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

DESENVOLVIMENTO DA BIBLIOTECA ACLIVEJS PARA SIMULAÇÕES DE FENÔMENOS FÍSICOS

Wallace Robert da Silva Nascimento

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação Campus UFRJ - Macaé no Curso de Mestrado Profissional de Ensino de Física (MNPEF), como parte dos requisitos necessários à obtenção do título de Mestre em Ensino de Física.

Orientador:
Prof. Dr. Claudio Ccapa Ttira

Macaé - RJ
Abril 2017

DESENVOLVIMENTO DA BIBLIOTECA ACLIVEJS PARA SIMULAÇÕES DE FENÔMENOS FÍSICOS

Wallace Robert da Silva Nascimento

Orientador:
Prof. Dr. Claudio Ccapa Ttira

Dissertação de Mestrado submetida ao Programa de Pós-Graduação Campus UFRJ-Macaé Professor Aloísio Teixeira no Curso de Mestrado Profissional de Ensino de Física (MNPEF), como parte dos requisitos necessários à obtenção do título de Mestre em Ensino de Física

Aprovada por:

_____
Prof. Dra. Marta Feijó Barroso – UFRJ/RJ

_____
Prof. Dr. Habib Salomon Dumet Montoya – UFRJ/Macaé

_____
Prof. Dra. Valéria Nunes Belmont – UFRJ/Macaé

Macaé
Abril 2017

MODELO de FICHA CATALOGRÁFICA

Nascimento, Wallace Robert da Silva
Desenvolvimento da Biblioteca AcliveJS para Simulações de Fenômenos Físicos / Wallace Robert da Silva Nascimento – Rio de Janeiro, 2017.

Orientador: Claudio Ccapa Ttira
Dissertação (mestrado) – Universidade Federal do Rio de Janeiro, Campus Macaé Professor Aloísio Teixeira, Programa de Pós-Graduação em Ensino de Física, 2017.
Referências Bibliográficas: f. 70-73.

1. Ensino de Física. 2. Simulação Computacional. 3. Modelos Físicos. 4. javascript. I. Ttira, Claudio Ccapa, orient. II. Desenvolvimento da Biblioteca AcliveJS para Simulações de Fenômenos Físicos.

Dedico este trabalho:

A todos os professores que, de alguma forma, buscam modificar sua prática para um melhor Ensino de Física.

À minha esposa Graziela Mendes Resende. Obrigado por tudo!

Ao meu filho Áquilla Odlanier Faria Nascimento.

Minha mãe Neusa Maria da Silva Nascimento.

Meu pai Hamilton Teixeira Fuly.

Agradecimentos

À minha esposa Graziela Mendes Resende por estar ao meu lado a todo instante, me dar forças para lutar e estimular a seguir em frente sempre em busca por um Ensino de Física no Brasil cada vez melhor.

A toda minha família. Principalmente minha mãe Neusa Maria da Silva e o meu pai Hamilton Teixeira Fuly.

RESUMO

DESENVOLVIMENTO DA BIBLIOTECA ACLIVEJS PARA SIMULAÇÕES DE FENÔMENOS FÍSICOS

Wallace Robert da Silva Nascimento

Orientador:
Prof. Dr. Claudio Ccapa Ttira

Dissertação de Mestrado submetida ao Programa de Pós-Graduação Campus UFRJ-Macaé Professor Aloísio Teixeira no Curso de Mestrado Profissional de Ensino de Física (MNPEF), como parte dos requisitos necessários à obtenção do título de Mestre em Ensino de Física.

Muitas vezes o processo Ensino-Aprendizagem sobre um determinado fenômeno físico fica comprometido devido à dificuldade de abstração por parte dos estudantes. Nestes casos, é possível utilizar recursos de mídia, sejam na forma de vídeos ou simulações computacionais. O uso dessas tecnologias busca facilitar o entendimento do que está sendo estudado. Partindo deste princípio como produto final da dissertação do Mestrado Profissional foi desenvolvida uma biblioteca em javascript, a AcliveJS, cujo propósito é permitir que professores possam mostrar ou desenvolver simulações de fenômenos físicos por meio de navegadores de internet. O objetivo deste trabalho é apresentar a ferramenta de simulação AcliveJS e mostrar ao professor como desenvolver uma simulação através de um exemplo. Apontar os resultados obtidos da aplicação da ferramenta em sala de aula e discutir a aceitação, vantagens e desvantagens do uso da ferramenta. Espera-se que este trabalho seja um convite para o professor melhorar sua prática profissional e incorporar cada vez mais o uso de novos recursos de mídia no Ensino de Física, propiciando o estudo e a reflexão por parte dos estudantes.

Palavras-chave: Ensino de Física. AcliveJS. Javascript. Simulação de Física.

Macaé
Março 2017

ABSTRACT

ACLIVEJS LIBRARY DEVELOPMENT FOR PHYSICAL PHENOMENA SIMULATIONS

Wallace Robert da Silva Nascimento

Supervisor:
Prof. Dr. Claudio Ccapa Ttira

Master's Dissertation submitted to the UFRJ-Macaé Campus Postgraduate Program Professor Aloísio Teixeira in the Professional Master's Degree Course in Physics Teaching (MNPEF), as part of the requirements for obtaining a Master's degree in Physics Teaching.

Often the Teaching-Learning process about a certain physical phenomenon is compromised due to the students' difficulty of abstraction. In these cases, you can use media resources, whether in the form of videos or computer simulations. The use of these technologies seeks to facilitate the understanding of what is being studied. Based on this principle as the final product of the Master's dissertation, a javascript library, AcliveJS, was developed, whose purpose is to enable teachers to show or develop simulations of physical phenomena through internet browsers. The objective of this work is to present the AcliveJS simulation tool and to show the teacher how to develop a simulation through an example. Aim the results obtained from the application of the tool in the classroom and discuss the acceptance, advantages and disadvantages of using the tool. It is hoped that this work will be an invitation for the teacher to improve his professional practice and to incorporate more and more the use of new media resources in Physics Teaching, favoring the study and the reflection on the part of the students.

Keywords: Physics education. AcliveJS. Javascript. Simulation of Physics.

Macaé
March 2017

Sumário

Capítulo 1 INTRODUÇÃO....11

Capítulo 2 REFERENCIAL TEÓRICO....15

2.1 A Aprendizagem Significativa de David Ausubel....15
2.2 Simulações Computacionais no Ensino de Física....17

Capítulo 3 ACLIVEJS....23

3.1 Como a AcliveJS funciona....25
3.2 A Estrutura de uma Simulação....28
3.3 Inserindo Textos e Imagens....31
3.4 Construindo Modelos com a AcliveJS....32
3.4.1 Escolha do Modelo: Movimento Harmônico Simples....33
3.4.2 A Física do MHS....34
3.4.3 Construindo a cena em Ambiente()....37
3.4.4 Equações de Evolução....43
3.4.5 Objetos Complementares....46

Capítulo 4 APLICANDO O PRODUTO: AS SIMULAÇÕES COMPUTACIONAIS SOBRE O MOVIMENTO RELATIVO COM O USO DA ACLIVEJS....49

4.1 O Movimento Relativo....51
4.1.1 Problema do Movimento Relativo em Uma Dimensão....56
4.1.2 Problema do Movimento Relativo em Duas Dimensões....59

Capítulo 5 RESULTADOS E ANÁLISE DE DADOS....63

Capítulo 6 PERSPECTIVAS FUTURAS E CONSIDERAÇÕES FINAIS....68

Referências Bibliográficas....70

APÊNDICE 1: LISTAGEM DO PROGRAMA EXIBINDO TEXTOS E IMAGENS...74

APÊNDICE 2: LISTAGEM DA SIMULAÇÃO DO MHS....79

APÊNDICE 3: LISTAGEM DA SIMULAÇÃO DO MOV. RELATIVO EM 1D....77

APÊNDICE 4: LISTAGEM DA SIMULAÇÃO DO MOV. RELATIVO EM 2D....81

APÊNDICE 5: MANUAL DA ACLIVEJS 0.10....85

Figuras

Figura 3.1: Pastas e arquivos da AcliveJS.....25
Figura 3.2: Esquema do princípio de funcionamento da AcliveJS.....26
Figura 3.3: Estrutura dos programas AcliveJS.....31
Figura 3.4: Exemplo de exibição de textos e imagens.....32
Figura 3.5: Layout da simulação MHS.....34
Figura 3.6: Sistema massa-mola sem atrito.....34
Figura 3.7: Estudo do MHS massa-mola como projeção do MCU.....37
Figura 3.8: Código base da AcliveJS.....37
Figura 3.9: Declarando variáveis.....38
Figura 3.10: Atribuição dos valores iniciais das variávies.....39
Figura 3.11: Codificação dentro da function Ambiente().....40
Figura 3.12: Ficheiro do comando Janela3D.....41
Figura 3.13: Primeira execução da simulação do programa MHS.....43
Figura 3.14: Equações de evolução do MHS.....43
Figura 3.15: MHS. Frequências angulares iguais.....45
Figura 3.16: MHS. Frequências angulares distintas.....45
Figura 3.17: Atualizações dos vetores em Simulacao().....46
Figura 3.18: Vetor velocidade e aceleração dos blocos 1 e 2.....47
Figura 4.1: Movimento relativo em uma dimensão.....52
Figura 4.2: Movimento relativo em duas dimensões.....56
Figura 4.3: Desenho para auxiliar na resolução do problema 118.....57
Figura 4.4: Simulação apresentada em sala referente ao problema 118.....58
Figura 4.5: Desenho para auxiliar na resolução do problema 77.....59
Figura 4.6: Representação dos vetores velocidades do problema 77.....60
Figura 4.7: Simulação a partir do referencial ‘solo’.....61
Figura 4.8: Simulação a partir do referencial ‘carro’.....61

Siglas

TIC – Tecnologias da Informação e Comunicação

OED – Objetos Educacionais Digitais

MHS – Movimento Harmônico Simples

OA – Objeto de Aprendizagem

EJS – Easy Java Simulations

PhET – Physics Education Technology (Tecnologia Educacional em Física)

API – Application Programming Interface (Interface de Programação de Aplicativos)

WebGL – Web Graphics Library (Biblioteca Gráfica para Web)

HTML – HyperText Markup Language (Linguagem de Marcação de Hipertexto)

PDF – Portable Document Format (Formato Portátil de Documento)

IDE – Integraded Development Environment (Ambiente de desenvolvimento integrado)

FPS – Frames per Second (Quadros por segundo)

POO – Programação Orientada a Objeto

Capítulo 1
Introdução

Como professor de Física do Ensino Superior, percebo que, na maioria das vezes, os estudantes têm dificuldade em compreender os conteúdos de Física das disciplinas iniciais dos cursos de Engenharia. As principais questões levantadas pelos alunos, desconsiderando-se o afastamento dos estudos por motivos pessoais, tais como trabalho, família ou doença, referem-se, principalmente, a um curso de Ensino Médio deficiente, que apresentaria poucas aulas de Física e falta de professores. Estes apontam, sobretudo, a dificuldade do professor em lecionar o conteúdo e a ausência de recursos didáticos além da tradicional aula expositiva.

A carência dos conceitos físicos provenientes desses fatores vem seguida de uma falta de habilidade matemática, mesmo no nível mais básico, como trabalhar com frações ou vetores. Segundo Santos et al. (2004), estas lacunas são devidas a um ensino através de fórmulas, definições e exercícios padronizados, que sobrepõem os conceitos realmente relevantes, tirando do contexto a formação de diversas habilidades e hábitos, dificultando a aprendizagem, ao ponto de afastar o aluno do interesse pelo conhecimento científico. Santos reforça que as dificuldades dos estudantes ocorrem devido à precária formação de professores e ao uso de livros que seguem os mesmos padrões de apostilas de cursos preparatórios, que ajudam a produzir pseudoconhecimentos desvinculados das necessidades de formação dos estudantes e de conhecimentos científicos relevantes. Luckesi (2008) evidencia que a atenção dos alunos está totalmente centrada nas promoções através de notas, não importando como elas são obtidas e, segundo Libâneo (1994), o professor passa a matéria e os alunos interagem reproduzindo o que está escrito, praticam o que foi passado em exercícios de classe e decoram tudo para a prova. Libâneo explora essa questão em seu livro *Didática*, escrito há mais de 20 anos e, ainda assim, este é o método predominante nos três níveis de Ensino em nosso país (Libâneo, 2012). Estas aulas estritamente teóricas, que enfatizam a memorização, carregadas de expressões matemáticas sem conexões com a realidade do aluno, acabam acarretando em consequências, tais como dificuldades de aprendizagem, notas baixas, desestímulo e frustração em relação à disciplina ministrada.

Diante desse panorama e da minha percepção quanto ao rumo que o processo de ensino-aprendizagem em Física tomaria, encontrei, na Teoria da Aprendizagem

Significativa de David P. Ausubel, o ponto de partida para o desenvolvimento dessa pesquisa. De acordo com Saraiva-Neves et al. (2006), a aprendizagem significativa de Ausubel é "um processo por meio do qual novas ideias e informações a serem aprendidas e retidas são ancoradas em conceitos específicos relevantes existentes na estrutura cognitiva do indivíduo".

Assim sendo, o interesse pelo estudo da Física, seja no Ensino Médio ou Superior, deve ocorrer através de um processo de ensino e aprendizagem diferenciado, partindo das referências que os próprios alunos trazem e da sua visão de mundo. É preciso investir cada vez mais na formação dos professores e dar as condições para que eles possam trabalhar a construção do conhecimento com os alunos. Ao introduzir aulas mais dinâmicas e críticas, com foco nos fenômenos e não estritamente na matemática, o professor ajuda a tornar os alunos agentes ativos no processo ensino-aprendizagem (Gomes et al., 2010).

Uma das linhas de pesquisa do Mestrado Nacional Profissional em Ensino de Física, trata do desenvolvimento de produtos e processos de ensino e aprendizagem que utilizem tecnologias de informação e comunicação, tais como aplicativos para computadores, mídia para tablets, plataforma para simulações e modelagem computacionais, aquisição automática de dados, celulares e redes sociais.

O objeto dessa pesquisa foi delineado pensando nas dificuldades dos alunos em abstrair a partir da explicação do professor e na possibilidade do uso de uma simulação personalizada para mediar o processo. Partindo da temática referente à plataforma para simulações e modelagem computacionais, levantei as seguintes questões para serem investigadas:

> (1) É possível criar uma biblioteca de funções escritas em javascript, para estimular a habilidade de programar do professor, permitindo-o criar suas próprias simulações de Física para serem utilizadas em sala de aula?
>
> (2) Que fatores facilitariam ou dificultariam o uso do recurso proposto?

Como hipótese de pesquisa, supus que seria possível aproveitar a habilidade de programar dos professores permitindo-os criar suas próprias simulações e que isto estimularia o aluno a aprender Física e favoreceria uma aprendizagem significativa. Ao criar a simulação, o professor aprimoraria cada vez mais seus conhecimentos, fosse na correção dos processos computacionais ou na observação do resultado final. Uma

simulação envolve o desenvolvimento de um modelo físico do fenômeno a ser reproduzido. Portanto, é preciso estudar a Física envolvida e entendê-la. É o aprender fazendo. Errando, depurando, construindo e analisando. Além disso, conjecturei que, se essa construção fosse feita em conjunto, a interatividade do professor com o aluno seria melhor. Quanto à segunda questão, presumi que a popularização do uso dos computadores e o acesso à internet seriam fatores que facilitariam o uso do recurso. Em contrapartida, a noção pré-concebida de que programação computacional é difícil e destinada às poucas pessoas, poderia dificultar o contato desse recurso com professores e alunos.

O referencial teórico foi construído a partir da teoria da aprendizagem significativa, proposta por David Paul Ausubel na década de 1960. Segundo Valadares et al. (2009), a aprendizagem significativa ocorre quando o aluno consegue, por meio de algum instrumento didático, atribuir significado ao que está sendo aprendido. Este significado é pessoal e está ligado ao conhecimento prévio do aluno. Caso esta atribuição não exista, então não é considerada como significativa e, sim, mecânica. Outro aspecto teórico abordado refere-se ao uso das Tecnologias de Informação e Comunicação (TICs) no processo ensino aprendizagem, com foco em simulações computacionais para o ensino de Física. De acordo com Studart (2015), o uso destas tecnologias digitais, quando aplicadas em educação, deve promover a motivação e a colaboração do aluno, tornando-o mais engajado diante do processo de ensino e aprendizagem. Alguns autores mostram que o uso de simulações computacionais no Ensino de Física tem contribuído para a efetivação de uma aprendizagem significativa (Araújo et al., 2004; Mendes et al., 2012). Entretanto, Santos et al. (2004) alerta que é preciso entender o recurso didático que está sendo usado e utilizá-lo como um elemento que mediará a relação do aluno com o conteúdo a ser ensinado e que isto deve ser realizado com um planejando consciente por parte do professor.

Tendo em vista as questões apontadas acima, criei a biblioteca de funções escritas em javascript, batizada de AcliveJS, que surgiu da reflexão sobre o ensino de Física no nível Superior. Num primeiro momento, com foco na aprendizagem significativa proposta por Ausubel (1968), utilizei o fenômeno do Movimento Harmônico Simples para explicar o princípio de funcionamento da ferramenta e suas possibilidades. Num segundo momento, apliquei duas simulações sobre Movimento Relativo, em uma e duas dimensões, respectivamente, numa turma de 1º período na

disciplina de Física I do curso de Engenharia Civil de uma Universidade particular da cidade de Cabo Frio, RJ.

O objetivo principal desta pesquisa foi contribuir com o processo ensino aprendizagem de Física, a partir da teoria da aprendizagem significativa de Ausubel, por meio de simulações computacionais utilizando navegadores de internet e a ferramenta AcliveJS.

O presente trabalho é estruturado de tal forma que, no Capítulo 2 exploro as bases de construção do referencial teórico dessa pesquisa. Nele, abordo a Teoria da Aprendizagem Significativa proposta por David Ausubel e em seguida apresento o uso de simulações computacionais no Ensino de Física através dos Objetos Educacionais Digitais. No Capítulo 3, apresento uma visão geral da ferramenta AcliveJS e como utilizá-la para a construção de uma simulação. O assunto escolhido para explicar algumas das funcionalidades da AcliveJS foi o Movimento Harmônico Simples. Sobre tal tema, e ainda no capítulo 3, abordo também a teoria da Física envolvida nessa pesquisa. Todas as simulações apresentadas neste trabalho foram criadas utilizando a AcliveJS, fazendo dela parte do produto instrucional que é apresentado no Apêndice 3 por meio de um manual de uso. No Capítulo 4, apresento a metodologia adotada para essa pesquisa, isto é, como o Objeto de Aprendizagem foi desenvolvido e aplicado em sala de aula. Nele, utilizo duas simulações sobre o movimento relativo, em uma e duas dimensões. O produto instrucional desenvolvido para esta pesquisa refere-se as simulações aplicadas em aula e também ao manual de como utilizar a AcliveJS para o desenvolvimento de simulações para o Ensino de Física. No capítulo 5, apresento os resultados obtidos e a análise dos dados qualitativos e quantitativos. Finalmente, no Capítulo 6, faço uma breve explanação sobre as perspectivas futuras para o uso da AcliveJS e de como ela poderia auxiliar estudantes e professores nas construções de simulações, com o objetivo de favorecer a aprendizagem significativa.

Capítulo 2
Referencial Teórico

2.1 A Aprendizagem Significativa de David Paul Ausubel

De acordo com Moreira (2012), a Aprendizagem Significativa, originalmente proposta por David Ausubel na década de 1960, se ocupa da aquisição significativa de um corpo organizado de conhecimentos em situação formal de ensino e aprendizagem. Segundo Ausubel, o fator isolado mais importante influenciando a aprendizagem é aquilo que o aprendiz já sabe e, portanto, é importante que o professor observe isto e o ensine de acordo. A Aprendizagem Significativa é um processo onde novas ideias e informações são ancoradas em conceitos específicos relevantes já existentes na estrutura cognitiva do aluno (Saraiva-Neves et al., 2006). Se um dado conhecimento prévio não servir de apoio para a aprendizagem significativa de novos conhecimentos então ele não passará por este processo. Moreira, ao fazer uma descrição detalhada sobre o que é a Aprendizagem Significativa, deixa evidente que existem duas condições para que este processo ocorra: o material de aprendizagem deve ser potencialmente significativo e o aprendiz deve apresentar uma predisposição a aprender. O termo 'potencialmente significativo' vem do fato de que o significado está nas pessoas e não nos materiais. Logo, não existe livro, aula ou simulação computacional significativo. O que dá significado ao processo ensino aprendizagem é justamente a interação que existe entre o professor e o aluno por intermédio destes recursos.

