(x1,y1) :: coordenadas iniciais
	(x2,y2) :: coordenadas finais
	Espessura :: espessura da linha 2D
	Cor :: cor no formato 'rgb(vermelho, verde, azul)'
	Borda :: assumi o valor reta ou boleada

OBSERVAÇÃO:

A Janela2D deve ter sido criada, caso contrário retornará erro.

OlharPara(x,y,z);	
Faz com o observador olhar para um ponto específico do espaço 3D.	
Parâmetros:	**X** :: coordenada x do observador
	Y :: coordenada y do observador
	Z :: coordenada z do obervador

OBSERVAÇÕES:

- Pode ser colocada tanto em Ambiente() como em Simulacao()
- O comando OlharPara não irá funcionar se o comando CamOrbita estiver habilitado

EXEMPLO:

```
Function Ambiente()
{
    CriarEspaco(0x3307ef);
   InserirObservador(45, 0.1, 20000);
   PosicaoDoObservador(30,30,30);
  OlharPara(0,0,0);
}
```

Ortografica(bool_x, bool_y, bool_z, fator);	
Cria uma câmera ortográfica.	
Parâmetros:	Bool_X :: Se **true**, exibe a visão 2D YZ;
	Bool_Y :: Se **true**, exibe a visão 2D XZ;
	Bool_Z :: Se **true**, exibe a visão 2D XY;
	Fator :: Aproximação da câmera (quanto maior o fator, mais próxima a câmera)
OBSERVAÇÕES:	
EXEMPLO: *Ortografica(true, false, false, 20);*	

Particula(nome, x, y, z);	
Cria uma particula.	
Parâmetros:	X :: coordenada X da partícula (cartesiana)
	Y :: coordenada Y da partícula (cartesiana)
	Z :: coordenada Z da partícula (cartesiana)
OBSERVAÇÕES: • Consideramos que toda a partícula é uma esfera de dimensões R=0.3 e divisão 8.8 • Chamada: **Nome = new Particula(x,y,z);**	
EXEMPLO: *Var P=[];* *P[1] = new Particula(10,10,10); // Cria uma particula de chamada p[1] na posição x=10, y=10 e z=10.*	

Pausa();
Pausa a simulação.
OBSERVAÇÃO: Este comando não tem parâmetros no argumento, no entanto, existe uma variável chamada **pausa_animacao** que pode assumir o valor true ou false. Por padrão, **pausa_animacao** = false.

Plano(L1, L2, L1Seg, L2Seg, x, y, z, rx, ry, rz, cor, visível, wireframe);	
Cria um plano no ambiente.	
Parâmetros:	(L1,L2) :: Medida dos lados do plano
	(L1Seg,L2Seg) :: Número de divisões do plano (por default é 1)
	(x,y,z) :: posição do plano no espaço 3D
	(rx,ry,rz) :: rotação em torno dos eixos x, y e z
	Cor :: cor da grade (formato 0xNNNNNN)
OBSERVAÇÕES: • Pode estar em Simulacao() ou Ambiente(); • L1Seg e L2Seg são serão visíveis se o modo wireframe for true; • A entrada para rotação deve ser o fator que multiplica PI para um resultado em radiano. Ex: 30º = (PI/6), então a entrada deve ser 1/6. • Chamada: Nome = new Plano(L1, L2, L1Seg, L2Seg, x, y, z, rx, ry, rz, cor, visível, wire)	
EXEMPLO: *P = new Plano (10.0, 0.0, 200.0, 1, 1, 5, 100, 0, 0, 0, 0, 0xff0000, true, false);*	

PosicaoDoObservador(x,y,z);	
Este comando posiciona o observador no ambiente tomando por base coordenadas cartesianas.	
Parâmetros:	**X** :: coordenada x do observador
	Y :: coordenada y do observador
	Z :: coordenada z do obervador

OBSERVAÇÕES:

- Pode ser colocada tanto em Ambiente() como em Simulacao()

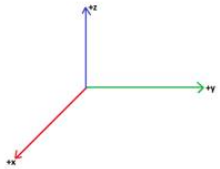
EXEMPLO:

```
Function Ambiente()
{
    CriarEspaco(0x3307ef);
   InserirObservador(45, 0.1, 20000);
   PosicaoDoObservador(30,30,30);
}
```

Retangulo (x1, y1, x2, y2, cor);	
Cria um retângulo na janela 2D.	
Parâmetros:	(x1,y1) :: coordenadas iniciais
	(x2,y2) :: coordenadas finais
	Cor :: cor no formato 'rgb(vermelho, verde, azul)'

OBSERVAÇÃO:

A Janela2D deve ter sido criada, caso contrário retornará erro.

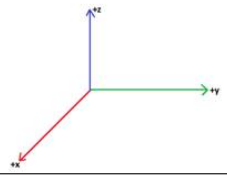
SistemaRetangular(n,boolx, booly,boolz);	
Este comando exibe os eixos do sistema cartesiano global que faz o papel de um sistema inercial, ou seja, em repouso absoluto no espaço relativo aos demais sistemas locais.	
Parâmetros: 	N :: tamanho dos eixos coordenados
	Boolx :: Se **true** vai exibir em pontilhado vermelho o eixo negativo de x
	Booly :: Se true vai exibir em pontilhado verde o eixo negativo de y
	Boolz :: Se true vai exibir em pontilhado azul o eixo negativo de z

OBSERVAÇÕES:

- Pode ser colocada tanto em Ambiente() como em Simulacao()
- O comando OlharPara não irá funcionar se o comando CamOrbita estiver habilitado

EXEMPLO:

```
Function Ambiente()
{
    CriarEspaco(0x3307ef);
   InserirObservador(45, 0.1, 20000);
   PosicaoDoObservador(30,30,30);
  OlharPara(0,0,0);
  SistemaRetangular(50, false, false, false);
}
```

SistemaRetangularL(n);	
Este comando cria um sistema de coordenadas cartesiano local. Diferente do SistemaRetangular, pode sofrer translação e rotação de acordo com objeto a qual está atrelado.	
Parâmetros: 	N :: tamanho dos eixos coordenados

OBSERVAÇÕES:

- Pode ser colocada tanto em Ambiente() como em Simulacao()

EXEMPLO:

Veja SistemaRetangular

Texto (mensagem, x, y, TextoCor, TamanhoTexto);	
Cria um texto na janela 2D.	
Parâmetros:	Mensagem :: string contendo o texto que deseja exibir
	(x,y) :: coordenadas da localização do texto na janela 2D
	TextoCor :: cor do texto
	TamanhoTexto :: tamanho do texto

OBSERVAÇÃO:

- A Janela2D deve ter sido criada, caso contrário retornará erro.
- Cor do texto no formato 'rgb(vermelho, verde, azul)'

UpdateComponentes(objeto);	
Este comando elimina as componentes cartesianas do objeto.	
Parâmetros:	Objeto :: nome do objeto

OBSERVAÇÕES:

- Deve estar em Simulacao();
- Para atualizar os componentes cartesianas na cena, após o comando UpdateComponentes(nome) devemos recriar o as componentes com o comando Componentes(x,y,z) que foi eliminado com UpdateComponentes(objeto) (veja o exemplo abaixo);
- Veja o comando Componentes;

EXEMPLO:

```
UpdateComponentes(c1);
c1 = new Componentes(px,py,pz);   /// Basta copiar o comando que usou para criá-lo logo depois de UpdateComponentes
```

UpdateVetor(objeto);	
Este comando elimina o vetor de nome "objeto".	
Parâmetros:	Objeto :: nome do objeto

OBSERVAÇÕES:

- Deve estar em Simulacao();
- Para atualizar o vetor na cena, após o comando Update(nome) devemos recriar o vetor que foi eliminado com Update (veja o exemplo abaixo);
- Veja o comando Vetor.

EXEMPLO:

```
UpdateVetor(R1);
R1 = new Vetor(0,0,0,px,py,pz,1,cinza);      // Basta copiar o comando que usou para criá-lo logo depois de UpdateVetor
```

Vetor(xo, yo, zo, vec_x, vec_y, vec_z, escala, cor);	
Cria um vetor cuja origem encontra-se em xo, yo e zo.	
Parâmetros:	Xo :: coordenada x da posição da origem do vetor
	Yo :: coordenada y da posição da origem do vetor
	Zo :: coordenada z da posição da origem do vetor
	Vec_x :: coordenada x do vetor (genérico)
	Vec_y :: coordenada y do vetor (genérico)
	Vec_z :: coordenada z do vetor (genérico)
	Escala :: fator de escala
	Cor :: cor do vetor

OBSERVAÇÕES:

- Os parâmetros Vec_x, Vec_y e Vec_z referem-se a qualquer tipo de vetor e não só ao vetor posição. Caso queira representar o vetor posição, então Vec_x deve ser descontado de xo, Vec_y de yo e Vec_z de zo.
- O fator de escala serve para ajustarmos o tamanho do vetor na cena. Se 1 então o vetor terá seu tamanho original, se 0.5 então será reduzido em 50% do valor original, se 2, terá o dobro do tamanho original. Este fator é importante quando trabalhamos com outras grandezas que não sejam de posição relativa, como força, velocidade, aceleração, entre outras.
- O parâmetro cor pode ser escrita como **0xNNNNNN**, discutido anteriormente (veja Janela3D). As seguintes palavras podem ser utilizadas para este parâmetro: **azule** (azul escuro), **vermelho**, **verde**, **amarelo**, **rosa**, **azulc** (azul claro), **cinza**, **branco**.
- Um vetor é um novo objeto na cena e deve ser criada do seguinte modo:
  Nome_do_vetor = new Vetor(xo,yo,zo, vec_x, vec_y, vec_z, escala, cor);