Subsunçores é o nome dado por Ausubel às informações e aos conceitos já existentes na estrutura cognitiva do indivíduo (Mendes et al., 2012). Para Ausubel, quando o aprendiz não dispõe de subsunçores que lhe permitam atribuir significados aos novos conhecimentos, então é possível fazer uso do que se chamou de organizadores prévios, que podem ser trabalhados num modo expositivo ou comparativo. O organizador expositivo é usado quando o aluno não tem subsunçores sobre o tema a ser explorado, e portanto, o professor faz a ponte entre o que o aluno sabe e o que deveria saber para que o material se torne potencialmente significativo. Já o organizador comparativo é utilizado quando o material é relativamente familiar e, neste caso, o professor irá utilizá-lo para auxiliar o aluno a integrar novos conhecimentos à estrutura cognitiva já existente. O organizador prévio é um recurso instrucional e pode ser um enunciado, uma pergunta, uma situação problema, uma demonstração, um filme, uma leitura introdutória ou uma simulação computacional.

Ao contrário da Aprendizagem Significativa, Ausubel também discute sobre a Aprendizagem Mecânica. Nesta última, há pouca ou nenhuma interação entre as novas informações e os conceitos relevantes já existentes na estrutura cognitiva do aluno. Trata-se de um armazenamento literal que não segue regras ou normas, sem fundamento lógico e que depende da vontade ou arbítrio daquele que age, no caso, o professor. Na Aprendizagem Mecânica, o professor é um elemento ativo enquanto o aluno é passivo, apenas memorizando aquilo que serve para a realização das provas e o assunto é esquecido logo em seguida. Moreira (2012) enfatiza que este tipo de aprendizagem é a conhecida 'decoreba', infelizmente, tão incentivada na escola. A Aprendizagem Significativa valoriza exatamente o oposto, pois não é arbitrária, possui significado, implicando na compreensão, na capacidade de explicar, descrever e enfrentar novas situações.

Apesar disso, Ausubel não separa a Aprendizagem Mecânica da Significativa. Para ele a Aprendizagem Mecânica acaba sendo necessária quando o aluno não possui subsunçores, neste caso, visam à construção dos mesmos. Ainda assim, ele enfatiza que é uma ilusão pensar que o aluno pode aprender de forma mecânica pois, ao final do processo, a aprendizagem acaba sendo significativa. Ausubel afirma que é uma ilusão pensar que uma boa explicação e um aluno interessado são condições suficientes para uma Aprendizagem Significativa. Além destes fatores, a existência de subsunçores adequados e o uso de materiais potencialmente significativos auxiliam o processo, entretanto, muitas vezes não é isto que predomina nas escolas, que enfatizam, durante a maior parte do tempo, a Aprendizagem Mecânica.

Moreira (2012) faz uma distinção entre uma aprendizagem receptiva e por descoberta. Aprender receptivamente significa que o aluno não precisa descobrir para aprender e a aprendizagem por descoberta implica em que o aprendiz primeiramente descubra o que vai aprender. Independentemente, um novo conhecimento pode ser trabalhado através de um livro, de uma aula, de um experimento no laboratório, de um filme, de uma simulação computacional, entre outros meios. E, ao contrário do que possa parecer, não necessariamente aprender receptivamente implica em passividade e é um erro pensar que a aprendizagem por descoberta garantirá a Aprendizagem Significativa. Em ambos os casos, não pode ser desprezado o conhecimento prévio adequado, a predisposição para aprender, um material potencialmente significativo e um professor engajado com o uso do recurso instrucional a ser utilizado e; conhecedor do

tema a ser explorado. É fato que a aprendizagem por recepção favorece a Aprendizagem Mecânica, mas isto não é regra. Entretanto, Moreira (2012) enfatiza ainda que, mesmo que o ensino seja centrado no aluno como se defende atualmente, a aprendizagem predominante no Ensino Médio e Superior continua sendo receptiva, sustentando a Aprendizagem Mecânica. O contexto social em que vivemos hoje exige 'provas' de que o aluno sabe ou não sabe. Este tipo de avaliação, baseada em números e reprodução, é comportamentalista e promove a Aprendizagem Mecânica já que não entra na questão do significado e da compreensão.

Ausubel (2003) reforça que a assimilação de conceitos é facilitada quando se parte de ideias mais gerais, que considera mais inclusivas, para ideias específicas, consideradas menos inclusivas. Esta abordagem está ancorada em duas hipóteses: (a) é mais fácil para o ser humano compreender as partes partindo do todo do que o inverso; (b) a organização do conteúdo de uma certa disciplina possui uma estrutura hierárquica onde o todo, baseado em ideias mais inclusivas, ocupa o topo e as partes, baseado em ideias menos inclusivas, ocupam setores hierárquicos inferiores.

Mesmo assim, Ausubel admite que o processo inverso deve ocorrer, de maneira que o sistema não pode ser uma via de mão única, ou seja, do geral para as partes. Esse procedimento é denominado de reconciliação integrativa e deve ser explorada pelo professor obedecendo os critérios que favorecem a Aprendizagem Significativa, mencionada em parágrafos anteriores.

Segundo Saraiva-Neves et al. (2006), para favorecer a Aprendizagem Significativa é importante que o professor leve em consideração quatro tarefas: (a) identificar a estrutura conceitual do que vai ser ensinado; (b) identificar quais os subsunçores são relevantes à aprendizagem do conteúdo e direcionar o ensino a partir deles utilizando um material educacional potencialmente significativo; (c) diagnosticar o que o aluno já sabe; (d) ensinar utilizando recursos e princípios que favoreçam a Aprendizagem Significativa.

2.2 Simulações Computacionais no Ensino de Física

Seymour Paper foi o teórico mais conhecido sobre o uso de computadores na educação, um dos pioneiros da inteligência artificial e criador da linguagem de programação LOGO em 1968 (Pocrifka et al., 2009). Paper admite que o professor deve

estimular o educando a construir seu próprio conhecimento por intermédio de alguma ferramenta, como por exemplo, o computador (Costa et al., 2012). De acordo com Costa, a justificativa para a não utilização do computador como um recurso instrucional passa por condicionantes externos baseados em argumentos como a falta da disponibilidade do microcomputador, de não haver tempo para a utilização do equipamento em aula ou de que os programas não dão orientações claras sobre o que fazer com eles. Portanto, ainda existe alguma resistência quanto ao uso do computador nas escolas como um recurso educacional. Costa acrescenta que a decisão do uso do computador deve estar ligada ao reconhecimento do potencial desta ferramenta e sua utilidade na aprendizagem, buscando sempre avaliar quais os impactos positivos no conhecimento e nos rendimentos dos alunos. Entretanto, não basta ter vontade e reconhecer o microcomputador como uma ferramenta importante para o ensino se falta a experiência para operá-lo. Neste caso será difícil tomar uma decisão fundamentada e esclarecida o que pode trazer resultados negativos. Studart et al. (2010), ao fazer uma análise sobre a utilização do computador em sala de aula por professores de Física no Ensino Médio, mostrou que os professores, embora não tenham dificuldades com o uso do computador não os utilizavam em sala a não ser como apoio às aulas expositivas, ou seja, de forma passiva. Isto é um fator limitante quando se tem à disposição outras possibilidades. A diversidade do uso do microcomputador fica evidente em um estudo sobre o proveito de tecnologias computacionais no Ensino de Física, realizada por Araújo et al. (2004), que classifica o uso do computador em sete modalidades pedagógicas a partir da análise de 109 trabalhos abordando o tema sobre o uso das Tecnologias de Comunicação e Informação (TIC). Tais artigos foram agrupados, de acordo com a função do computador, nas seguintes modalidades: (a) Programas demonstrativos; (b) Modelagem e simulação computacional; (c) Coleta e análise de dados em tempo real; (d) Recursos de multimídia; (e) Comunicação à distância; (f) Resolução algébrica / numérica e visualização de soluções matemáticas; (g) Estudo de processos cognitivos.

Segundo Studart (2015), as TIC são tecnologias e instrumentos usados para compartilhar, distribuir e reunir informações, bem como para comunicar-se umas com as outras mediante o uso de computadores ou equipamentos conectados, como é o caso da internet. Os OED são uma classe de TIC chamadas Objetos Educacionais Digitais que podem ser utilizadas tanto no ensino presencial como à distância. De acordo com o

autor, qualquer entidade digital que pode ser utilizado para a aprendizagem, educação ou treinamento pode ser classificada como OED.

Com tantas possibilidades nas quais o computador pode ser aproveitado, faz-se necessário, cada vez mais, estimular o seu uso de forma ativa, visando a melhorar o processo do ensino de física nas escolas. Alguns trabalhos evidenciam a importância do uso do software Modellus para o desenvolvimento de simulações de física, principalmente no estudo da Mecânica (Araújo et al., 2004; Mendes et al., 2012; Mendes, 2014). Figueira (2005) ressalta a importância do uso de outro software, o Easy Java Simulations (EJS), e acrescenta que o propósito do programa é simplificar e agilizar a construção de aplicativos e, mais especificamente, a modelagem de problemas teóricos nas áreas de ciências. Silva et al. (2011) apresenta o software SimQuest como alternativa ao Modellus e ao EJS. O autor reforça que o ensino de física é uma das áreas de estudo que mais pode se beneficiar com o uso destas novas tecnologias computacionais. Mariano não faz uma comparação do SimQuest com os dois programas citados, entretanto, deixa evidente alguns aspectos negativos desses softwares, como a ausência de alguns elementos de animação no Modellus e a necessidade de digitar códigos na linguagem java em algumas situações no EJS. Não é a intenção deste trabalho evidenciar aspectos positivos ou negativos de um software que pode utilizado como um Objeto de Aprendizagem (OA), entretanto, por se tratar de um programa que já vem pronto é possível que não atenda às necessidades de alguns professores. De acordo com Barroso et al. (2009), o conceito de OA surgiu na década de 1990 como uma proposta de paradigma na elaboração de materiais instrucionais.

Os programas mencionados correspondem a apenas três alternativas entre outras disponíveis. A vantagem do Modellus, EJS e SimQuest está no fato de serem opções gratuitas e que executam diretamente na máquina, ou seja, não é preciso estar conectado à internet. Todas eles podem ser úteis para professores que estão ávidos por construir seus próprios objetos de aprendizagem. Outra possibilidade está no uso de banco de simulações presentes na internet, o que acaba atendendo àqueles que não dispõem de tempo hábil para desenvolver os próprios modelos. Studart et al. (2010) apresenta o canal comPADRE (www.compadre.org), considerando-o um repositório digital disponibilizando OA de excelente qualidade e totalmente gratuito. Este canal possui muitas simulações prontas desenvolvidas com o EJS e apresenta tópicos sobre programação e outras ferramentas que podem ser usadas no ensino de física. O canal

comPADRE encontra-se na Língua Inglesa, o que pode gerar alguma resistência por professores brasileiros. Outra iniciativa na produção de simulações para o ensino de Física foi desenvolvida por Carl Wieman, prêmio Nobel de 2001, ao criar o PhET, sigla em inglês para *Tecnologia Educacional em Física* (https://phet.colorado.edu/pt_BR/). O PhET apresenta uma gama de simulações sobre diversos tópicos da Física, com a vantagem do site estar em português e os programas poderem ser executados no modo online ou offline, já que existe opção de download. Segundo Barroso et al. (2006), ensinar Física constitui um desafio permanente e uma das tarefas do professor é buscar, por meio de materiais disponíveis ou produção própria, OAs adequados a seus estudantes. Diante desta iniciativa, Barroso dispõe de um repositório constituído de aplicativos computacionais disponíveis online (www.if.ufrj.br/~marta). O site oferece a opção de download de um arquivo compactado contendo todas as simulações para ser gravado num pendrive ou CD-rom. Barroso et al. (2009) reforça ainda que a ideia de construir repositórios de materiais didáticos surge do conceito de OA. Estes Objetos de Aprendizagem, devidamente organizados e disponibilizados para que professores e alunos tenham acesso, economiza tempo e dinheiro, facilitando a utilização por parte do professor no desenvolvimento de cursos e processos instrucionais.

Um outro aspecto importante do uso de simulações no ensino de Física, reside do fato de que é possível, para o usuário mais experiente, desenvolver os próprios programas utilizando uma linguagem de programação. Apesar de exigir do professor um conhecimento técnico mais refinado, a vantagem, quando comparado com softwares prontos ou bancos de simulações disponíveis na internet, é que quando o educador cria uma simulação através de linhas de códigos ele se envolve com ela e sabe, de antemão, os caminhos necessários para suprir as necessidades dos seus alunos e pode ajustar o programa para atender casos particulares. Segundo Souza Filho et al. (2009), o uso de uma linguagem de programação como o *Python* (www.python.org), pode resolver algumas dificuldades encontradas em laboratórios de Física, como custos de equipamentos, dificuldades na realização de medidas, entre outros. O autor salienta a importância do uso do *vPython* (www.vpython.org), ou *Visual Python*, em simulações de Mecânica Clássica através da transformação de equações dos fenômenos conhecidos em algoritmos computacionais como, por exemplo, o algoritmo de Euler, por ser de fácil entendimento e implementação. De acordo com Sherwood et al. (2009), estudantes novatos que tiveram instruções sobre programação utilizando *vPython*, duas horas de

treinamento mostraram ser o suficiente para que eles fossem capazes de desenvolver modelagens computacionais sérias. Assim como o *Python*, o *vPython* é um programa gratuito e de código aberto e encontra-se disponível para Windows, Linux, e Macintosh. Uma alternativa online para o uso do *vPython* é o *GlowScript* (www.glowscript.org) onde toda codificação e execução das simulações ocorrem diretamente no navegador de internet, com armazenamento gratuito na nuvem via cadastro no site. O *Python* é apenas uma linguagem entre tantas que podem ser utilizadas para o desenvolvimento de simulações. O próprio EJS possui o recurso de construção de modelos via linguagem java. Muitos a*pplets* com simulações de Física disponíveis na internet foram programados em java, sendo um dos mais famosos o repositório de Walter Fendt (www.walter-fendt.de/ph14br/), que disponibiliza as simulações em diversos idiomas, incluindo o português do Brasil. Segundo Studart (2015), os *applets* são uma categoria de simulações que visam uma atividade específica e são utilizados para simular situações de aprendizagens. Barroso et al. (2006) deixa evidente que as simulações apresentadas em seu repositório foram criadas utilizando o *Action Script*, no *Macromedia Flash*. Desta forma, a princípio qualquer linguagem pode ser utilizada para o desenvolvimento de simulações de Física, dentre elas é possível citar C/C++, Basic, Pascal, Fortran, Javascript, Lua, entre outras.

De acordo com Costa et al. (2012), é necessário reconhecer o potencial dos OED, suas utilidades na aprendizagem e, posteriormente, escolher dentre tantas possibilidades qual delas utilizar. A escolha de um software, ou o desenvolvimento de um programa próprio, deve passar por um processo de adequação que depende de como este se insere nas práticas de ensino, das dificuldades dos alunos e da análise das situações vivenciadas por eles, buscando desta forma uma aprendizagem significativa, tornando o material instrucional utilizado potencialmente significativo. Vencida estas barreiras o próximo passo consiste em conhecer e dominar cada ferramenta, percebendo o seu potencial pedagógico. Além disso, é necessário que o professor responda à pergunta: Vale a pena recorrer a estas tecnologias para atingir os objetivos previstos em termos de economia de tempo ou maior eficácia? A decisão pelo uso de simulações em aulas de Física deve buscar uma ação transformadora baseada numa decisão profissional de querer mudar a direção para um modelo em que o aluno seja o elemento central, isto é, faça parte do processo ensino aprendizagem de forma ativa e que, juntos, vão

construindo o saber por meio de uma aprendizagem significativa (Costa et al., 2012; Barroso et al., 2009).

Dependendo da metodologia utilizada para ensinar Física através do uso de simulações, existem duas maneiras de se usar OA em atividades de ensino aprendizagem: a exploratória e a expressiva. Em atividades exploratórias os estudantes utilizam modelos e representações desenvolvidas por outras pessoas para estudar o assunto de interesse. Neste tipo de atividade a interação do estudante com a simulação é feita através de mudanças de parâmetros via teclado, mouse, entre outros. Portanto, educador deve ter muito cuidado ao sugerir uma simulação de Física ao educando sem que esse tenha tido uma orientação prévia ou que esteja fora do ambiente de ensino. Barroso et al. (2009) reforça que um aplicativo sem uma orientação instrucional apropriada, seja por guia impresso ou pela presença do professor, pode ser utilizado de forma incorreta. Simulações em atividades exploratórias não são suficientes para garantir uma aprendizagem significativa e, portanto, depende de orientações externas para o seu uso. No modo expressivo, os estudantes são estimulados à construção de seus próprios modelos e determinam a maneira de representar seus resultados. Este tipo de atividade dá ao estudante amplo espaço de exploração e intervenção. É possível adotar uma combinação dos dois métodos, propondo que os estudantes façam modificações nos modelos desenvolvidos pelos professores, adaptando-os as novas situações.

De acordo com Machado et al. (2009), a utilização de uma ferramenta computacional proporciona condições para que o educando possa aprender e assimilar de forma significativa quando comparada as limitações do lápis e papel.

Sob essa perspectiva, para que a hipótese apresentada na introdução fosse testada, sobre se seria possível utilizar a habilidade de programar do professor para desenvolver os próprios objetos de aprendizagem, a ideia de utilizar uma simulação disponível em repositórios da internet foi excluída e criou-se uma ferramenta, a AcliveJS, especialmente para a presente pesquisa. Esse OED e suas particularidades será apresentado no capítulo a seguir.

Capítulo 3
AcliveJS

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.

O foco principal da AcliveJS está em permitir ao usuário utilizar modelos físicos e matemáticos elaborados por outras pessoas através de atividades exploratórias ou desenvolver seus próprios modelos matemáticos numa atividade expressiva por meio de uma linguagem de programação. Diante das três possibilidades apresentadas – softwares prontos, repositórios de simulações disponíveis na internet e uso de uma linguagem de programação –, a AcliveJS se enquadra na terceira opção, oferecendo 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 numa atividade exploratória.

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.

Não é objetivo neste trabalho explicar como a AcliveJS foi desenvolvida, entretanto, ela foi criada especificamente para esta pesquisa, a fim de auxiliar na elaboração do produto instrucional. Desta forma, será apresentada através de uma breve explicação do seu princípio de funcionamento, avaliando suas funcionalidades e potencialidades, sem adentrar em questões técnicas do seu desenvolvimento. É importante ressaltar que, além da opção pelo uso de uma linguagem de programação, foram dadas a esta, novas funções que visam a atender algumas necessidades específicas por intermédio de uma biblioteca, totalmente escrita na linguagem *javascript*, o que culminou na AcliveJS.

Antes da construção do material instrucional, definido como as simulações de física sobre o Movimento Relativo em uma e duas dimensões utilizando a AcliveJS, foram discutidos os critérios que seriam utilizados para a aplicação do mesmo. O primeiro critério escolhido 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.

3.1 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 3.1.

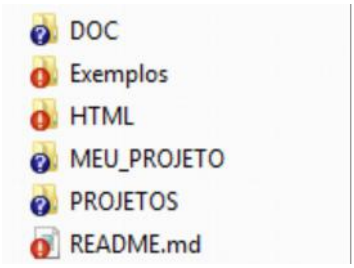


Figura 3.1: Pastas e arquivos da AcliveJS

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 3.2 exibe um fluxograma do seu princípio de funcionamento.

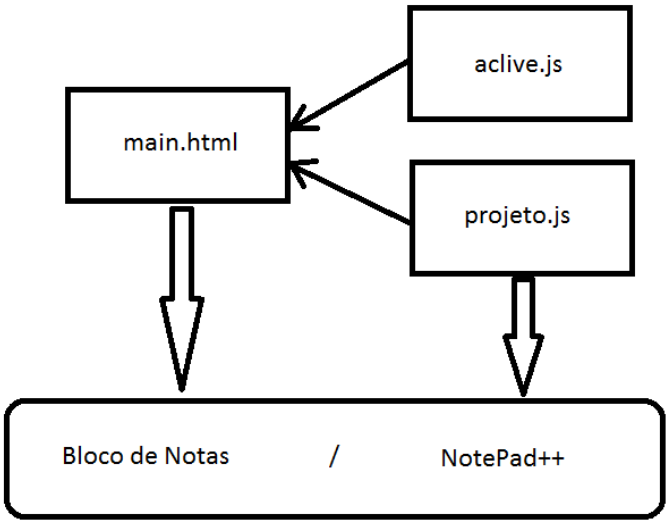


Figura 3.2: Esquema do princípio de funcionamento da AcliveJS

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 3.2 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 3.2 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.

3.2 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 3.3 exibe um fluxograma da estrutura dos programas construídos com a *AcliveJS.*

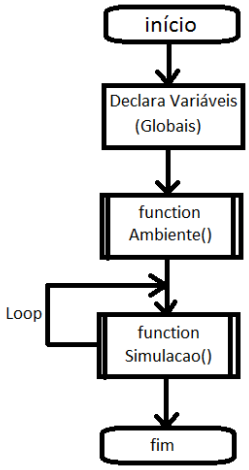


Figura 3.3: estrutura dos programas AcliveJS

3.3 Inserindo textos e imagens

Tanto o Modellus como 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 3.4 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 3.4: 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 permitirá copiar o código gerado e colar na região demarcada para este fim no arquivo *main*.