EXEMPLO:

```
// Vetor posição R com origem em (0,0,0) e extremidade em (px, py, pz). Escala 1 e cor verde.
R = new Vetor(0,0,0, px, py, pz,1, verde);

// Vetor velocidade V com origem em (px,py,pz) e coordenadas (vx, vy, vz). 50% menor e na cor azul escuro.
V = new Vetor(px,py,pz, vx, vy, vz,0.5, azule);

// Vetor posição relativa de nome Rel. Note que Vec_x = x-xo (posição final – posição inicial). Idem para Vec_y e Vec_z.
Rel = new Vetor(xo, yo, zo, x-xo, y-yo, z-zo, 1, vermelho);
```

8 CONSIDERAÇÕES FINAIS AO PROFESSOR

Preparar uma aula de Física que vai além do tradicional uso do quadro de escrever através do método expositivo não é nada fácil já que exige tempo e conhecimento que nem sempre o educador dispõe. Dentre algumas possibilidades de uso do computador para Ensinar Física, através de simulações, a AcliveJS aparece como mais uma alternativa baseada em construção através de linhas de código. A AcliveJS foi desenvolvida para permitir ao educador planejar e elaborar aulas mais ricas com atividades que permitam os estudantes compreender o assunto de forma diferenciada.

Outro aspecto importante da AcliveJS está na interatividade, permitindo que as simulações desenvolvidas sejam utilizadas como um laboratório virtual. Uma simulação disponibilizada online poderá ser acessada pelos alunos e, desta forma, poderá ser verificado os efeitos de suas ações sobre a simulação permitindo ao educador obter respostas quantitativas e qualitativas proporcionadas pelo sistema simulado.

É importante sempre ressaltar que a simulação pode ser executada offline ou online, como uma página da internet. Roda em diferentes navegadores, desde que possuam compatibilidade com o WebGL. O professor poderá desenvolver um conjunto de páginas com textos ou roteiros para serem explorados pelo estudante em casa, como atividade extraclasse. A combinação entre interatividade e visualização 3D fazem da AcliveJS uma ferramenta que pode vir ajudar no processo ensino-aprendizagem de Física.

Outro fator importante está na contribuição que novos usuários poderão dar para a melhoria da biblioteca. É importante que as simulações desenvolvidas pelos usuários possam ser disponibilizadas na página do projeto para que outros profissionais façam uso dela no futuro. Se o número de contribuições aumentarem a página do projeto poderá se tornar um repositório de simulações desenvolvidos pelos usuários que utilizam a AcliveJS. Isto auxiliará os educadores que evitam o uso da ferramenta por não entenderem de programação. Entretanto, as contribuições não podem se resumir ao desenvolvimento de simulações somente, mas também a criação de novas funcionalidades. Em sua versão 0.10 a AcliveJS dispõe de algumas dezenas de comandos, e no entanto, muitos aspectos ainda precisam ser trabalhados para torná-la uma biblioteca cada vez mais robusta e que atendam necessidades cada vez mais diversas. Ao longo dos anos a quantidade de funções pode alcançar a casa das centenas e com isso tornar as simulações cada vez mais atraentes e realistas. É importante que a

AcliveJS seja divulgada e explorada como uma possibilidade de ferramenta a ser utilizada em sala de aula.

Este trabalho está longe de ser concluído, pelo contrário, inicia-se a jornada com a versão 0.10 e é o uso constante que fará a ferramenta se ajustar as necessidades dos seus usuários. Mas mesmo em sua primeira versão já mostra a possibilidade de se elaborar atividades diferenciadas. É importante incentivar a formação de equipes para um trabalho em conjunto via GitHub propiciando um engajamento no planejamento de atividades com um valor pedagógico consistente e que atendam a todos os níveis e áreas da Física.

Apêndice 1

<!----Seu código HTML5 - Entradas de dados via formulários aqui (veja o manual) ---------->

* <h2>MOVIMENTO RETILÍNEO E UNIFORME (MRU)</H2> *

<table align=center width=80% border="0">

<tr>

<td>

**

</td>

<td valign=top align=justify>

*<h3>   *

*Com o objetivo de achar as leis que governam as várias mudanças que acontecem nos corpos conforme o tempo passa, devemos ser capazes de descrever as mudanças e ter alguma maneira de gravá-las. A mudança mais simples que pode ser observada em um corpo é a aparente mudança de sua posição com o tempo, que chamamos de movimento.
     *

*Na maioria das vezes o movimento de um corpo é complicado, como o movimento de um automóvel. É possível considerar um exemplo mais tranquilo, que obedecem leis mais simples. Podemos levar em consideração o Movimento Uniforme (MU).
*

*   *

O movimento uniforme é aquele em que o corpo percorre distâncias iguais em em intervalos de tempos iguais. É importante ressaltar que isto deve valer para quaisquer intervalos de tempo, pois pode acontecer que um corpo percorra distãncias iguais em intervalos de tempo iguais e, no entanto, as distâncias percorridas durante parte dessas frações de tempo sejam diferentes, embora os intervalos de tempo sejam iguais. </h3>

</td>

</table>

Apêndice 2

```
// Variáveis Globais aqui

var m=[]; var k=[]; var w=[]; var A=[]; phi=[];

var x=[]; var y=[]; var v=[]; var a=[]; var t;

var CE=[];

var vCE=[]; var aCE=[];

function Ambiente()

{           // Condições iniciais aqui

      m[1]=1.0; k[1]=1.0; A[1]=50.0; phi[1]=(Math.PI/180)*0.0;

      w[1]=Math.sqrt(k[1]/m[1]);

      x[1]=A[1]*Math.cos(w[1]*t + phi[1]); y[1]=-1;

      v[1]=(-1)*w[1]*A[1]*Math.sin(w[1]*t + phi[1]);

      a[1]=(-1)*Math.pow(w[1],2)*y[1];

      m[2]=1.0; k[2]=1.0; A[2]=50.0; phi[2]=(Math.PI/180)*90.0

      w[2]=Math.sqrt(k[2]/m[2]);

      x[2]=A[2]*Math.cos(w[1]*t + phi[2]); y[2]=4;

      v[2]=(-1)*w[2]*A[2]*Math.sin(w[1]*t + phi[2]);

      a[1]=(-1)*Math.pow(w[2],2)*y[2];

      t = clock.start();

      Janela3D('rgb(220,240,240)', 0.75, 0.95);

      InserirObservador(45,0.1,20000);

      PosicaoDoObservador(0.0,0.0,80.0);

      Ortografica(0,0,15,false,false,true,25);

      SistemaRetangular(50,false,false,false);

      GradeXY('rgb(0,0,255)','rgb(50,50,0)',100,5,false);

      CE[1]=new CorpoExtenso(2,2,0.5,x[1],y[1],0,0,0,0,'rgb(255,0,0)',false,0);

      CE[2]=new CorpoExtenso(2,2,0.5,x[2],y[2],0,0,0,0,'rgb(255,0,0)',false,0);

      vCE[1]=new Vetor(x[1],y[1],0,0,v[1],0,0.3,'rgb(255,0,0)');

      vCE[2]=new Vetor(x[2],y[2],0,0,v[2],0,0.3,'rgb(255,0,0)');
```

```
        aCE[1]=new Vetor(x[1]-0.2,y[1],0,0,a[1],0,0.15,'rgb(0,0,255)');
        aCE[2]=new Vetor(x[2]-0.2,y[2],0,0,a[2],0,0.15,'rgb(0,0,255)');
}
function Simulacao()
{
            x[1]=A[1]*Math.cos(w[1]*t + phi[1]);
            x[2]=A[2]*Math.cos(w[2]*t + phi[2]);
            v[1]=(-1)*A[1]*w[1]*Math.sin(w[1]*t + phi[1]);
            v[2]=(-1)*A[2]*w[2]*Math.sin(w[2]*t + phi[2]);
            a[1]=(-1)*Math.pow(w[1],2)*x[1];
            a[2]=(-1)*Math.pow(w[2],2)*x[2];
            CE[1].CorpoExtenso.position.set(x[1],y[1],0);
            CE[2].CorpoExtenso.position.set(x[2],y[2],0);
        //Update Vetores
        UpdateVetor(vCE[1]);  vCE[1]=new Vetor(x[1],y[1],0,v[1],0,0,0.3,'rgb(255,0,0)');
        UpdateVetor(vCE[2]);  vCE[2]=new Vetor(x[2],y[2],0,v[2],0,0,0.3,'rgb(255,0,0)');
        UpdateVetor(aCE[1]);  aCE[1]=new Vetor(x[1]-0.2,y[1],0,a[1],0,0,0.15,'rgb(0,0,255)');
        UpdateVetor(aCE[2]);  aCE[2]=new Vetor(x[2]-0.2,y[2],0,a[2],0,0,0.15,'rgb(0,0,255)');
        t = parseFloat(clock.getElapsedTime());
}
```