3.4 Construindo Modelos com a AcliveJS

Nos tópicos 3.2 e 3.3, 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 produto instrucional, que se baseia na construção das simulações sobre o Movimento Relativo para serem aplicadas em sala de aula.

3.4.1 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 3.5 ilustra o layout escolhido para esta simulação:

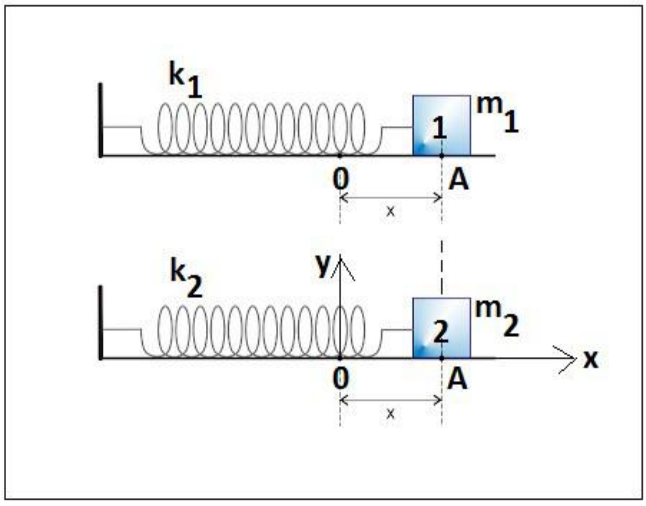


Figura 3.5: Layout da Simulação MHS

3.4.2 A Física do MHS

Considere o sistema massa-mola representado na Figura 3.6 abaixo.

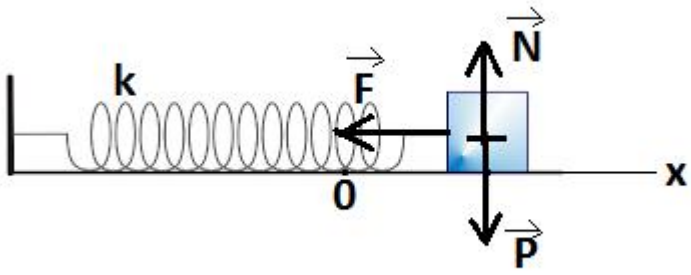


Figura 3.6: Sistema massa-mola sem atrito

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 3.6 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 3.4.1, 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 3.4.1 indica que o sentido da força F e do deslocamento em torno do ponto de equilíbrio estável são sempre contrários.

$$F = -kx \qquad (3.4.1)$$

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 3.4.2. Esta é uma equação diferencial ordinária de 2ª ordem.

$$\frac{d^2x}{dt^2} + \frac{k}{m}x = 0 \qquad (3.4.2)$$

Equação 3.4.2 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 3.4.3. Esta será a equação de estado, ou evolução, utilizada para esta simulação.

$$x(t) = A.cos(\omega t + \phi) \qquad (3.4.3)$$

Sendo:

$$\omega = \sqrt{\frac{k}{m}} \qquad (3.4.4)$$

Segue as grandezas presentes nas Equações 3.4.3 e 3.4.4:

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 3.4.3, 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 3.7.

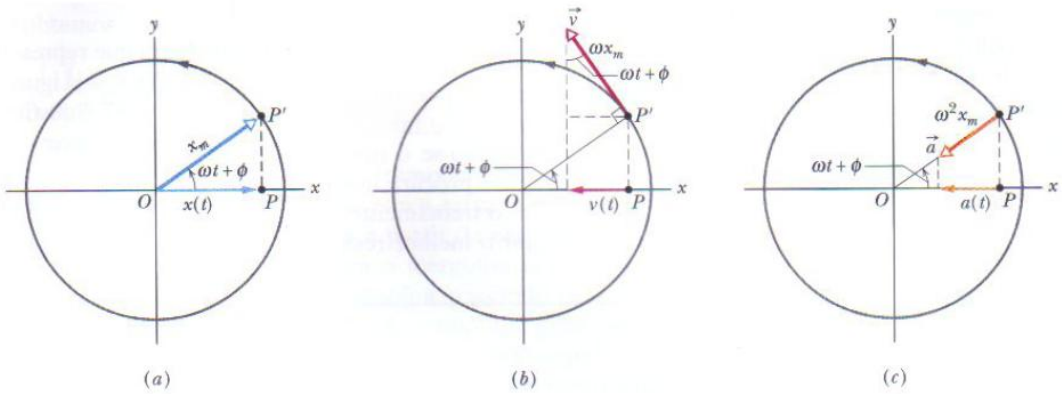


Figura 3.7: Estudo do MHS massa-mola como projeção do Movimento Circular Uniforme (MCU)

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.

3.4.3 Construindo a cena em Ambiente()

A codificação foi realizada utilizando o IDE ConText (http://www.contexteditor.org). A Figura 3.8 ilustra o código base digitado no ConText.

```
//
function Ambiente()
{

}

function Simulacao()
{

}
```

Figura 3.8: 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 3.9 ilustra esse processo.

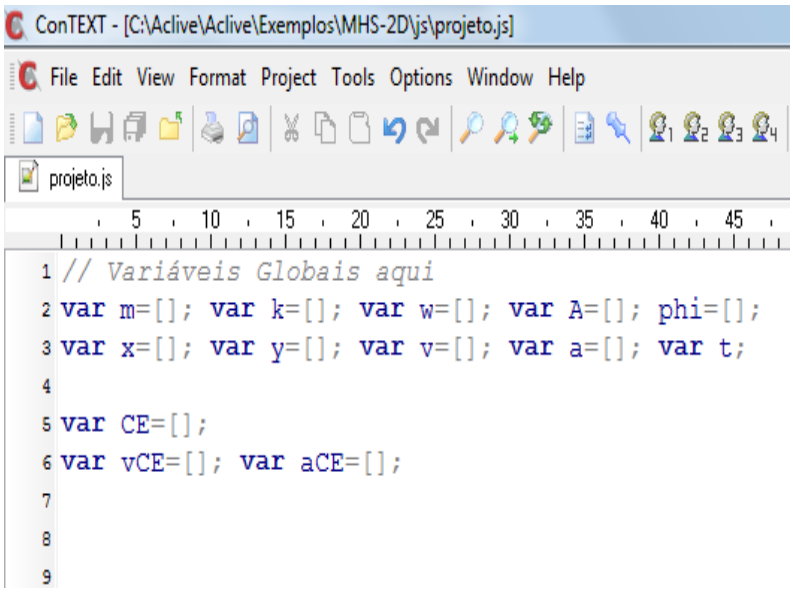


Figura 3.9: Declarando variáveis.

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 3.10 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 3.10: 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 3.4.4, 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 3.4.3. 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 3.4.3 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 3.4.3. 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 3.11 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 3.11: 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 3.11, 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 3.12 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 3.12: 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 3.13 mostra a cena com todos os elementos descritos na listagem.

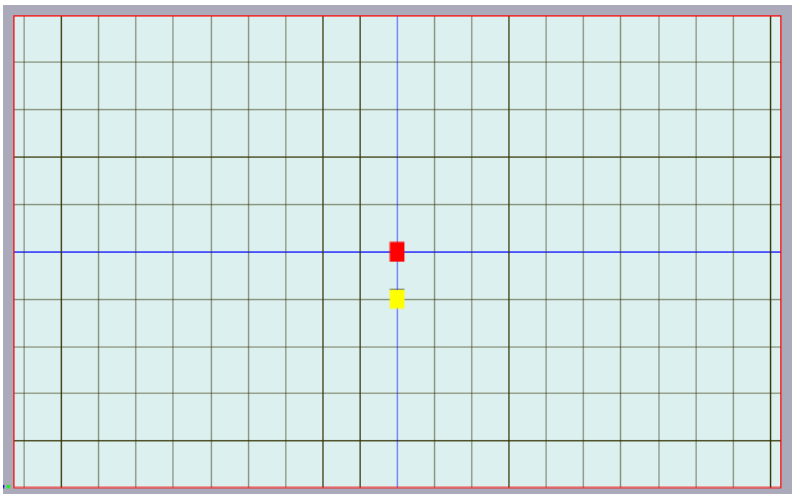


Figura 3.13: Primeira execução da simulação do MHS.

3.4.4 Equações de Evolução

A Equação 3.4.3 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 3.14.

```
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 3.14: 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 3.4.3. 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 3.15 mostra um instante da simulação com os valores iniciais adotados.

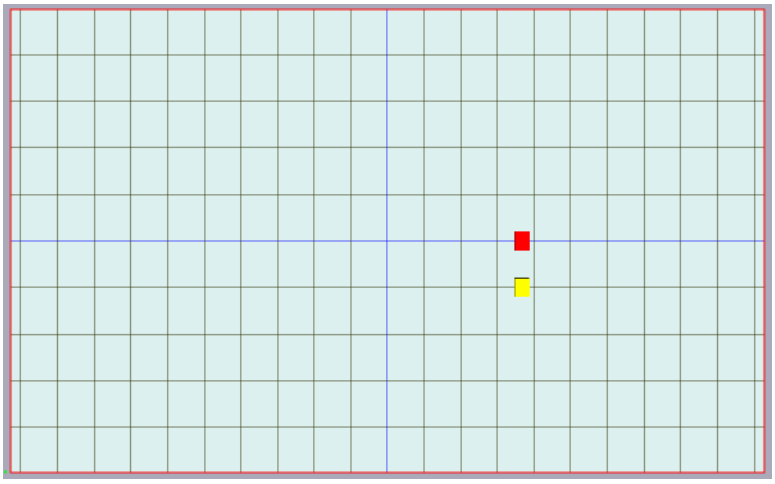


Figura 3.15: MHS. Frequências angulares iguais.

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 3.16 um momento qualquer dessa simulação onde os corpos oscilam com frequências distintas.

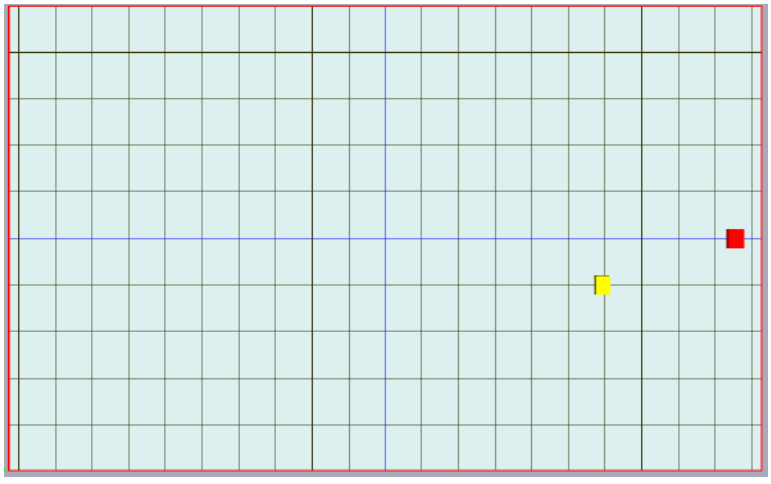


Figura 3.16: MHS. Frequências angulares distintas.

3.4.5 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 3.4.3 encontra-se a velocidade do bloco representada através da Equação 3.4.5 abaixo:

$$v = -A\omega . sen(\omega t + \phi) \qquad (3.4.5)$$

Derivando a Equação 3.4.5 chega-se a Equação 3.4.6, que é a equação de evolução para a aceleração:

$$a = -\omega^2 x \qquad (3.4.6)$$

Com as Equações 3.4.5 e 3.4.6 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 3.17 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 3.17: 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 3.18 mostra a execução tomada num instante qualquer onde aparece o vetor velocidade, em vermelho, e aceleração, em azul, de ambos os blocos.

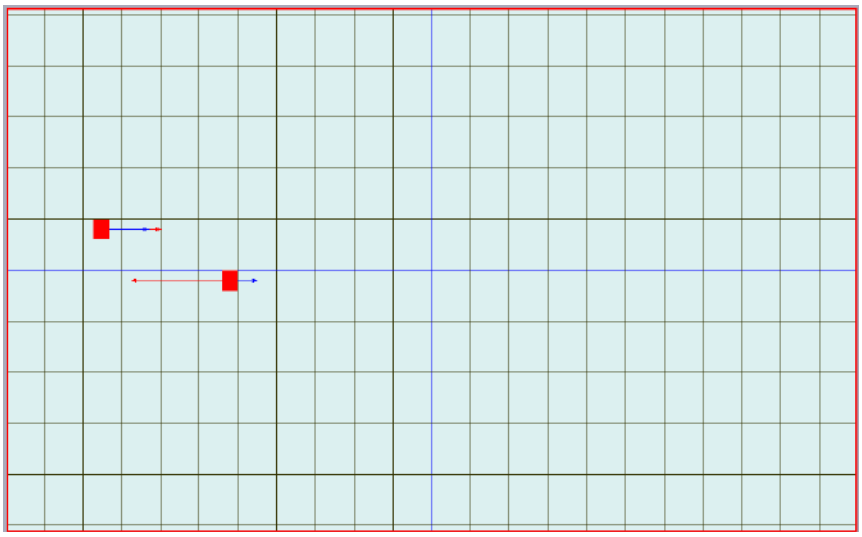


Figura 3.18: Vetor velocidade e aceleração dos blocos 1 e 2.

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 exibição 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 um produto potencialmente significativo. A AcliveJS, apresentada neste projeto, encontra-se na versão 0.10.

Capítulo 4:
A Construção da Metodologia e a Aplicação do Produto: as simulações computacionais sobre o movimento relativo com o uso da AcliveJS.

Para mostrar as potencialidades da AcliveJS na elaboração de simulações computacionais no ensino de Física, foram desenvolvidas duas simulações sobre o movimento relativo, em uma e duas dimensões, respectivamente. A escolha por esse tema surgiu da observação das dificuldades de aprendizagem apresentadas pelos alunos da disciplina Física I, do 2º período de um curso de Engenharia Civil, de uma universidade particular do município de Cabo Frio, RJ. Após a escolha do tema, foi preciso definir como seriam criadas as simulações.

A opção metodológica foi a de elaborar simulações a partir de dois exercícios extraídos do livro Fundamentos de Física, volume 1 (Halliday et al., 2014). As pesquisas nos repositórios presentes na internet, e que foram citados anteriormente no Capítulo 2, a saber: comPADRE, Phet, Walter Fendt e Barroso, mostraram que todas elas procuraram ajudar na construção de um conhecimento geral sobre o tema, desde que sob orientações adequadas. Nenhum repositório pesquisado possuía uma simulação sobre um exercício específico de algum livro ou material. Na internet, as simulações sobre o tema são gerais. O exercício de um livro apresenta um resultado, uma resposta. Quando a simulação é feita utilizando-se um exercício, é possível verificar sua acurácia, já que os resultados apresentados na simulação precisam ser os mesmos apontados no exercício.

Em sala de aula, para analisar as potencialidades da AcliveJS como um OED no Ensino de Física, as atividades foram desenvolvidas com 15 alunos que, ao final do processo, foram estimulados a escrever suas percepções em relação à ferramenta. A exposição do tema Movimento Relativo foi trabalhada inicialmente através de uma aula ministrada utilizando o método tradicional de ensino. Esta aula serviu como um organizador prévio, como discutido no Capítulo 2. O que dá significado ao processo ensino e aprendizagem é a interação entre o professor e o aluno. Levando em consideração que os alunos possuíam pouco ou nenhum subsunçores sobre o tema abordado, foi utilizado o organizador expositivo sem o uso de qualquer tipo de recurso

tecnológico, mas tão somente a exposição oral e o quadro de escrever, daí o uso do termo ‘tradicional’. Esta aula foi trabalhada através de uma leitura introdutória, explorada no tópico 4.1, em seguida foram feitas as demonstrações das equações utilizadas no movimento relativo, a saber as Equações 4.1.8 e 4.1.9. O próximo passo foi a exposição de duas situações problemas por meio dos dois exercícios do livro texto adotado. De acordo com Ausubel (1968) a assimilação dos conceitos é facilitada quando se parte de ideias mais gerais para as mais específicas. Então, o organizador prévio foi trabalhado da seguinte maneira: primeiramente foi realizada uma leitura introdutória, em seguida foram feitas as demonstrações matemáticas e os enunciados que levaram às situações problema. Não foi aplicado qualquer tipo de questionário nesta aula, mas tão somente apresentado o movimento relativo. Os exercícios resolvidos em sala foram os apresentados nos tópicos 4.1.1 e 4.1.2.

Semanas depois, o tema do movimento relativo foi retomado utilizando o produto instrucional, as simulações desenvolvidas com a AcliveJS. Partiu-se do princípio que os alunos seriam capazes de reconhecer os elementos da Matemática e da Física envolvidos, tais como sistema cartesiano, vetores, posição relativa, velocidade relativa e referencial adotado. A ideia consistiu em aproveitar os subsunçores dos alunos, construídos a partir do organizador prévio, visando buscar um sentido mais amplo desses conhecimentos, permitindo dar significado a um novo conhecimento, neste caso, o movimento de um corpo relativo a outro que também se encontra em movimento. Para esta aula, foram explorados novamente os conceitos físicos do movimento relativo através dos exercícios do livro texto Fundamentos de Física, vol. 1 (Halliday et al., 2014), presentes nos tópicos 4.1.1 e 4.1.2, agora, por meio de duas simulações. Alguns pontos da leitura inicial, trabalhada na aula anterior, foram retomados à medida que os alunos foram sendo estimulados a perguntar sobre o tema a partir do que era observado nas simulações. Este tratamento inverso foi proposital e visou explorar o que Ausubel chamou de reconciliação integrativa. A estrutura conceitual do tema abordado, a identificação dos subsunçores relevantes à aprendizagem do conteúdo e o ensino utilizando um recurso instrucional, por meio de simulações – que buscaram favorecer a aprendizagem significativa – constituíram os critérios abordados no presente trabalho.

Buscou-se identificar posteriormente, através das respostas dos alunos, se o produto instrucional representou para eles algo relevante, e se este ajudou-os na construção do conhecimento sobre o movimento relativo ao se fazer a comparação entre a aula 'tradicional' e a que utilizou as simulações.

Após a aula aplicando o produto instrucional em sala, foi solicitado aos estudantes que descrevessem a impressão que tiveram do uso das simulações com o Objeto de Aprendizagem quando comparada à aula ministrada, sobre o mesmo tema, sem o uso das simulações. Um questionário semi-estruturado foi elaborado com o intuito de obter dados quantitativos e qualitativos. Portanto, foram sugeridos alguns tópicos:

a. Como você classifica a aula sobre o Movimento Relativo com o uso das simulações utilizando a ferramenta AcliveJS?
b. Você teve uma aula expositiva sobre este tema, semanas antes, na qual o quadro e o pincel foram utilizados. Como você classifica este tipo de aula tradicional quando comparada ao uso das simulações computacionais?
c. Quais foram os prós e contras do uso deste tipo de Objeto de Aprendizagem na sua opinião? Você utilizaria a AcliveJS para desenvolver suas próprias simulações?

No tópico abaixo, será explorado o tema escolhido para o desenvolvimento da presente pesquisa, isto é, o Movimento Relativo, bem como os dois exercícios escolhidos do livro de Halliday. A seguir, no capítulo 5, serão apresentados os resultados obtidos nessa pesquisa, bem como sua análise.

4.1 O movimento relativo

No livro Fundamentos de Física, vol. 1 (Halliday et al., 2014), o capítulo 3 é dedicado ao estudo de vetores e o capítulo 4 ao do movimento em duas e três dimensões, que utilizará extensivamente o que foi aprendido no capítulo anterior desse livro. O movimento relativo é o último tópico do capítulo 4 e é explorado inicialmente em uma dimensão, ou seja, sem a necessidade do uso da notação vetorial. A abordagem é realizada de forma simples, com dois observadores situados em referenciais distintos, um em repouso e outro descrevendo um movimento uniforme numa direção constante,

ambos observando um carro situado num ponto qualquer na mesma direção que passa pelos dois observadores. A Figura 4.1 ilustra esta situação.

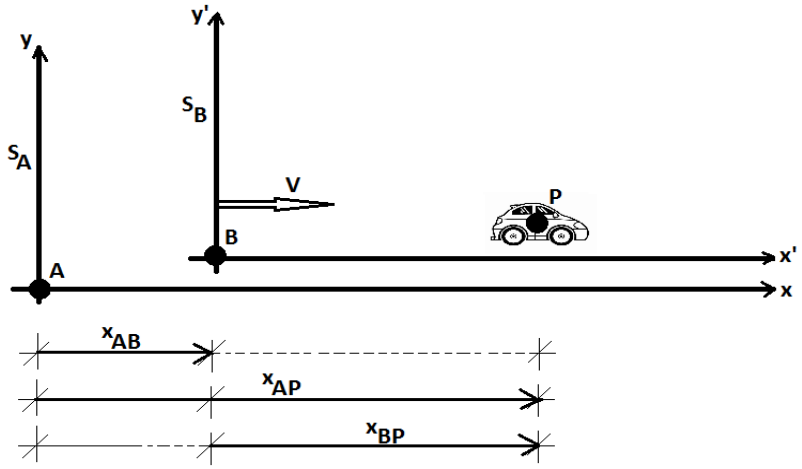


Figura 4.1: Movimento Relativo em uma dimensão.

O referencial S_A do observador A encontra-se em repouso. O referencial S_B do observador B move-se com velocidade constante V no sentido positivo do eixo x. Ambos os referenciais coincidem na direção xx´ e os observadores estão na origem dos seus respectivos sistemas de coordenadas, sendo representados por um disco pintado de preto. A Figura 4.1 mostra S_B ligeiramente deslocado para facilitar a visualização. Uma partícula P, representada por um automóvel, está numa posição qualquer que coincide com a direção xx´. É importante salientar que estão sendo considerados referenciais em repouso ou que se movem com velocidade constante, como é o caso do S_B. No entanto, esta restrição não se aplica à partícula P, cuja velocidade pode variar. As distâncias representadas na Figura 4.1 devem ser interpretadas da seguinte maneira:

x_{AB} – posição do observador B relativo ao observador A.
x_{AP} – posição da partícula P relativa ao observador A.
x_{BP} – posição da partícula P relativa ao observador B.

Pela observação da Figura 4.1 é possível deduzir a Equação 4.1.1.

$$x_{BP} = x_{AP} - x_{AB} \qquad (4.1.1)$$

A Equação 4.1.1 fornece a posição da partícula P em relação ao referencial B que se move com rapidez constante ao longo do eixo x positivo. Matematicamente, esta posição é obtida pela diferença da posição de P relativa ao referencial A com a posição de B também relativa ao mesmo referencial, sendo este último adotado como referencial fixo. Este resultado pode ser encontrado com mais facilidade ao observarmos as setas logo abaixo dos sistemas de coordenadas S_A e S_B da Figura 4.1.

Ao derivar a Equação 4.1.1 em relação ao tempo, obtém-se a velocidade da partícula P relativa ao observador B. Isto pode ser verificado nas equações 4.1.2 e 4.1.3.

$$\frac{d}{dt}(x_{BP})=\frac{d}{dt}(x_{AP})-\frac{d}{dt}(x_{AB}) \qquad (4.1.2)$$

Desta forma as velocidades estão relacionadas através da Equação 4.1.3.

$$v_{BP}=v_{AP}-v_{AB} \qquad (4.1.3)$$

Para relacionar as acelerações de P medidas por B e por A em um mesmo instante, basta calcular a derivada da Equação 4.1.3 em relação ao tempo. Isso pode ser observado nas equações 4.1.4 e 4.1.5.

$$\frac{d}{dt}(v_{BP})=\frac{d}{dt}(v_{AP})-\frac{d}{dt}(v_{AB}) \qquad (4.1.4)$$

Como v_{AB} é constante, o último termo da Equação 4.1.4 é nulo, levando ao resultado a Equação 4.1.5.

$$a_{BP}=a_{AP} \qquad (4.1.5)$$

De acordo com a Equação 4.1.5, a aceleração de uma partícula, medida por observadores em referenciais que se movem com velocidade constante um em relação ao outro, é exatamente a mesma.

De acordo com Bassalo (1997), aparentemente, foi o filósofo grego Zenão de Eléia (~500 a.C) o primeiro a se preocupar com o movimento relativo dos corpos. Zenão questionava o movimento por meio de paradoxos, posteriormente conhecidos

como paradoxos de Zenão. Entretanto, esta questão só foi retomada no século XVII pelos italianos Giordano Bruno (1548-1600) e Galileu Galilei (1564-1642). Giordano Bruno propõem experimentos que poderiam ser realizados a bordo de um navio em movimento uniforme, com o objetivo de explicar o movimento relativo. Estes experimentos são divididos em duas partes nos quais, em ambos, são colocados observadores no solo, considerado como um referencial fixo, ou em um navio que, obrigatoriamente, deve descrever um movimento retilíneo e uniforme. No primeiro experimento, dois observadores estão em um navio que se desloca em MRU. Um observador encontra-se no extremo do mastro e o outro sobre um ponto qualquer no tombadilho. Uma pedra é abandonada pelo observador no ponto mais alto e ambos seguem a trajetória da pedra com o olhar até esta atingir o piso. Segundo Giordano Bruno, ambos os observadores devem concordar que, independente da velocidade do navio, a trajetória da pedra é retilínea. No segundo experimento, um observador está no ponto mais alto do mastro e o outro está situado nas margens sobre um elevado cuja altura coincide com a do primeiro observador. Quando o navio avança em MRU e as posições dos observadores se "alinham", cada um deixa cair uma pedra em sincronia. De acordo com Giordano Bruno cada pessoa verá cair sua pedra numa trajetória retilínea. No entanto, a trajetória descrita pela pedra abandonada por uma dessas pessoas, vista pela outra, será uma curva.

Galileu retoma a análise da queda de um corpo em um navio parado ou em movimento em seu livro *Diálogo sobre os dois Principais Sistemas do Mundo: o Ptolomaico e o Copernicano*, publicado em 1632. Ele discute também a queda de um corpo do alto de uma torre, o movimento de projéteis e o voo das aves em uma Terra em movimento. É importante ressaltar que, naquela época, acreditava-se que a Terra era estática e ocupava o centro do Universo. Galileu utiliza o Princípio da Relatividade do Movimento ou Princípio da Independência dos Movimentos para refutar as objeções aristotélicas sobre o movimento de nosso planeta. Em seu livro *Discursos e Demonstrações Matemáticas em torno de Duas Novas Ciências*, publicado em 1638, Galileu utiliza este mesmo princípio para demonstrar a trajetória parabólica de corpos lançados horizontalmente de uma superfície acima do solo. Atualmente este princípio é conhecido como Princípio de Galileu ou Lei de Composição de Velocidade de Galileu, e afirma que a velocidade de um objeto, em relação a um corpo em repouso, é igual à velocidade que ele possui em relação a um outro corpo que se desloca com velocidade constante em relação ao corpo parado, acrescida desta última velocidade.

Na linguagem atual, escrita de forma analítica, o Princípio de Galileu está representada na Equação 4.1.6, que é uma forma distinta de escrever a equação 4.1.3.

$$v_{AP} = v_{BP} + v_{AB} \qquad (4.1.6)$$

Uma notação diferente para a Equação 4.1.6 consiste em adotar V, em maiúscula, para a velocidade do referencial em MRU relativo ao referencial fixo (v_{BA}), e, v' para a velocidade da partícula P relativo ao referencial com velocidade v (v_{BP}). Então a velocidade v para a partícula P relativa ao referencial em repouso (v_{AP}), é dada pela equação 4.1.7.

$$v = v' + V \qquad (4.1.7)$$

Galileu mostrou em seu livro *Diálogos*, mencionado anteriormente, que este princípio levava a um outro resultado importante, no qual afirma que é impossível determinar se um navio está parado ou em movimento uniforme, por intermédio de uma experiência mecânica realizada em um dos seus camarotes fechados.

Esta afirmação vem do resultado analítico mostrado na Equação 4.1.5. Galileu percebeu que os corpos, de um modo geral, se comportavam da mesma maneira quando submetidos a referenciais que estão em repouso ou se moviam em linha reta com velocidade constante (Feynman, 2008). Sendo assim, estes sistemas são equivalentes e a percepção humana é incapaz de distinguir entre um ou outro. Referenciais em repouso ou que descrevem MRU são chamados referenciais inerciais. E esta afirmação leva ao que ficou conhecido como Princípio da Equivalência de Galileu. A Equação 4.1.5 mostra que ambos observadores medem a mesma aceleração, ou seja, a aceleração de uma partícula é a mesma para todos os observadores em movimento relativo de translação uniforme. A aceleração, portanto, permanece invariável ao passarmos de um referencial a um outro qualquer, que encontra-se em movimento relativo de translação uniforme.

A Figura 4.2 mostra a mesma situação, só que em duas dimensões. A linha de raciocínio é a mesma, com a diferença que será preciso escrever as Equações 4.1.1, 4.1.3 e 4.1.5 em sua forma vetorial.

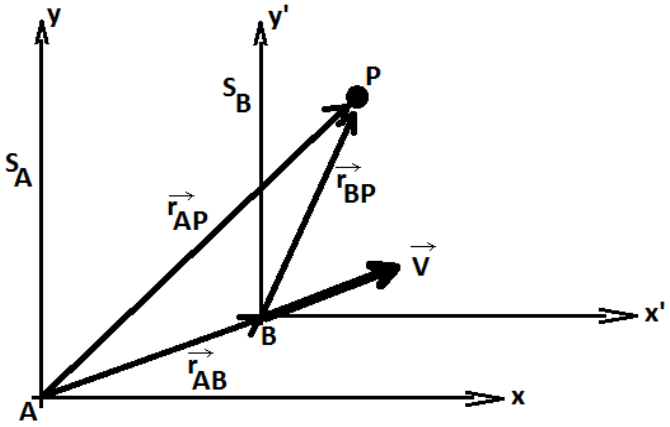


Figura 4.2: Movimento Relativo em duas dimensões.

As Equações 4.1.8, 4.1.9 e 4.1.10, correspondem à posição, à velocidade, e à aceleração da partícula P relativa ao referencial B, respectivamente.

$$\vec{r}_{BP}=\vec{r}_{AP}-\vec{r}_{AB} \qquad (4.1.8)$$

$$\vec{v}_{BP}=\vec{v}_{AP}-\vec{v}_{AB} \qquad (4.1.9)$$

$$\vec{a}_{BP}=\vec{a}_{AP} \qquad (4.1.10)$$

O produto instrucional desenvolvido nessa pesquisa corresponde a duas simulações referentes a dois exercícios do livro Fundamentos da Física Vol. 1 que foram aplicadas em aula e, como material de apoio, foi criado um manual de como utilizar a AcliveJS para o desenvolvimento de simulações para o Ensino de Física. O exercício do movimento relativo em uma dimensão visa a reforçar a explicação discutida a partir da Figura 4.1 e as Equações 4.1.1, 4.1.3 e 4.1.5. Já a simulação em duas dimensões retoma a análise sobre a trajetória relativa avaliada por Giordano Bruno no segundo exemplo do navio e permite trabalhar as Equações 4.1.8, 4.1.9 e 4.1.10.

4.1.1 Problema do Movimento Relativo em Uma Dimensão

O enunciado do problema proposto, extraído do Cap. 4 do livro Fundamentos da Física, volume 1, p. 90, problema 118 é apresentado a seguir:

Um aeroporto dispõe de uma esteira rolante para ajudar os passageiros a atravessar um longo corredor. Lauro não usa a esteira rolante e leva 150 s para atravessar o corredor. Cora, que fica parada na esteira rolante, cobre a mesma distância em 70 s. Marta prefere andar na esteira rolante. Quanto tempo leva para Marta atravessar o corredor? Suponha que Lauro e Marta caminhem com a mesma velocidade (Halliday et al., 2014).

Segue a solução analítica da questão. A Figura 4.3 ilustra o esquema que foi desenhado no quadro no momento de resolução. A esteira foi representada por uma base hachurada. Os nomes e velocidades correspondentes a cada componente do problema também estão indicados na figura.

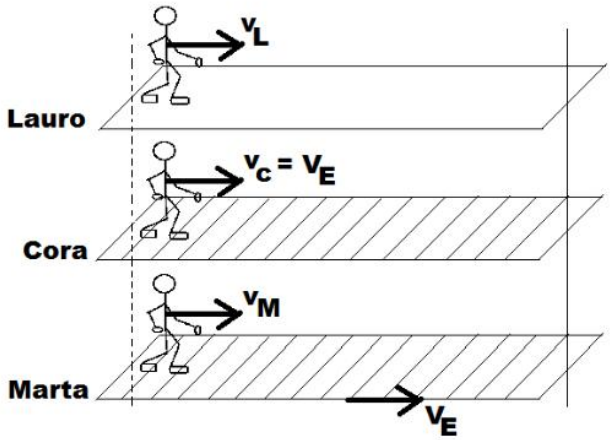


Figura 4.3: Desenho para auxiliar na resolução do problema 118.

Seguindo a notação utilizada na equação 4.1.9, as velocidades foram representadas obedecendo à mesma regra.

Assim:

$\vec{v}_L \equiv \vec{v}_{FL}$ (velocidade de Lauro relativo ao referencial fixo)

$\vec{v}_C \equiv \vec{v}_{FE}$ (velocidade da Esteira relativo ao referencial fixo)

De fato, como Cora fica apenas parada na esteira, então sua velocidade relativa à esteira é nula. Um observador parado no solo verá Cora se movendo com a mesma velocidade da esteira.

$\vec{v}_M \equiv \vec{v}_{EM}$ (velocidade de Marta relativa à esteira)

$\vec{v}_{FL} \equiv \vec{v}_{EM}$ (Lauro e Marta caminham com a mesma velocidade)

Será obtida a 4.1.11 após a substituição das velocidades utilizando a notação da equação 4.1.9.

$$\vec{v}_{EM} = \vec{v}_{FM} - \vec{v}_{FE} \qquad (4.1.11)$$

Nas três situações o movimento é uniforme, sendo assim, basta substituir cada rapidez pela equação da velocidade média do MU adotando um comprimento L para o tamanho do corredor. Dessa forma, chega-se a equação 4.1.12 cuja resolução fornece a resposta pra este problema, $\Delta t_M = 47{,}7\ s$.

$$\frac{1}{\Delta t_M} = \frac{1}{\Delta t_L} + \frac{1}{\Delta t_C} \qquad (4.1.12)$$

A Figura 4.3 mostra um instantâneo da simulação apresentada aos estudantes. Nela é possível observar a janela 2D para exibição de textos e das grandezas envolvidas no problema. Ao lado, aparece a janela 3D onde estão representadas as esteiras e os personagens que devem percorrê-las. Na parte superior, existem botões para pausar ou continuar a simulação, um botão para 'liberar' a câmera pelo movimento do mouse para melhor visualização da cena e instruções para uma visualização em tela cheia e para reiniciar a simulação. Na parte inferior encontra-se o enunciado do problema.

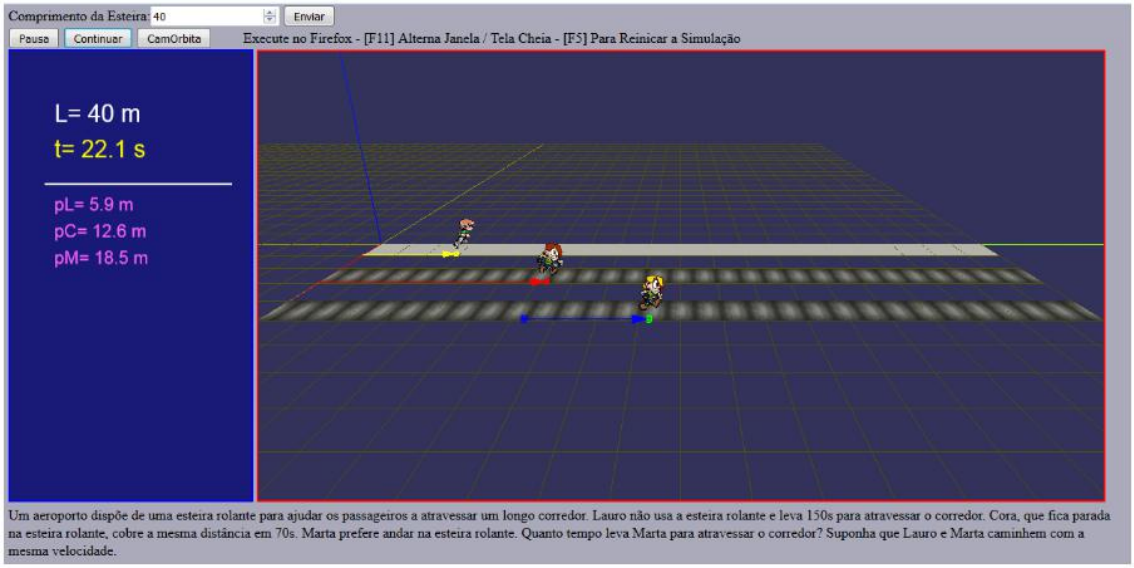


Figura 4.4: Simulação apresentada em sala referente ao problema 118.

Na simulação temos três personagens. Lauro anda pelo corredor sem esteira, representado na simulação na cor cinza claro. Cora e Marta estão sobre esteiras, onde foi atribuída uma textura que se move na direção x. O design gráfico da simulação lembra uma tela de videogame para atrair a atenção dos estudantes. Esta simulação exigiu a atenção e participação ativa dos alunos, pois o método utilizado foi o expositivo. No Apêndice 3 encontra-se a listagem completa desta simulação.

4.1.2 Problema do Movimento Relativo em Duas Dimensões

Segue o enunciado do segundo problema proposto:

> *A neve está caindo verticalmente com uma velocidade constante de 8 m/s. Com que ângulo, em relação a vertical, os flocos parecem estar caindo do ponto de vista do motorista de um carro que viaja em uma estrada plana e retilínea a uma velocidade de 50 km/h?* (Halliday et al., 2014, vol. 1, Cap. 4, p. 86, problema 77)

A Figura 4.5 ilustra o esquema que foi desenhado no quadro no momento de resolução.

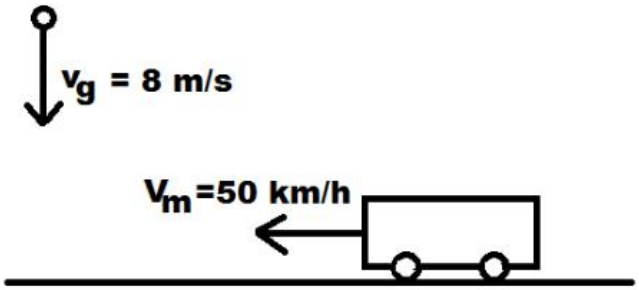


Figura 4.5: Desenho para auxiliar na resolução do problema 77.

Seguindo a notação utilizada na equação 5.1.4, as velocidades foram representadas obedecendo à mesma regra.

Assim:

$\vec{v}_M \equiv \vec{v}_{FM}$ (velocidade do motorista relativo ao referencial fixo)

$\vec{v}_N \equiv \vec{v}_{FN}$ (velocidade da neve relativo ao referencial fixo)

$\vec{v}_{MN}$ é a velocidade da neve relativo ao motorista)

Pelos dados do problema:

$$\left|\vec{v}_{FN}\right| = 8\,m/s \quad \text{e} \quad \left|\vec{v}_{FM}\right| = 13{,}9\,m/s \quad \text{(No S.I.)}$$

A questão pede para encontrar o ângulo da direção da queda da gota relativo ao motorista do carro. Utilizando a equação 5.1.4, o objeto 2 é a neve que cai e o objeto 1 o motorista. Ao fazer a substituição, será obtida a equação 5.1.7.

$$\vec{v}_{MN} = \vec{v}_{FN} - \vec{v}_{FM} \qquad (4.1.13)$$

Diferentemente do problema anterior, no qual foi possível 'abandonar' a notação vetorial pelo uso de sinais por se tratar de apenas uma dimensão, os vetores velocidades são perpendiculares, o que caracteriza um problema em duas dimensões. Portanto, deve ser representado através de um método geométrico, como por exemplo, o paralelogramo. A Figura 5.5 mostra o desenho que foi feito no quadro de escrever durante a resolução.

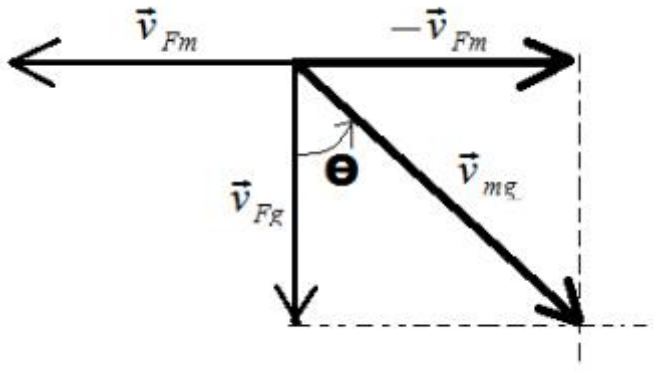


Figura 4.6: Representação dos vetores velocidades

Logo: $v_{MN} = \sqrt{v_{FM}^2 + v_{FN}^2}$, cujo resultado é: $v_{MN} = 16{,}1\,m/s$

Para o cálculo do ângulo:

$$\tan(\theta) = \frac{v_{FM}}{v_{FN}} \quad \text{, onde} \quad \theta = 60^o$$

As Figuras 4.7 e 4.8 mostram a simulação vista a partir de dois referenciais distintos, de um observador no solo e a de um observador no carro, respectivamente. Na Figura 4.8 é possível ver o resultado ao final da simulação, o ângulo de 60º formado pelo vetor velocidade do floco de neve relativo ao motorista do carro, em relação a direção vertical.

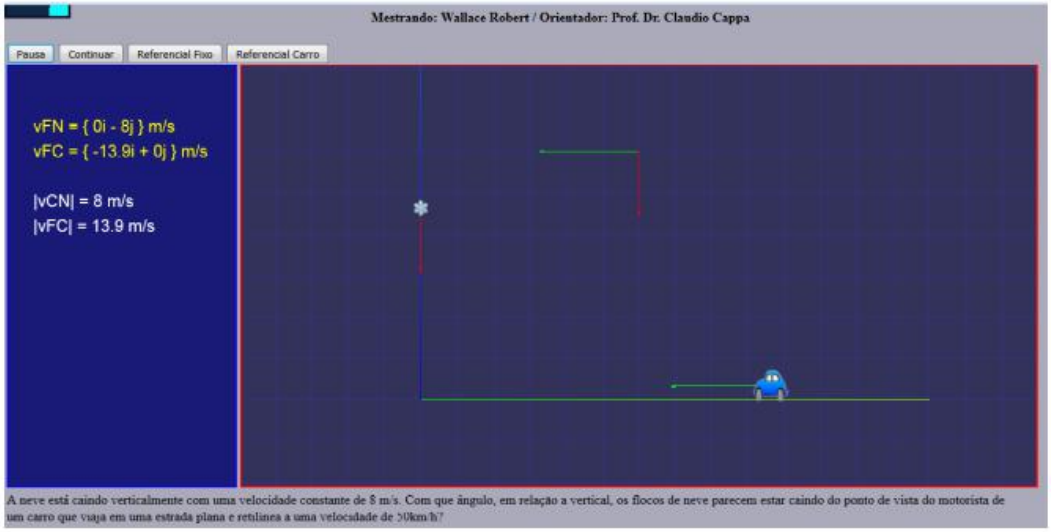


Figura 4.7: Simulação a partir do referencial ‘solo’.

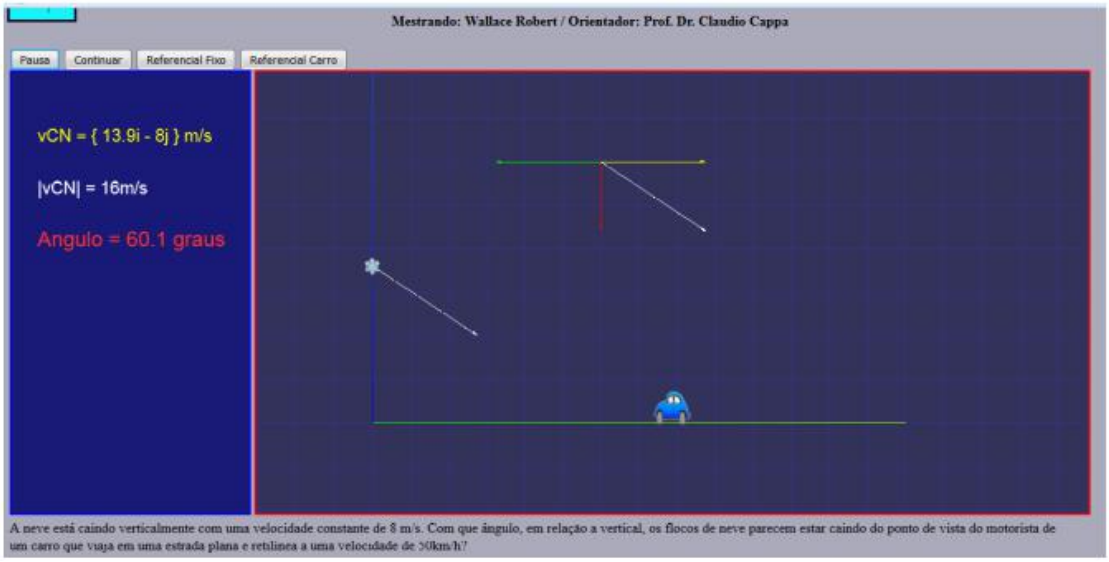


Figura 4.8: Simulação a partir do referencial ‘carro’.

É possível alternar entre um tipo de referencial ou outro durante a simulação, através do botão “referencial fixo” e “referencial carro”. Especificamente para esta simulação, durante a aplicação, muitos alunos não compreenderam esta alternância de referencial, entretanto, com a ajuda do professor foram capazes de abstrair o que estava

acontecendo e, com isso, entenderam o significado de um observador situado em locais distintos. No Apêndice 4 encontra-se a listagem completa desta simulação.

Capítulo 5

Resultados e Análise de dados:

Como visto anteriormente, na metodologia adotada para essa pesquisa, à medida que a biblioteca AcliveJS foi sendo construída, alguns professores foram convidados a participar do projeto para aprender a utilizar a ferramenta. Entretanto, apesar de acharem a iniciativa do projeto relevante e inicialmente mostrarem interesse, todos os convites foram recusados pelos professores, quando estes perceberam que deveriam utilizar uma linguagem de programação. Isso contraria, à primeira vista, a hipótese inicialmente levantada nessa pesquisa, de que seria possível aproveitar a habilidade de programar dos professores permitindo-os criar suas próprias simulações e que isto estimularia o aluno a aprender Física e favoreceria uma aprendizagem significativa. Porém, acredita-se que o número de professores convidados e o tempo disponível nesse processo foi pequeno e que, portanto, a amostra poderia ser ampliada em pesquisas futuras. Isso serve também para a conjectura sobre o aumento da interatividade do professor com o aluno através do uso da AcliveJS.

Desta forma, tendo em vista que a AcliveJS foi desenvolvida também para auxiliar no processo de abstração dos estudantes, o foco da pesquisa foi redirecionado para a aplicação de simulações sobre um tema específico com os alunos. Isso posto, diante da alteração metodológica adotada, são apresentadas, a seguir, os resultados e análise dos dados obtidos a partir da aula ministrada para os alunos de graduação do curso de Engenharia.

A análise dos dados consistiu em verificar a receptividade das simulações elaboradas utilizando a AcliveJS. No total, 15 estudantes responderam às questões sugeridas. O Gráfico 4.1 mostra a aceitação do uso deste tipo de Objeto de Aprendizagem em sala de aula.

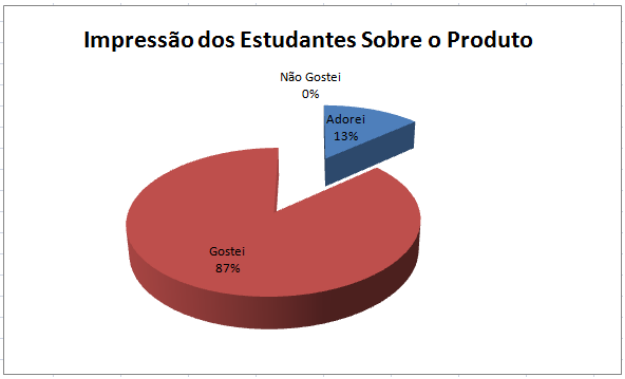


Gráfico 4.1: Gráfico da impressão dos estudantes sobre o produto.

O gráfico mostra que 87% dos estudantes gostaram desse tipo de OED e 13% disseram que adoraram o material educacional apresentado no formato de simulações. Este resultado indica que a aula utilizando este tipo de recurso pode atrair a atenção dos estudantes, já que na maioria das vezes, os mesmos estão expostos apenas às aulas tradicionais. É importante ressaltar que os alunos tiveram acesso apenas a uma parte produto instrucional, que consistia nas simulações sobre o movimento relativo. Não foi apresentado aos educandos o manual de como utilizar a AcliveJS, entretanto, foi sugerido que visitassem a página do projeto no repositório GitHub para obterem informações técnicas sobre o seu uso. A parte impressa do produto instrucional, consiste no manual presente no Apêndice 5, que explora os requisitos técnicos necessários para construções de simulações voltadas para o Ensino de Física utilizando a AcliveJS.

Alguns comentários apontam para a relevância do uso deste tipo de OED, tais como os transcritos abaixo:

> *"Noções de movimento, direção, sentido e ângulo, por exemplo, ficaram bem mais claras." (1)*
>
> *"Perfeito como uma ferramenta auxiliar, porém não dispensa a explicação prévia do professor." (2)*
>
> *"A aula com as simulações facilitaram no acompanhamento do raciocínio junto com o professor." (3)*
>
> *"A aula com as simulações são mais interessantes, tendo em vista que ajuda na abstração." (4)*
>
> *"A aula com as simulações são mais ilustrativas. Facilitando a compreensão. Além do mais, é algo novo, diferente do que estão acostumados." (5)*
>
> *"Facilita a compreensão das questões e pontos de análises de um problema. Ajuda na aprendizagem de lógica de programação." (6)*
>
> *"Ferramenta oferece muito dinamismo e ilustra, de fato, como tudo ocorre em termos práticos." (7)*

As respostas dos alunos mostraram que as atenções se voltaram para a aula em geral, ou seja, a utilização do material instrucional despertou o interesse pelo método 'diferente' de ensino aplicado com o uso do produto instrucional (comentários 3,4,5 e 6). Dois comentários, o 2 e o 7, indicam o interesse pela ferramenta utilizada na construção das simulações, a AcliveJS. É importante ressaltar o segundo comentário, em que o aluno compreende que as simulações por si só não dispensam a presença do professor, o que foi discutido no capítulo 2. Apenas o comentário número 1 faz alusão

ao movimento relativo, onde o aluno parece entender alguns pontos do conteúdo antes não compreendidos. Como as respostas eram optativas, 8 alunos não emitiram suas opiniões por escrito referente a esta pergunta.

A Tabela 4.1 apresenta as respostas transcritas referentes às resoluções dos problemas propostos comparando a aula tradicional com a aula utilizando o OED.

Aula tradicional	Aula utilizando o OED
"Difícil visualização do espaço 3D. Objetos estáticos."	*"Dá pra ver os movimentos dos corpos."*
"Num ambiente 3D, a dinâmica vetorial pode ficar muito abstrata."	*"Prático de visualizar os gráficos e entender melhor o que o problema propõe."*
"Fico tentando abstrair e acabo me distraindo."	*"Achei que a aula ficou mais interativa e prestei mais atenção do que o habitual."*
"Certas vezes, a aula pode parecer corrida em meio a tantos números e funções, o que dificulta a transferência do problema para o mundo real."	*"Achei interessante poder associar problemas com a animação do programa. Forneceu um melhor entendimento do que estava acontecendo."*
"Em uma dimensão é fácil entender a teoria, mas complicado entender o movimento."	*"Fácil observação da Física acontecendo quando o simulador aplica as leis, auxiliando o entendimento da matéria."*
-	*"Bem mais ilustrativo, facilitando a compreensão. Além do mais representa algo novo!"*

Tabela 4.1: Respostas dos alunos ao compararem a aula tradicional e a aula utilizando o produto instrucional.

É possível notar que a maior dificuldade da aula tradicional está em compreender o que está ocorrendo devido à falta de capacidade de abstração, ou seja, os estudantes não conseguem visualizar o problema mesmo com desenhos feitos no quadro. Comparado com o uso do OED AcliveJS, a aula tradicional apresenta-se como 'mais difícil de ser compreendida'.

Finalmente, foram obtidos os dados sobre os prós e contras relativos ao uso do produto em sala. Em relação a esta questão, foi questionado aos estudantes se haveria interesse deles em produzir as próprias simulações utilizando a AcliveJS.

Na Tabela 4.2 é possível observar algumas destas impressões.

Prós	Contras
"Facilita a visualização. Alguns professores não possuem a habilidade didática para mostrar com clareza os fenômenos apenas utilizando o quadro."	*"É preciso entender de programação para manusear o programa."*
"Auxilia a aula teórica, ficando fácil o entendimento do exercício, ampliando o conhecimento."	*"Conhecimentos específicos de programação."*
"Tornou a aula mais dinâmica e facilitou o entendimento devido ao tipo visualização."	*"Desestimula a escrita (copiar), pois é uma das maneiras que utilizo para fixação do conhecimento."*
"Achei muito legal, e fez com que eu, particularmente, entendesse as questões, o que não ocorreu quando foi usado apenas o quadro."	*"Achei o fundo escuro, não dava para observar bem o contraste das cores dos vetores com o fundo escolhido."*
"Ajudou a entender como raciocinar na resolução das questões."	*"Seria mais fácil disponibilizar a ferramenta já com uma gama de simulações prontas, pois poucos seriam aqueles que gostariam de programar suas próprias simulações."*
"Didático, pois com esta interatividade aproxima mais o aluno da realidade. Melhor aproveitamento da aula."	*"Dificuldade de utilização devido a parte da programação."*
"Visualização mais clara."	-

Tabela 4.2: Prós e contras do uso do OED em aula.

A tabela acima aponta para percepções favoráveis ao uso da ferramenta, quando os alunos citam principalmente o entendimento da teoria através da visualização das ações do movimento relativo. Outras expressões que indicam o aumento da aprendizagem via ferramenta são citadas, tais como clareza, ampliação do conhecimento, dinamismo da aula, didática. Tais dados remetem à importância de promover aulas mais dinâmicas e com foco nos fenômenos em vez de na matemática, como bem aponta Gomes et al. (2010). Outros alunos apontam a interatividade e a aproximação do aluno da realidade, em detrimento das ferramentas utilizadas na aula tradicional, o que pode indicar a aquisição e a melhoria do processo de aprendizagem com a utilização da OED AcliveJS, já que os alunos se tornariam agentes ativos na construção do conhecimento.

Outras impressões favoráveis em relação ao produto instrucional aplicado em aula apontam para a facilidade que este tipo de Objeto Instrucional trouxe para a abstração dos estudantes, quando comparado a aula tradicional. Além disso, reforça o auxílio que o produto instrucional promoveu em facilitar a abstração dos alunos, em relação às demonstrações matemáticas no contexto da Física, quando comparado à aula tradicional.

Em relação aos contras, a maioria dos estudantes mostrou-se avessa à possibilidade de utilizar a ferramenta para desenvolver uma simulação devido ao fato de terem de programar. Entretanto, por se tratar de uma ferramenta de código aberto e disponível na internet, espera-se que novas contribuições aumentem o número de simulações disponíveis como exemplos para que possam ser utilizadas por professores que não possuem interesse em desenvolverem os próprios programas. De acordo com as respostas apresentadas na Tabela 4.2, criar simulações utilizando linhas de código não atraiu os estudantes apesar de demonstrarem interesse por este tipo de aula.

Em termos gerais, o OED e a ferramenta utilizada para desenvolvê-la, no caso a AcliveJS, teve uma receptividade boa. O produto instrucional foi elaborado utilizando a versão 0.10 da AcliveJS.

Capítulo 6

Perspectivas Futuras e Considerações Finais

Neste trabalho foi constante a preocupação em desenvolver um produto instrucional que desse oportunidade aos estudantes para compreender o movimento relativo em uma e duas dimensões, por meio de simulações computacionais. A a biblioteca AcliveJS foi criada para atender as necessidades deste projeto e para que alunos e professores pudessem criar as próprias simulações, utilizando recursos avançados do *HTML5* e *WebGL* por meio de comandos em português. A abrangência do uso do computador na escola busca promover a aprendizagem dos alunos, ajuda na construção dos conceitos físicos e permite estimular um processo de ensino aprendizagem cada vez mais crítico. Disponibilizar uma ferramenta de desenvolvimento de simulações por meio de linhas de código com os principais comandos em português, visa a atrair a atenção dos professores para possibilidades de ensino mais interativo, dinâmico, no qual as aulas poderão ser trabalhadas com uma ferramenta que oferece recursos avançados de entrada, controle e exibições em duas e três dimensões e que podem ser utilizadas na web.

A biblioteca AcliveJS é uma ferramenta gratuita e de código aberto, portanto, espera-se que contribuições futuras sejam cada vez mais frequentes e que possam ser expandidas suas funcionalidades, e também o número de simulações exemplos, por meio de contribuições dos usuários. Algumas dessas expansões poderão ser janelas para exibição de gráficos em duas e três dimensões, elaboração de comandos para integração numérica de funções matemáticas, uso da câmera do computador para captura de movimento, entre outras. Em versões futuras, as simulações desenvolvidas com a AcliveJS poderão ser executadas diretamente em tablets e smarthphones já que tratam-se de páginas *HTML5* que executam simulações via OpenGL ES, largamente utilizada nestes tipos de aparelhos.

Quanto às questões propostas na presente pesquisa, no que se refere a construção de uma biblioteca de comandos para o desenvolvimento de simulações, a AcliveJS mostrou-se como uma ferramenta que permite a construção de programas voltados para o Ensino de Física. Os fatores que poderiam facilitar ou dificultar o uso e elaboração de simulações em sala de aula, foi discutida no capítulo 5 a partir das impressões que os estudantes tiveram sobre a ferramenta ao responder as fichas. A partir da análise das respostas, viu-se que o OED AcliveJS apresenta-se como um produto que despertou o

interesse e que facilita na abstração auxiliando no processo de ensino e aprendizagem no Ensino de Física. Deve-se ter em mente que o produto instrucional de um modo geral atende apenas aos educadores e estudantes que possuem acesso ao computador. Sem o acesso ao computador ou internet, outros recursos didáticos devem ser utilizados para enriquecer as aulas. Apesar de não ter sido utilizado diretamente com os alunos, o manual presente no Apêndice 3 foi elaborado como parte do produto instrucional para atender às necessidades dos usuários interessados em desenvolver simulações. Esta pesquisa pretende abrir novas possibilidades para que o educador ou educando possa aprender ou aperfeiçoar o processo de construção de uma simulação, com o objetivo de desenvolver os próprios programas para adquirir conhecimento ou ensinar Física atendendo necessidades específicas.

Tendo isto em vista, espera-se que a AcliveJS possa vir a se tornar uma ferramenta computacional relevante, tanto no apoio ao ensino de Física como na pesquisa em que se faz presente o uso de modelagem computacional.

Referências Bibliográficas

ARAUJO, I. S.; VEIT, E. A.; MOREIRA, M. A. Uma revisão da literatura sobre estudos relativos a tecnologias computacionais no ensino de física. **Rev. Bras. de Pesquisa em Educação em Ciências**, Belo Horizonte, v. 4, n. 3, p. 5–18, 2004. Disponível em:

<https://seer.ufmg.br/index.php/rbpec/article/view/2270/1669> Acesso em: 14 maio 2017

ARAUJO, I. S.; VEIT, E. A.; MOREIRA, M. A. Atividades de modelagem computacional no auxílio à interpretação de gráficos da Cinemática. **Rev. Bras. Ensino Fís.**, São Paulo, v. 26, n. 2, p. 179-184, 2004. Disponível em: <http://www.scielo.br/scielo.php?script=sci_ arttext&pid

=S1806-11172004000200013& lng=en&nrm=iso> Acesso em: 14 maio 2017

AUSUBEL. D. P. **Educational Psychology:** a cognitive view. New York: Holt, Rinechart and Winston, Inc. 1968.

AUSUBEL. D. P. **Aquisição e retenção de conhecimentos:** uma perspectiva cognitiva. Lisboa: Plátano Edições Técnicas, 2003.

BASSALO. J. M. F. Aspectos históricos das bases conceituais das relatividades. In: **Rev. Bras. Ens. Física,** São Paulo, v. 19, n. 2, jun. 1997. Disponível em: < http://www.sbfisica.org.br/rbef/pdf/v19_180.pdf> Acesso em: 17 maio 2017

BARROSO, M. F.; BEVILAQUA, D. V.; FELIPE, G. Visualização e interatividade no ensino de Física e a produção de aplicativos computacionais. In: XVIII Simpósio Nacional de Ensino de Física, 2009, Vitória. **Anais eletrônicos do XVIII SNEF**. São Paulo: Sociedade Brasileira de Física, 2009. Disponível em: <http://www.sbf1.sbfisica.org.br/eventos/snef/xviii/sys/

resumos/T0082-1.pdf> Acesso em: 17 maio 2017

BARROSO, M. F.; FELIPE, G.; SILVA, T. Aplicativos computacionais e ensino de Física. In: X Encontro de Pesquisa em Ensino de Física, 2006, Londrina. **Anais eletrônicos do X EPEF.** São Paulo: Sociedade Brasileira de Física, 2006. Disponível em: <http://www.if.ufrj.br/~marta/artigosetal/2006-epef10-aplicativos.pdf> Acesso em: 17 maio 2017

COSTA, F. A. (coord.). **Repensar as TIC na educação:** o professor como agente transformador. 2012. (Coleção Educação em Análise). Disponível em: <https://www.researchgate.net/ publication/299455917 Repensar_as_TIC_na_Educacao

_ _O_Professor_como_Agente_Transformador> Acesso em: 20 maio 2017

FEYNMAN, R. P.; LEIGHTON, R. B.; SANDS, M. **Feynman:** lições de Física. v. 1. Porto Alegre: Bookman, 2008.

FIGUEIRA, J. S. Easy Java simulations: Modelagem computacional para o ensino de Física. **Rev. Bras. de Ensino de Física,** v. 27, n. 4, p. 613-618, São Paulo, 2005. Disponível em: <http://www.sbfisica.org.br/rbef/pdf/v27_613.pdf> Acesso em: 14 maio 2017

GOMES, J. C.; CASTILHO, W. S. Uma visão de como a Física é ensinada na escola brasileira e a experimentação como estratégia para mudar essa realidade. In: 1ª Jornada de Iniciação Científica e Extensão do IFTO, 2010, Tocantins. **Anais eletrônicos da 1ª Jornada de Iniciação Científica e Extensão do IFTO.** Tocantins, 2010. Disponível em: <http://www.ifto.edu.br/jornadacientifica/wp-content/uploads/2010/12/

12-UMA-VIS%C3%83O.pdf> Acesso em: 02 junho 2017

HALLIDAY, D.; RESNICK, R.; WALKER, J. **Fundamentos de Física:** mecânica. 8. ed. Rio de Janeiro: LTC, 2014.

LIBÂNEO, J. C. **Didática.** São Paulo: Cortez, 1994.

LIBÂNEO, J. C. O dualismo perverso da escola pública brasileira: escola do conhecimento para os ricos, escola do acolhimento social para os pobres. **Educ. Pesqui.**, São Paulo, v. 38,n. 1,p. 13-28, Mar. 2012 . Disponível em: <http://www.scielo.br/scielo.php? script=sci_arttext&pid=S1517-97022012000100002&lng=en&nrm=iso>. Acesso em: 22 maio 2017

LUCKESI, C. C. **Avaliação da aprendizagem escolar:** estudos e proposições. 19. ed. São Paulo: Cortez, 2008.

MACHADO, A. F.; COSTA, L. M. A utilização do software MODELLUS no ensino da Física. **Interagir: pensando a extensão**, Rio de Janeiro, n. 14, p. 45-50, jan./dez. 2009. Disponível em: <http://www.e-publicacoes.uerj.br/index.php/interagir/article/view/1814/1383> Acesso em: 14 maio 2017

MENDES, E. S. **Modelagem computacional em Física usando o software Modellus:** uma abordagem alternativa no ensino de cinemática. 2014. 157 p. Dissertação

(Mestrado Profissional em Ensino de Ciências Exatas). UNIVATES, Lageado, RS. 2014.

MENDES, J. F.; COSTA, I. F.; DE SOUSA, C. M. S. G. O uso do software Modellus na integração entre conhecimentos teóricos e atividades experimentais de tópicos de mecânica. **Rev. Bras. Ensino Fís.**, São Paulo, v. 34, n. 2, p. 1-9, Jun. 2012. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S1806-11172012000200011&lng=en&nrm=iso>. Acesso em 13 de maio 2017.

MOREIRA, M. A. **Mapas conceituais e aprendizagem significativa.** São Paulo: Centauro Editora, 2010.

MOREIRA, M. A. ¿Al afinal, qué es aprendizaje siginificativo? **Qurriculum:** revista de teoría, investigación y práctica educativa. La Laguna, Espanha. n. 25, mar. 2012, p. 29-56. Disponível em: <http://publica.webs.ull.es/ upload/REV%20QURRICUL

UM/25%20-%202012/02.pdf> Acesso em: 05 junho 2017

POCRIFKA, D. H., SANTOS, T. W. Linguagem logo e a construção do conhecimento. In: IX Congresso Nacional de Educação – EDUCERE, 2009, Curitiba. Anais... Paraná: PUCPR, 2009. p. 2469-2479. Disponível em: <http://www.pucpr.br/eventos/educere/educere2009 /anais/pdf/2980_1303.pdf> Acesso em: 28 maio 2017

SANTOS, E. I.; PIASSI, L. P. C.; FERREIRA, N. C. Atividades experimentais de baixo custo como estratégia de construção da autonomia de professores de física: uma experiência em formação continuada. In: IX Encontro Nacional de Pesquisa em Ensino de Física, 2004, São Paulo. **Anais...** São Paulo: Instituto de Física da USP, Faculdade de Educação da USP, 2004. p. 1-18. Disponível em: <http://www.sbf1.sbfisica.org.br/eventos/epef/ix/sys/resumos/T0058-1.pdf> Acesso em: 28 maio 2017

SARAIVA-NEVES, M.; CABALLERO, C.; MOREIRA, M. A. Repensando o papel do trabalho experimental, na aprendizagem da física em sala de aula: um estudo exploratório. **Investigações em Ensino de Ciências**, Porto Alegre, v. 11, n. 3, p. 383-401, 2006. Disponível em: <http://www.lume.ufrgs.br/handle/10183/141761>. Acesso em: 02 junho 2017

SHERWOOD, B.; CHABAY, R. VPython: 3D programming for ordinary mortals. In: Multimedia in Physics Teaching and Learning MPTL 14 International Workshop, 2009. **Anais...** Udine: University of Udine, Italy, Sep. 2009. p. 1-3. Disponível em:

<http://www.fisica.uniud.it/URDF/mptl14/ftp/full_text/WS3%20Full%20Paper.pdf> Acesso em: 16 maio 2017

SILVA, J. R.; GERMANO, J. S. E.; MARIANO, R. S. SimQuest: ferramenta de modelagem computacional para o ensino de física. **Rev. Bras. Ensino Fís.**, São Paulo, v. 33, n. 1, p. 01-08, mar. 2011.

SOUZA FILHO, C. A.; RAMALHO, E. F. R.; SILVA, A. O.; GOMES, J. L.; OLIVEIRA J. R. R. Uso do Python como laboratório virtual na Física. In: IX Jornada de Ensino, Pesquisa e Extensão (JEPEX), Centro de Ensino de Graduação, 2009, Recife. **Resumos...** Pernambuco: UFRPE, 2009. p. 1-3. Disponível em: <http://www.eventosufrpe.com.br/ jepex2009/cd/resumos/R0966-1.pdf> Acesso em: 16 maio 2017

STUDART, N.; ARANTES, A. R.; MIRANDA, M. S. Objetos de aprendizagem no ensino de física: usando simulações do PhET. **Física na Escola**, São Paulo, v. 11, n. 1, p. 27-31, 2010. Disponível em: <http://www1.fisica.org.br/fne/index.php/edicoes/category/9-n-1-abril>. Acesso em: 20 maio 2017

STUDART, N. Simulação, games e gamificação no ensino de Física. In: **XXI Simpósio Nacional de Ensino de Física**, jan. 2015, Uberlândia, MG. Uberlândia: UFU, 2015. Disponível em:<http://doczz.com.br/doc/16546/simula%C3%A7%C3%A3o--

games-e-gamifica%C3%A7%C3%A3o-no-ensino-de-f%C3%ADsica> Acesso em: 20 maio 2017

TAYLOR, J. R. **Mecânica Clássica.** Porto Alegre: Bookman, 2013.

VALADARES, J.; MOREIRA, M. A. **A teoria da aprendizagem significativa:** sua fundamentação e implementação. Coimbra: Edições Almedina, 2009.

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());
}
```

Apêndice 3

```
// Declara as variáveis do Problema (fora das functions Ambiente e Simulação)

var pL, pC, pM;

var vFL, vFC, vFM;

var vEC, vEM;

var vFE;

var L;

var t = new Cronometro();

var offsetyL, offsetyC, offsetyM;

function Ambiente()

{

        // Atribuir valores iniciais (obrigatoriamente dentro da function Ambiente)

        offsetyL=0; offsetyC=0; offsetyM=0;

        L = parseFloat(document.Form1.L.value);

        t =  clock.start();

        vFL = L/150;

        vFE = L/70;

        vEC = 0;

        vEM = vFL;

        pL = 0, pC = 0, pM = 0

        vFC = vEC + vFE;

        vFM = vEM + vFE;

        // Inicia a Engine

        Janela3D('rgb(50,50,90)',0.7,0.8)

        InserirObservador(45,0.1,20000)

        PosicaoDoObservador(30,20,10);

        OlharPara(-70,20,-30);

        SistemaRetangular(50,false,false,false);

        GradeXY('rgb(150,150,0)','rgb(80,80,20)',L,2,false);
```

```
        Desempenho(true,false,false);
        Janela2D('rgb(25,25,120)',0.2,0.8)
                // Cria as esteiras
                BLauro = new Plano(2, L, 1, 10, 1, L/2, 0, 0, 0, 0, 'rgb(180,180,170)',true,false);
                BCora = new PlanoT('imagens/esteira.jpg', 2, L, 1, 10, 5, L/2, 0, 0, 0, 0, true,false);
                BMarta = new PlanoT('imagens/esteira.jpg', 2, L, 1, 10, 9, L/2, 0, 0, 0, 0, true,false);
                // Insere os personagens
                 Lauro = new Sprite("imagens/Lauro.png", 2, 2, 1, 0, 1, 8, 1, 8, 200);
                 Cora = new Sprite("imagens/Cora.png", 2, 2, 5, 0, 1, 6, 1, 6, 150);
                 Marta = new Sprite("imagens/Marta.png", 2, 2, 9, 0, 1, 6, 1, 6, 80);
                 // Insere os dummies (caixinhas coloridas de controle)
                 DLauro = new Caixa(0.2, 0.2, 0.2, 2, 0, 0.1, 0, 0, 0, 'rgb(255,255,0)', true, 1);
                 DCora = new Caixa(0.2, 0.2, 0.2, 6, 0, 0.1, 0, 0, 0, 'rgb(255,0,0)', true, 1);
                 DMarta = new Caixa(0.2, 0.2, 0.2, 10, 0, 0.1, 0, 0, 0, 'rgb(0,255,0)', true, 1);
                 DEsteira = new Caixa(0.2, 0.2, 0.2, 10, 0, 0.1, 0, 0, 0, 'rgb(0,0,255)',true, 1);
                 // Cria os vetores
                 PosicaoL = new Vetor(2, 0, 0.1, 0, pL, 0, 1, 'rgb(255,255,0)');

                 PosicaoC = new Vetor(6, 0, 0.1, 0, pC, 0, 1, 'rgb(255,0,0)');
                 PosicaoM = new Vetor(10, 0, 0.1, 0, pM, 0, 1, 'rgb(0,0,255)');
Pausa();
}
function Simulacao()
{
        if (pM<L)
                {
                        t = parseFloat(clock.getElapsedTime());
                        pL = round(vFL*t,1);
                        pC = round(vFC*t,1);
                        pM = round(vFM*t,1);
```

```
        }
        else
        {
                Pausa()
        }

// Atualiza as posições dos Players  (também serve: Nome.Player.position.set(x,y,z);
Lauro.Player.position.y=pL;
Cora.Player.position.y=pC;
Marta.Player.position.y=pM;

// Anima os Players
Lauro.Anima.update(15);
Marta.Anima.update(8);

// Anima as esteiras (tambem serve: Nome.TexturaPlano.offset.set(x,y);
BCora.TexturaPlano.offset.y=offsetyC-=0.008;
BMarta.TexturaPlano.offset.y=offsetyM-=0.008;
// Atualiza as posições dos Dummies (também serve: Nome.Caixa.set(x,y,z);
DLauro.Caixa.position.y=pL;
DCora.Caixa.position.y=pC;
DMarta.Caixa.position.y=pM;
DEsteira.Caixa.position.y=pC;

// Atualiza Vetores
UpdateVetor(PosicaoL);  PosicaoL = new Vetor(2, 0, 0.1,  0, pL, 0, 1, 'rgb(255,255,0)');
UpdateVetor(PosicaoC); PosicaoC = new Vetor(6, 0, 0.1, 0, pC, 0, 1, 'rgb(255,0,0)');
UpdateVetor(PosicaoM); PosicaoM = new Vetor(10, pC, 0.1, 0, pM-pC, 0, 1, 'rgb(0,0,255)');

// Exibe o tempo e posições de Lauro, Cora e Marta
```

```
        Retangulo(0,0,600,1000,'rgb(25,25,120');
        Texto ("L= "+round(L,1)+" m", 50, 80, 'rgb(255,255,255)', 20,1);
        Texto ("t= "+round(t,1)+" s", 50, 120,'rgb(255,255,0)',20,1);
        Texto ("pL= "+pL+" m", 50, 180,'rgb(255,100,255)',15,0);
        Texto ("pC= "+pC+" m", 50, 210,'rgb(255,100,255)',15,0);
        Texto ("pM= "+pM+" m", 50, 240,'rgb(255,100,255)',15,0);
        Linha2D (40,150,250,150,2,'rgb(255,255,255)',reta);
}
```

Apêndice 4

```
// Declara as variáveis do Problema (fora das functions Ambiente e Simulação)

var pNx, pNy, pNz;

var pCx, pCy, pCz;

var old_pC;

var vFN, vFNy, vFNz;

var vFC, vFCy, vFCz;

var vCN; vCNy, vCNz;

var t;

var angulo;

var RefFixo;

function Ambiente()

{

        // Atribuir valores iniciais (obrigatoriamente dentro da function Ambiente)

        RefFixo=true;

        pNx=0; pNy=0; pNz=30;

        pCx=0; pCy=60; pCz=1.7;

        vFNy=0; vFNz=8.0; vFN = 8.0;

        vFCy=50.0/3.6; vFCz=0; vFC = 50.0/3.6;

        vCNy= -vFCy; vCNz=vFNz; vCN = Math.sqrt(Math.pow(vFN, 2) + Math.pow(vFC, 2));

        angulo = Math.atan(vFC/vFN)*(180/Math.PI);

        t = clock.start();

        // Inicia a Engine

        Janela3D('rgb(50,50,90)',0.7,0.8)

        InserirObservador(45,0.1,20000)

        PosicaoDoObservador(30,20,10);

        Ortografica(0,30,15,true,false,false,25);

        GradeYZ('rgb(50,50,150)','rgb(50,50,150)',200000,5,false);

        SistemaRetangular(70,false,false,false);
```

```
        Desempenho(true,false,false);

        Janela2D('rgb(25,25,120)',0.2,0.8);

        // Insere os Sprites na cena

        Neve = new SpriteB('imagens/neve.png', 2, 2, pNx, pNy, pNz);

        Carro = new SpriteB('imagens/carro.png',5,5,pCx,pCy,pCz);

        // Cria os vetores

        Vel_FN = new Vetor(0,0,pNz,0,0,-vFNz,1,'rgb(255,0,0)');

        Vel_FC = new Vetor(0,pCy,2.5,0,-vFCy,0,1,'rgb(0,255,0)');

        Dummie_FN = new Vetor(0,30,30, 0, 0, -vFNz, 1,'rgb(255,0,0)');

        Dummie_FC = new Vetor(0,30,30, 0, -vFCy, 0, 1, 'rgb(0,255,0)');

        Dummie_SFC = new Vetor(0,30,30, 0, +vFCy, 0, 1, 'rgb(255,255,0)');
        UpdateVetor(Dummie_SFC);

        Dummie_CN = new Vetor(0,30,30, 0,+vFCy, -vFNz, 1, 'rgb(255,255,255)');
        UpdateVetor(Dummie_CN);

        Dummie_CN2 = new Vetor(0,30,30, 0,+vFCy, -vFNz, 1, 'rgb(255,255,255)');
        UpdateVetor(Dummie_CN2);

        Pausa();

}

function Simulacao()

{

        if (pNz>0)

        {

                t = parseFloat(clock.getElapsedTime()/1000);

                pNz = round(pNz – vFNz*t,2);

                pCy = round(pCy – vFCy*t,2);

        }

        else {Pausa();}

        if (RefFixo)

        {

        // Atualiza as posições dos sprites

        Neve.SpriteB.position.set(pNx, pNy, pNz);
```

```
        Carro.SpriteB.position.set(pCx,pCy,pCz);

        // Atualiza os vetores

        UpdateVetor(Dummie_SFC); UpdateVetor(Dummie_CN);
        UpdateVetor(Dummie_CN2);

        UpdateVetor(Vel_FN); Vel_FN = new Vetor(pNx,pNy,pNz,0,0,-vFNz,1,'rgb(255,0,0)');

        UpdateVetor(Vel_FC); Vel_FC = new Vetor(pCx,pCy,pCz,0,-vFCy,0,1,'rgb(0,255,0)');

        UpdateVetor(Dummie_FC); Dummie_FC = new Vetor(0,30,30, 0, -vFCy, 0, 1, 'rgb(0,255,0)');

        UpdateVetor(Dummie_FN); Dummie_FN = new Vetor(0,30,30, 0, 0, -vFNz, 1,'rgb(255,0,0)');

        // Exibe o tempo e as velocidades

        Retangulo(0,0,600,1000,'rgb(25,25,120');

Texto ("vFN = { "+round(vFNy,1)+"i - "+round(vFNz,1)+"j } m/s", 30, 80, 'rgb(255,255,0', 15, 1);

Texto ("vFC = { -"+round(vFCy,1)+"i + "+round(vFCz,1)+"j } m/s", 30, 110, 'rgb(255,255,0)', 15, 1);

Texto ("|vCN| = "+round(vFN,1)+" m/s", 30, 170, 'rgb(255,255,255)', 15, 1);

Texto ("|vFC| = "+round(vFC,1)+" m/s", 30, 200, 'rgb(255,255,255)',15,1);      UpdateOrtografica();
        }

        else

        {

        // Atualiza as posições dos sprites

        Neve.SpriteB.position.set(pNx,pNy,pNz);

        Carro.SpriteB.position.y=pCy;

        camera.position.set(1,pCy,15);

UpdateVetor(Dummie_FC); UpdateVetor(Vel_FN); UpdateVetor(Vel_FC);

UpdateVetor(Dummie_SFC); Dummie_SFC = new Vetor(0,30,30, 0, +vFCy, 0, 1, 'rgb(255,255,0)');

UpdateVetor(Dummie_CN); Dummie_CN = new Vetor(0,30,30, 0,+vFC, -vFNz, 1, 'rgb(255,255,255)');

UpdateVetor(Dummie_FN); Dummie_FN = new Vetor(0,30,30, 0, 0, -vFNz, 1,'rgb(255,0,0)');

UpdateVetor(Dummie_CN2); Dummie_CN2 = new Vetor(pNx,pNy,pNz, 0,+vFCy, -vFNz, 1,
'rgb(255,255,255)');

UpdateVetor(Dummie_FC); Dummie_FC = new Vetor(0,30,30, 0, -vFCy, 0, 1, 'rgb(0,255,0)');

// Exibe o tempo e as velocidades

Retangulo(0,0,600,1000,'rgb(25,25,120');

Texto ("vCN = { "+round(vFCy,1)+"i - "+round(vCNz,1)+"j } m/s", 30, 80, 'rgb(255,255,0)', 15, 1);
```

```
Texto ("|vCN| = "+round(vCN,1)+ "m/s", 30, 140, 'rgb(255,255,255)', 15, 1);

Texto ("Angulo = "+round(angulo,1)+" graus", 30, 200, 'rgb(255,50,50)',18,1);

}

}

function RF() {RefFixo=true;}

function RC() {RefFixo=false;}
```

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....88

2 COMO A ACLIVEJS FUNCIONA....90

2.1 A Estrutura de uma Simulação....93
2.2 Inserindo Textos e Imagens....96
2.3 Construindo Modelos com a AcliveJS....97
2.3.1 Escolha do Modelo: Movimento Harmônico Simples (MHS)....98
2.3.2 A Física do MHS....99
2.3.3 Construindo a cena em Ambiente()....101
2.3.4 Equações de Evolução....107
2.3.5 Objetos Complementares....110

3 VISITANDO A PÁGINA DO PROJETO....113

3.1 Arquivo LEIA-ME PRIMEIRO.txt....114

4 O AMBIENTE DE DESENVOLVIMENTO INTEGRADO (IDE)....117

4.1 NotePad++....117
4.2 Sublime....118
4.3 ConText....118

5 INTRODUÇÃO AO JAVASCRIPT....120

5.1 Comentários....121
5.2 Variáveis....121
5.3 Operadores....123
5.4 Expressões Condicionais....123
5.4.1 If / Else / IfElse (encadeado)....124
5.4.2 Switch....125
5.5 Estruturas de Repetição....125
5.5.1 For / While / Do While....126
5.6 With....127
5.7 Functions....127
5.8 Criando Objetos....128
5.9 Hierarquia do Objeto....129

6 COMANDOS DA ACLIVEJS....130

6.1 Janela 3D....130
6.2 Janela 2D....132
6.3 Sistema de coordenadas 3D....133
6.4 CamOrbita....134
6.5 Grade xy-xz-yz....135
6.6 Particula....136
6.7 Corpo Extenso....138
6.8 Texto 2D....139
6.9 Vetores....140
6.10 Componentes....142

6.11 Referencial Local.....143
7 FICHEIROS HTML.....144
8 CONSIDERAÇÕES FINAIS AO PROFESSOR.....154

APÊNDICE 1: LISTAGEM DO PROGRAMA EXIBINDO TEXTOS E IMAGENS. 156
APÊNDICE 2: LISTAGEM DA SIMULAÇÃO DO MHS.....157

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.

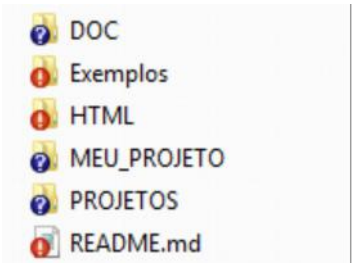


Figura 2.1: Pastas e arquivos da AcliveJS

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.

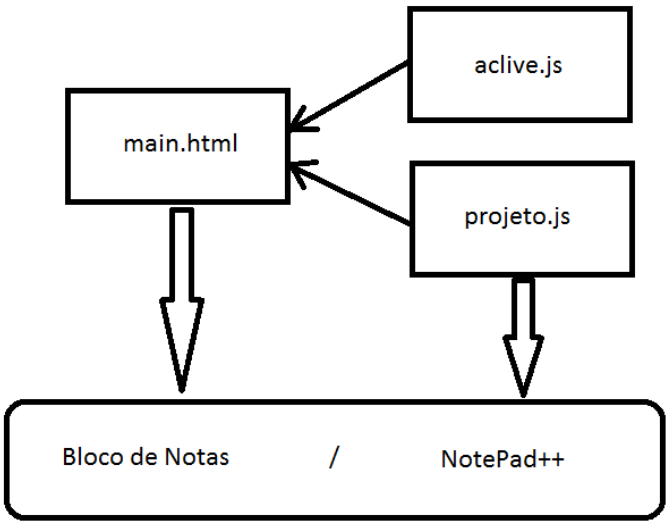


Figura 2.2: Esquema do princípio de funcionamento da AcliveJS

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.*

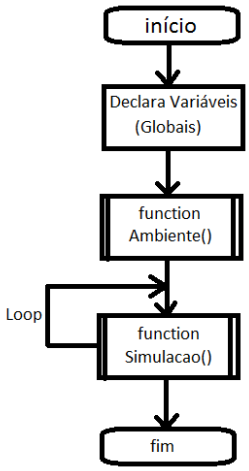


Figura 2.3: estrutura dos programas AcliveJS

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:

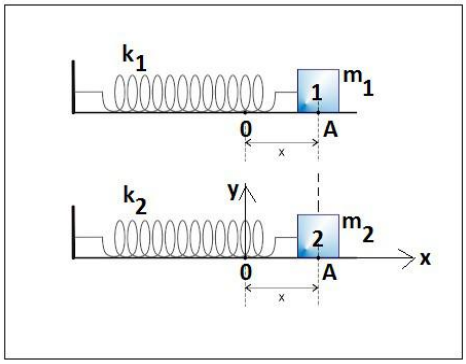


Figura 2.5: Layout da Simulação MHS

2.3.2 A Física do MHS

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

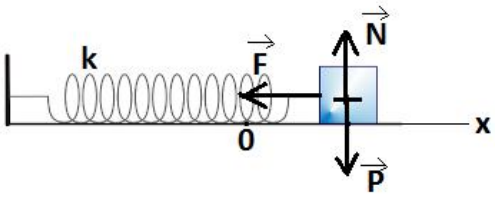


Figura 2.6: Sistema massa-mola sem atrito

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.

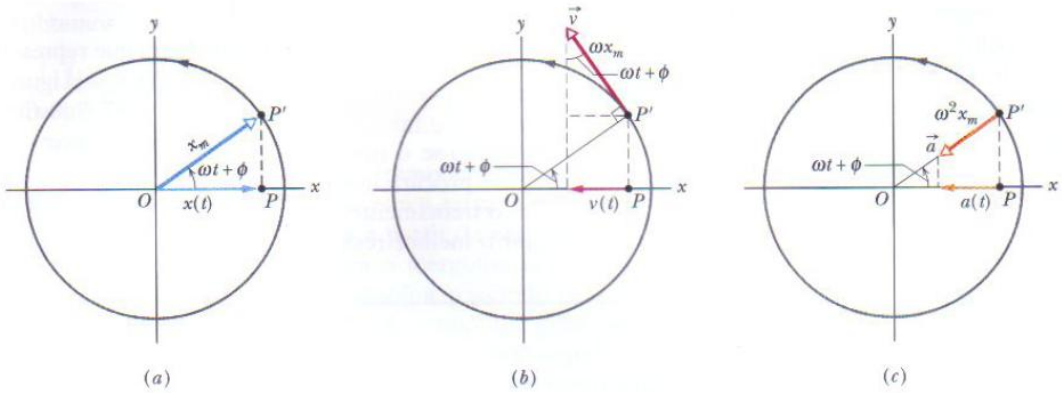


Figura 2.7: Estudo do MHS massa-mola como projeção do MCU.

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.

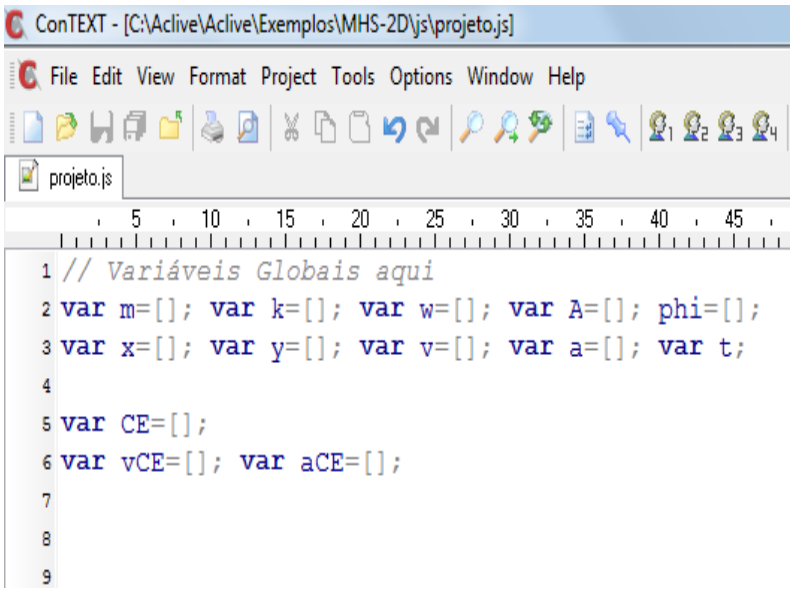


Figura 2.9: Declarando variáveis.

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
```

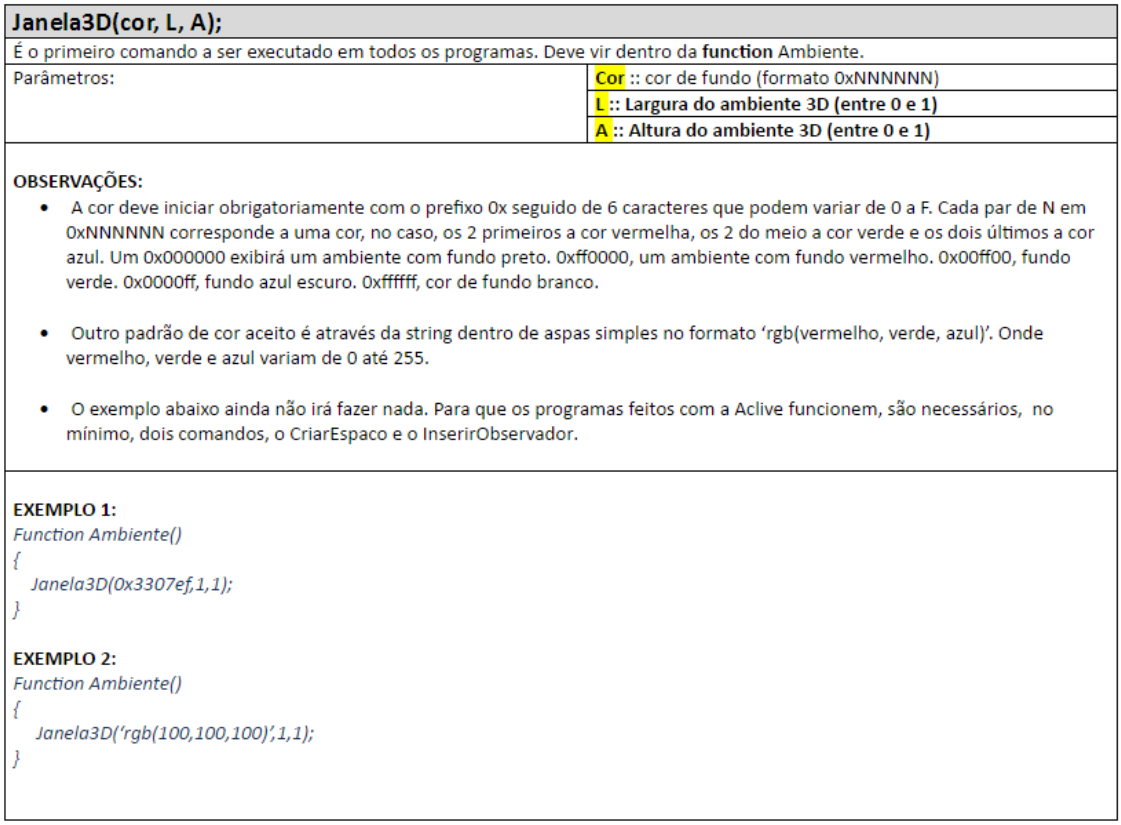
Figura 2.11: Codificação dentro da *function Ambiente()*.

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.

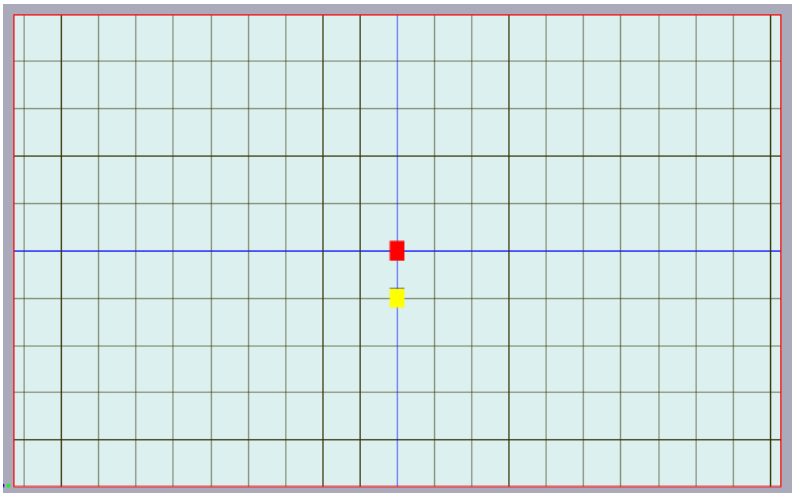


Figura 2.13: Primeira execução da simulação do MHS.

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.

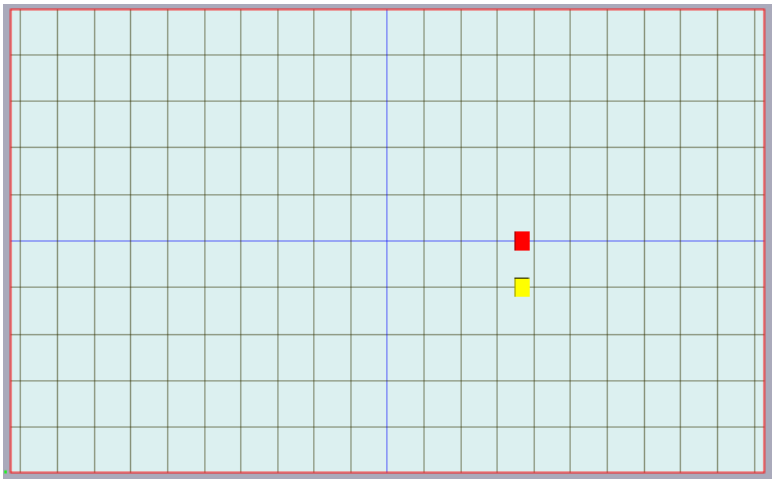


Figura 2.15: MHS. Frequências angulares iguais.

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.

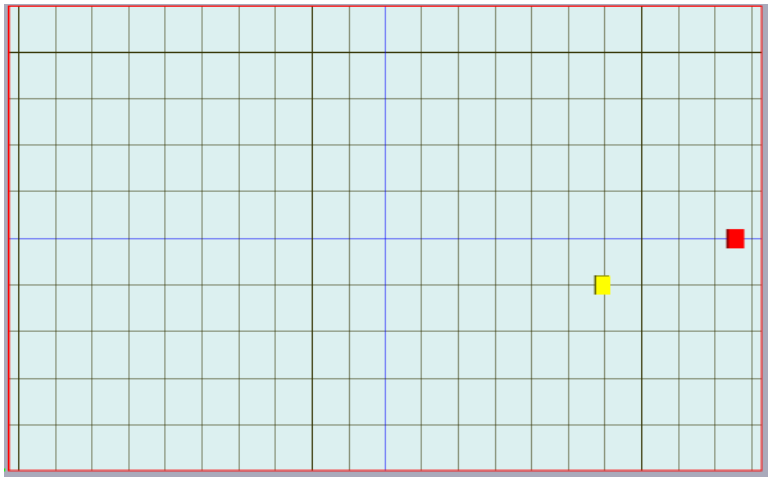


Figura 2.16: MHS. Frequências angulares distintas.

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.

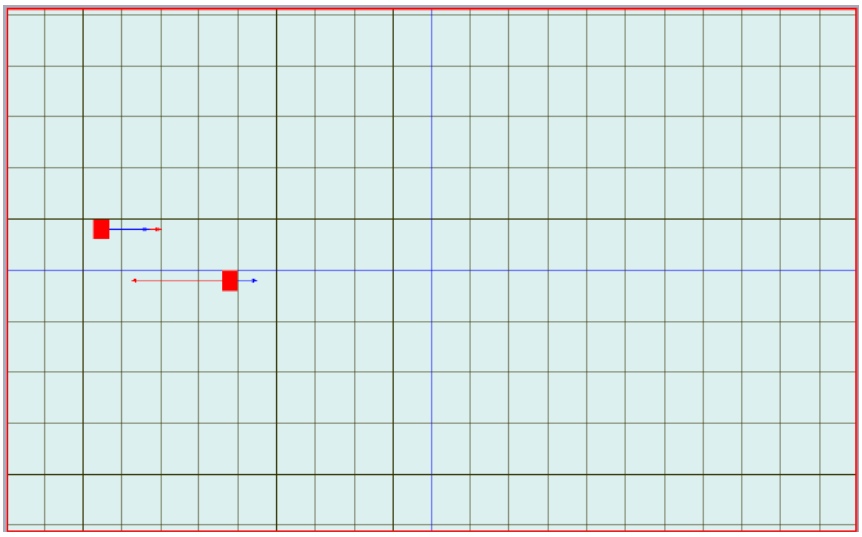


Figura 2.18: Vetor velocidade e aceleração dos blocos 1 e 2.

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.

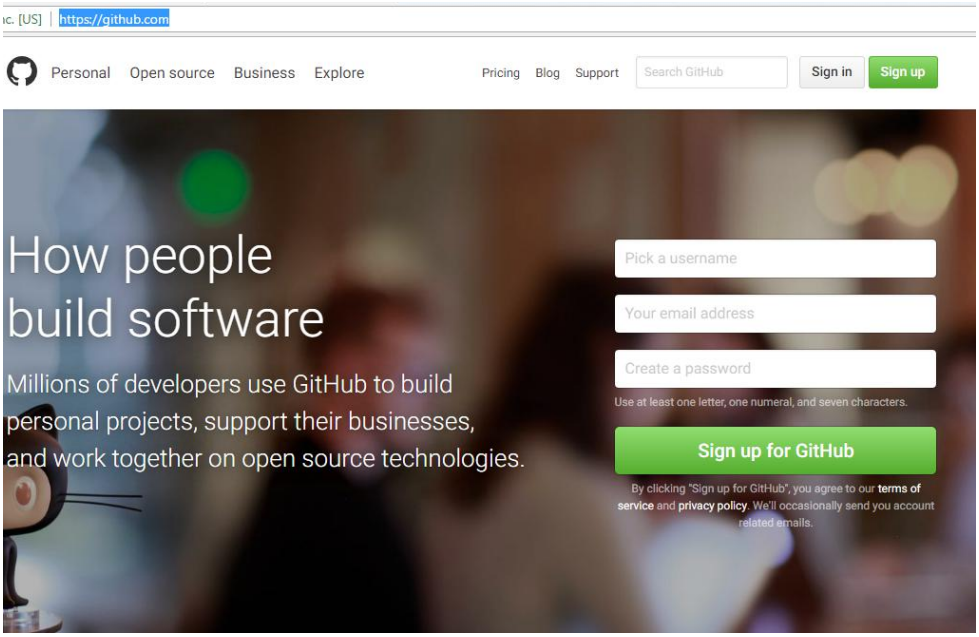


Figura 3.1: Página do GitHub.

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

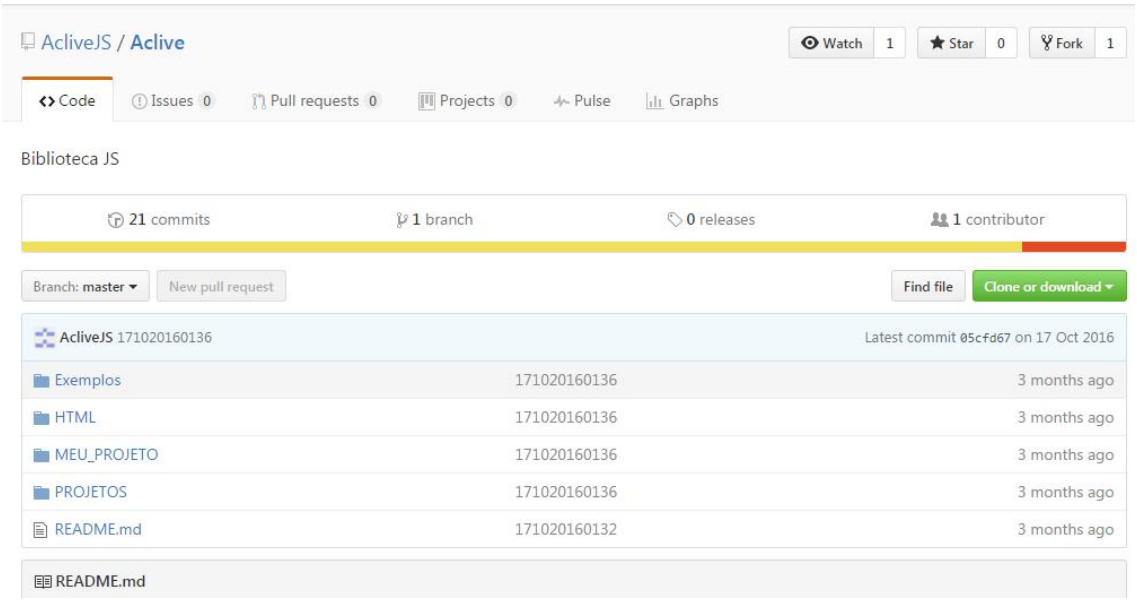


Figura 3.2: Página da AcliveJS no GitHub.

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.

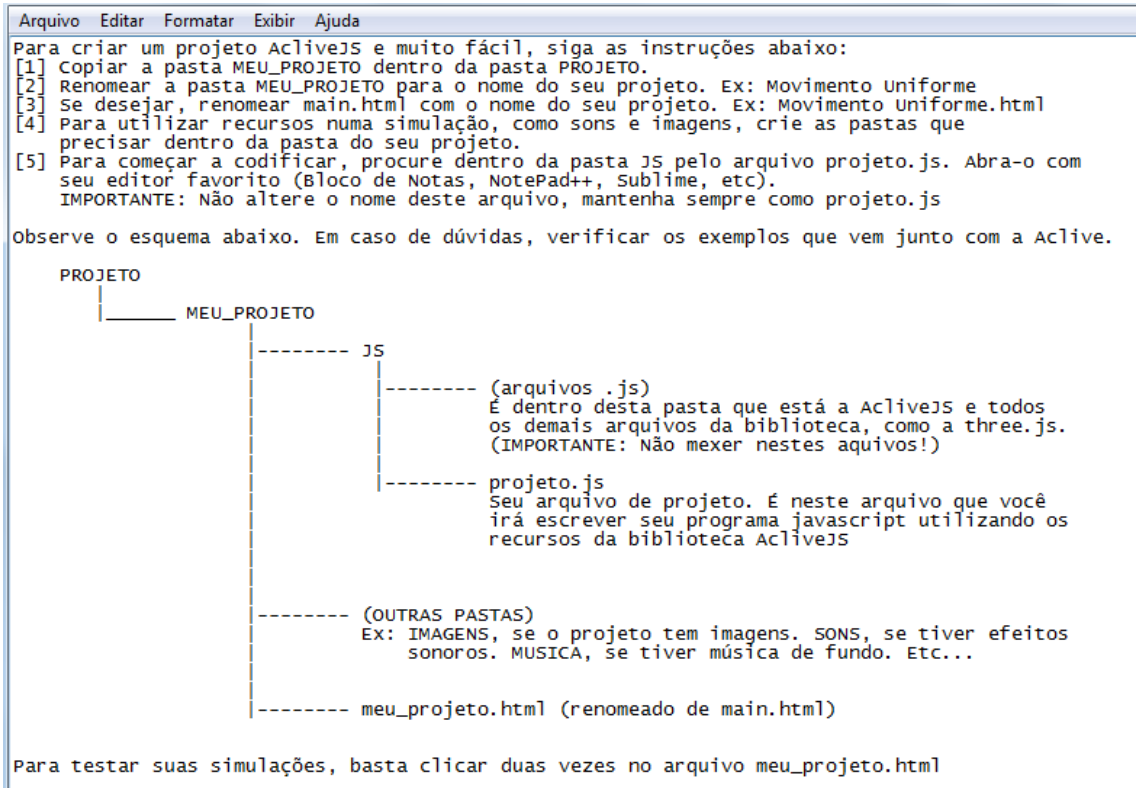


Figura 3.3: Conteúdo do arquivo LEIA-ME PRIMEIRO.txt.

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++.

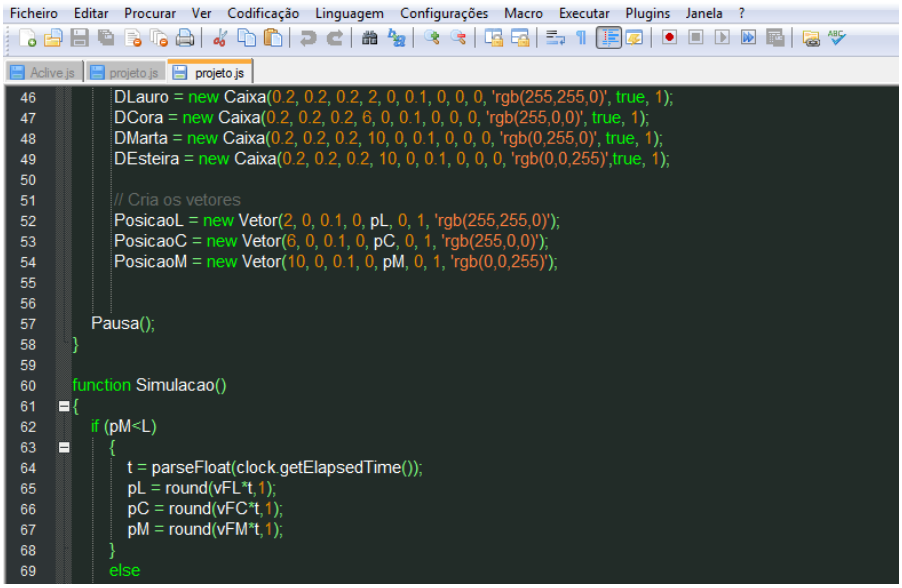


Figura 4.1: Visão do IDE NotePad++

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.

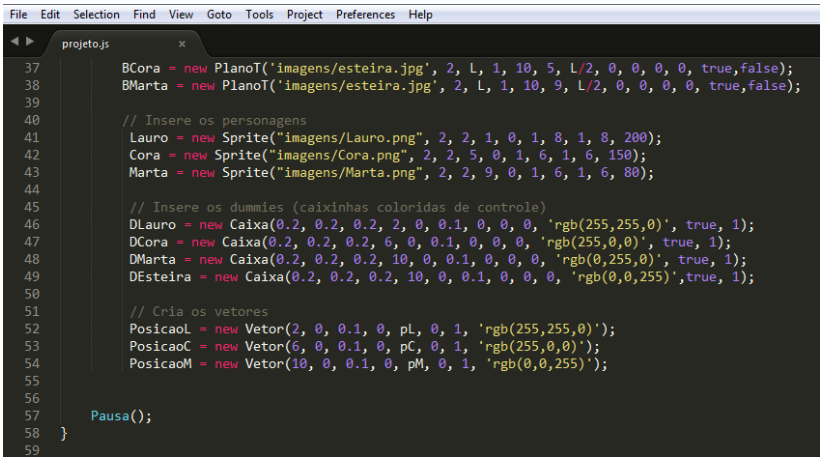


Figura 4.2: Visão do IDE Sublime.

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.

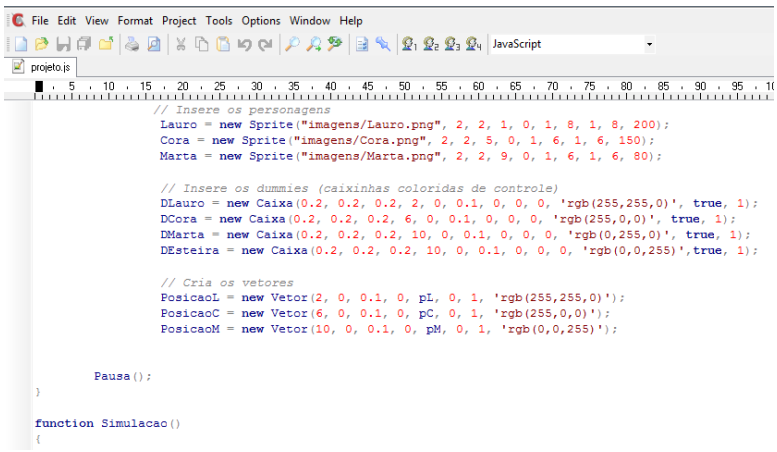


Figura 3.3: Visão do IDE ConText.

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:

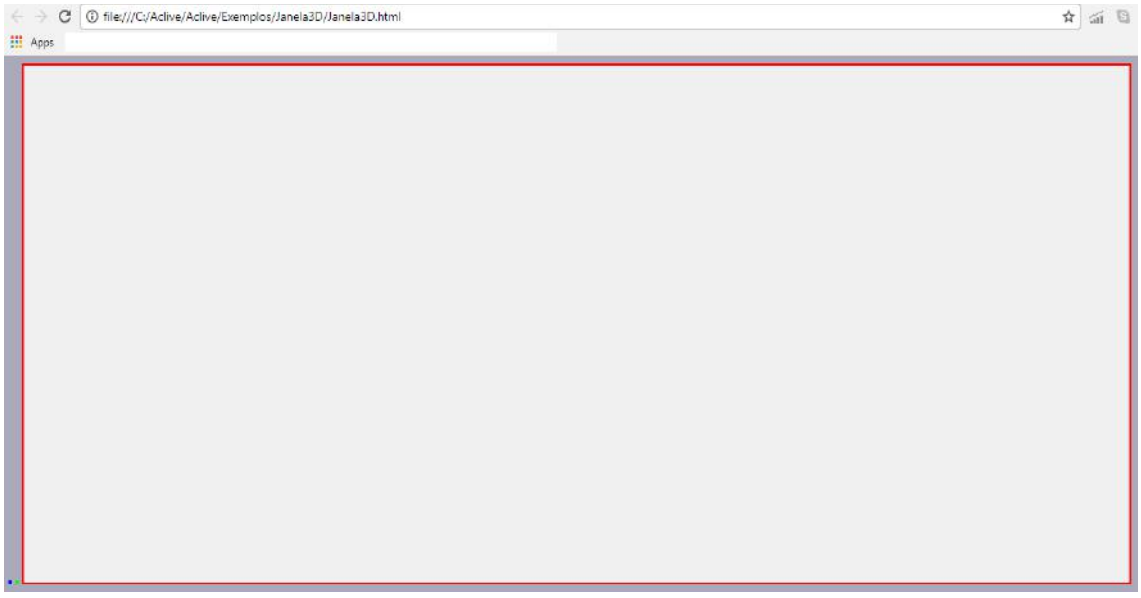


Figura 6.2: Exemplo *Janela3D*.

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:

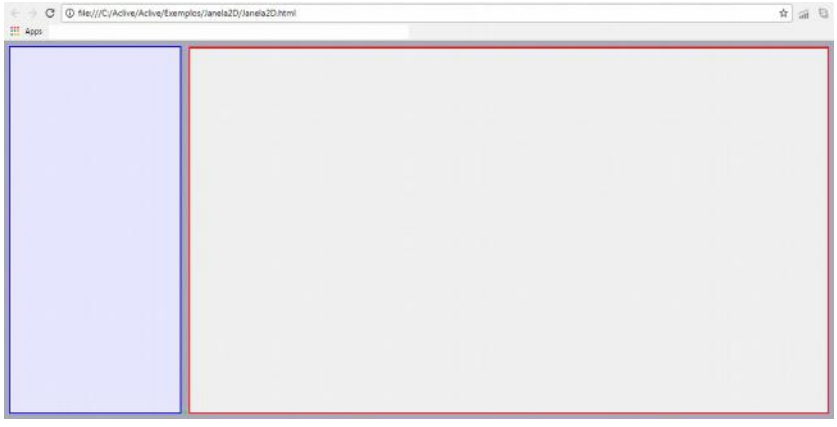


Figura 6.4: Exemplo *Janela2D* em execução no Google Chrome.

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:

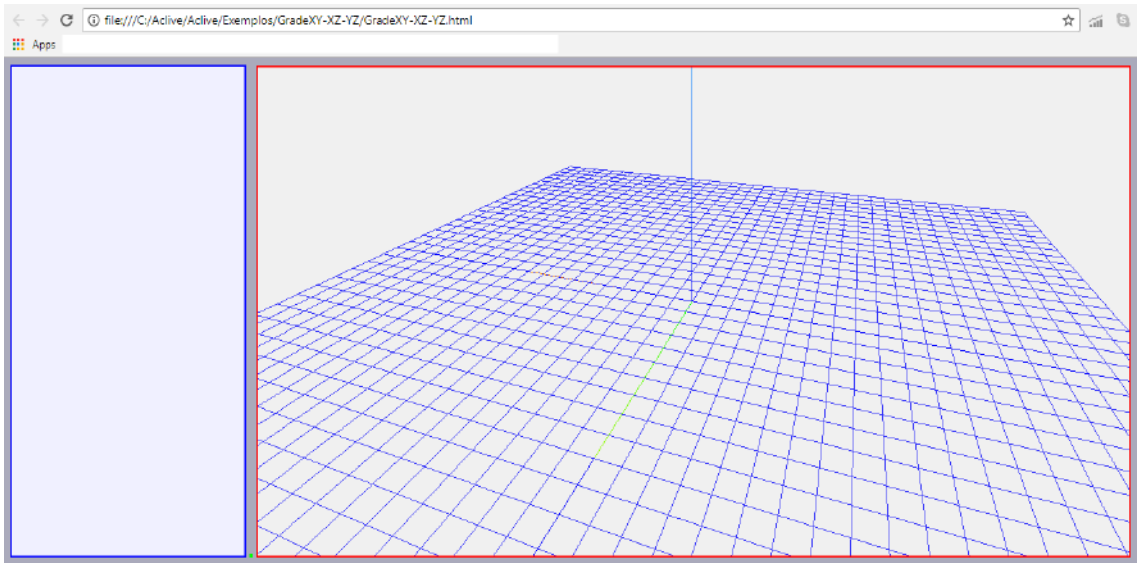


Figura 6.9: *GradeXY* em execução no Google Chrome.

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:

```
2  var p1, p2, p3, p4;
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(60.0, 60.0, 60.0);
10         OlharPara(0.0, 0.0, 0.0);
11         SistemaRetangular(50,false, false, false);
12         GradeXY('rgb(255,255,0)', 'rgb(50,50,0)',100, 5, false)
13         CamOrbita();
14
15         // Cria as partÃ-culas e coloca no ambiente 3D nas posiÃ§Â
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         Janela2D('rgb(40,40,100)',0.2,0.95);
22
23 }
```

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:

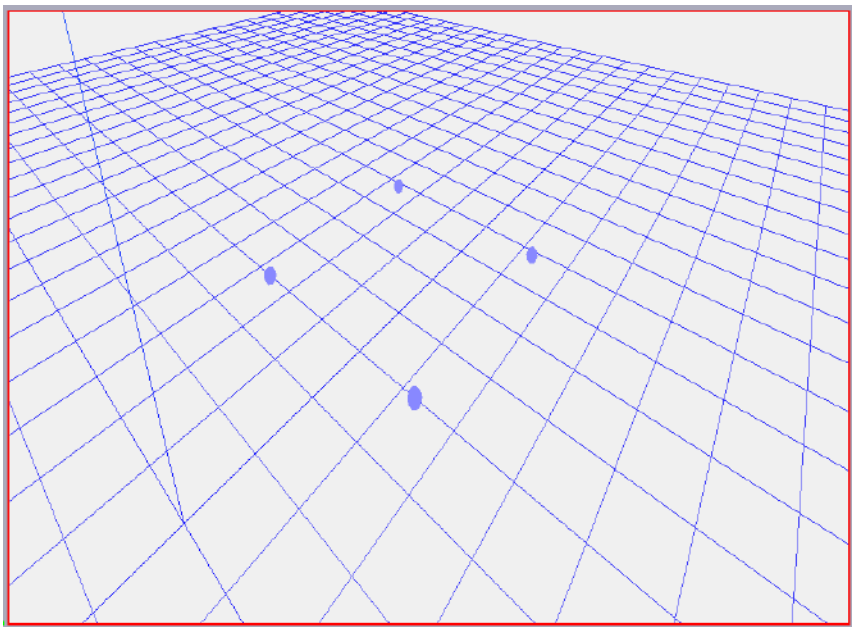


Figura 6.11: *Particula* em execução no Google Chrome.

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:

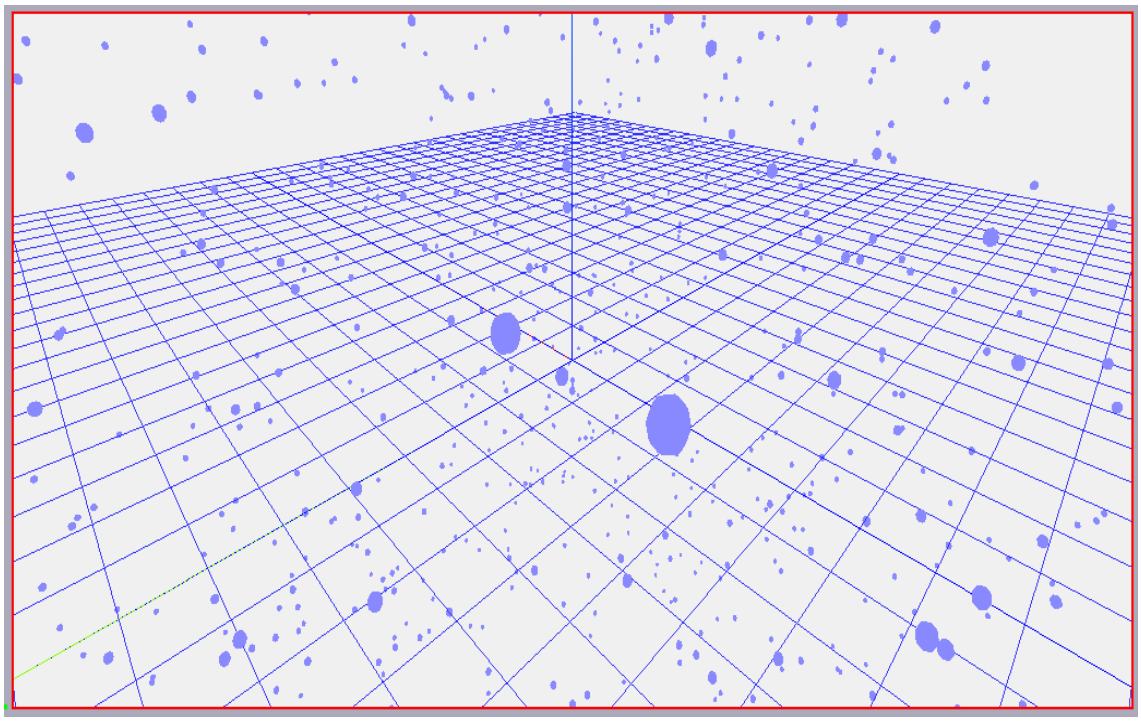


Figura 6.13: *Particula* em execução. Existem 1000 partículas nesta imagem.

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.

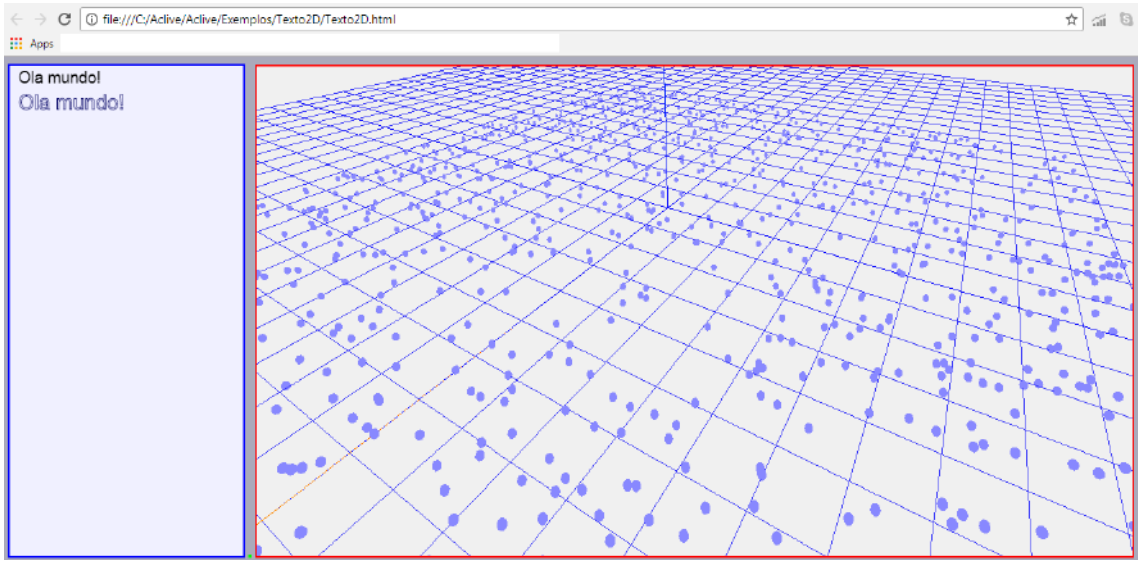


Figura 6.16: Exemplo *Texto2D* "Olá mundo!" em execução.

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.

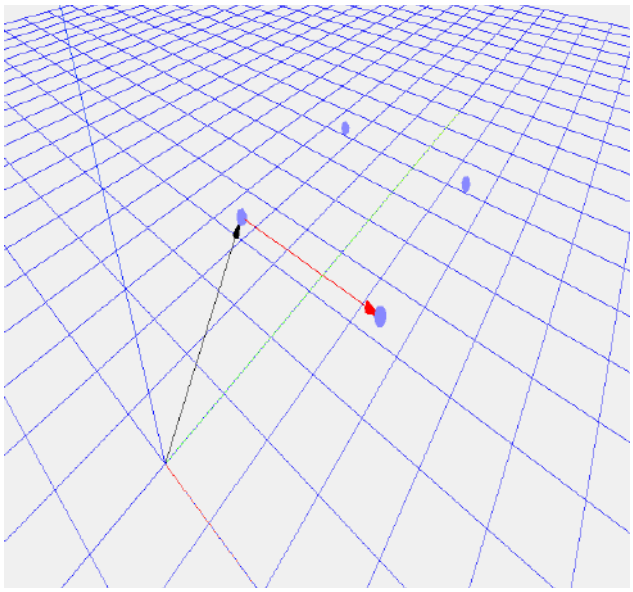


Figura 6.18: Exemplo Vetores em execução no Google Chrome.

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.

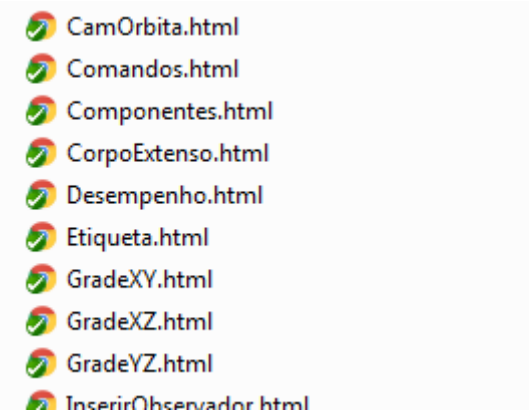


Figura 7.1: Arquivos de ajuda no formato html.

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);

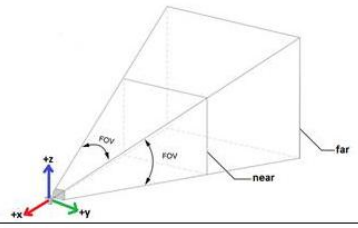
GradeYZ(CorMeio, CorGrade, tamanho, divisão, bool);	
Este comando exibe uma grade quadrada no plano YZ. Útil para visualização das projeções neste plano.	
Parâmetros:	CorMeio :: Cor das linhas centrais da grade
	CorGrade :: Cor da grade.
	Tamanho :: tamanho da grade
	Divisão :: número de subdivisões 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()

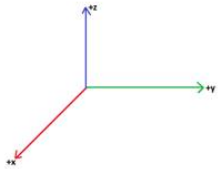
EXEMPLO:

```
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.

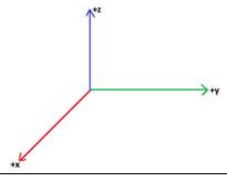
SistemaRetangular(n,boolx, booly,boolz);	
Este comando exibe os eixos do sistema cartesiano global que faz o papel de um sistema inercial, ou seja, em repouso absoluto no espaço relativo aos demais sistemas locais.	
Parâmetros: 	N :: tamanho dos eixos coordenados
	Boolx :: Se **true** vai exibir em pontilhado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