Manual de utilização MAWI

Nascido a partir de um PIBIC, MAWI é um projeto da criação de um motor 2D multiplataforma para o ambiente web, usando recursos do estado da arte introduzidos pelo HTML5, não só para criar aplicações visualmente interessantes, mas também para tirar vantagem dos novos recursos que o HTML5 tem a oferecer criando aplicações interativas. A criação de jogos digitais e expansão das bibliotecas necessárias ao seu funcionamento é uma atividade multidisciplinar que envolve diversas áreas da computação e por tratar-se de uma atividade lúdica, visa também motivar o interesse dos alunos em diversas áreas da computação, além de qualificá-los a respeito das competências necessárias para criação de aplicações web utilizando HTML5.

MAWI é inteiramente feito em JavaScript. Por ter um objetivo didático maior do que comercial, todos os códigos fontes disponíveis não estão compactados e nem muito otimizados, para que seja possível entender os conceitos mais facilmente. A inclusão dos arquivos pode ser realizado no cabeçalho de qualquer página web:

- ```
  <!-- Arquivo que contém a estrutura que representa o laço principal e objeto drawable -->
  <script type="text/javascript" src="mawi/engine.js"></script>
  ```
- ```
  <!-- Arquivo que contém funções utilitárias usadas no motor -->
  <script type="text/javascript" src="mawi/core.js"></script>
  ```
- ```
  <!-- Arquivo que contém algumas funções para calcular colisão ou calculo com vetores -->
  <script type="text/javascript" src="mawi/physics.js"></script>
  ```
- ```
  <!-- Arquivo que contém algumas classes para auxiliar a comunicação via webservice -->
  <script type="text/javascript" src="mawi/net.js"></script>
  ```
- ```
  <!-- Arquivo que contém o grafo de cena e os nós onde são pendurados os atores -->
  <script type="text/javascript" src="mawi/scenegraph.js"></script>
  ```

Ainda temos alguns recursos não obrigatórios que podem ser incluídos:

- ```
  <!-- Framework para auxiliar na criação de efeitos de chuva. Exemplo no arquivo chuva.html -->
  <script type="text/javascript" src="chuva.js"></script>
  ```
- ```
  <!-- Framework para auxiliar na criação de efeitos de explosão. Exemplo no arquivo explosion.html-->
  <script type="text/javascript" src="explosion.js"></script>
  ```
- ```
  <!-- Framework para auxiliar na criação de efeitos de “brilho”. Exemplo no arquivo shine.html -->
  <script type="text/javascript" src="shine.js"></script>
  ```
- ```
  <!-- Framework para auxiliar na criação de smiles animados. Exemplo no arquivo smile.html -->
  <script type="text/javascript" src="smile.js"></script>
  ```
- ```
  <!-- Framework para auxiliar na criação de efeitos de fumaça, fogo ou nuvem. Exemplo no arquivo smoke.html -->
  <script type="text/javascript" src="smoke.js"></script>
  <script type="text/javascript" src="smokeCloud.js"></script>
  ```
- ```
  <!-- Framework para auxiliar na criação de atores no formato de peixe. Exemplo no arquivo peixe.html -->
  <script type="text/javascript" src="peixe.js"></script>
  ```
-
  ```

Estas páginas mostram alguns efeitos predefinidos disponíveis no MAWI. Para auxiliar na programação, o motor gera o código fonte correspondente ao efeito visualizado, bastando então colocar o código gerado na cena desejada (e ajustar de acordo com a hierarquia)

Engine

Engine é a principal classe classe do motor, representa o motor e seu *game loop*. É essencial para funcionamento do mesmo. Gerencia as chamadas de renderização, processamento de entrada do usuário, processamento de detecção de colisão e movimento, som e comunicação.

- **`init(canvas, largura, altura)`**: Define o canvas onde o motor atuará e a altura e largura que este canvas deve ter.
- **`run()`**: Começa a rodar o motor. A partir da chamada deste método o motor inicia o game loop.
- **`stop()`**: Para momentaneamente o *game loop*. Para continuar a execução, basta invocar novamente o método run();
- **`render()`**: Método que trata as chamadas de renderização. Não é necessário invocar este método diretamente. Limpa a tela e invoca o objeto *SceneGraph* para renderizar os objetos da cena.
- **`physics()`**: função que define como será o tratamento da física entre os objetos. Deve ser setado se deseja ser usado, o tratamento *default* não realiza ação alguma.
- **`playSound(som, volume)`**: Toca um arquivo de som.
- **`mainLoop()`**: *Game loop*, aqui são realizadas as chamadas de renderização, física e controle dos FPS. Chamado automaticamente pelo motor, não necessita ser invocado diretamente.

Exemplo de utilização:

```
Engine.init('canvasId', 640, 480);
//CRIAR CENA AQUI
Engine.run();
```

Drawable

Drawable: Todo objeto dentro do grafo de cena que será renderizado deve possuir algumas propriedades específicas: uma matriz de coordenadas homogêneas(m), uma função de renderização (render) e propriedades de largura e altura (width e height)

- **`m[6]:`** Matriz de transformação de coordenadas homogêneas 2D do objeto. Armazena a translação, rotação e escala do objeto obtidas através de transformações afins. As 6 posições representam, na ordem, as 3 linhas da matriz (exceto a coluna 0,0,1) de transformação 2D.
- **`render(ctx)`**: função responsável por renderizar o objeto. A renderização default desenha um retângulo de acordo com a largura e altura do objeto.
- **`height:`** altura do objeto (usado para auxiliar na renderização e eventualmente até na detecção de colisão)
- **`width`**: largura do objeto (usado para auxiliar na renderização e eventualmente até na detecção de colisão)

Exemplo de definição de um objeto Drawable:

```
function Esfera(){
  render: function(ctx){
            ctx.beginPath();
            /*x, y, raio, ângulo inicial, ângulo final, reverse*/
```

```
            ctx.arc(100, 100, 15, 0, 2 * Math.PI, false);
            ctx.closePath();
            ctx.fill();
        }
}
Esfera.prototype = new Drawable();


/*Instanciando o objeto*/
var bola = new Esfera();
bola.m= [1, 0, 0, 1, 100, 500];
bola.height= 15;
bola.width= 15;
```

Scenegraph

Cada elemento renderizável é armazenado em um grafo de cena com uma referência global: SceneGraph. Neste grafo de cena, é possível adicionar nós hierarquicamente. Cada nó,
possui um objeto (atributo obj) e uma lista de filhos. O atributo obj por sua vez deve ser um objeto Drawable que possui sua própria matriz de transformação 2D que representa sua transformação em relação ao seu pai. A estrutura de dados que define um nó é a classe Node:

```
function Node(obj,parent){
   this.obj = obj;
   this.parent = parent;
   this.children = new Array();
   this.add = function(obj){
      var node = new Node(obj,this);
      this.children.push(node);
      return node;
   }
}
```

- obj: objeto renderizável. Deve ser um objeto Drawable que possua sua matriz de transformação 2D e um método render()
- add(filho): adiciona um nó filho ao nó atual e retorna a referência para este nó filho.

Os nós devem então ser adicionados ao SceneGraph. Este possui um método render() que inicia da raiz, invoca o rendeR() do obj da raiz e parte para seus filhos, repetindo o processo até cobrir todos os elementos do grafo de cena. O objeto SceneGraph é invocado automaticamente pelo Engine em seu gameloop. Portanto, tudo que deve ser renderizado está pendurado neste SceneGraph.

Exemplo de criação de objeto e adição ao grafo:

```
var piso1 = new Bloco();
piso1.m= [1,0,0,1,0,  175];
piso1.height= 30;
piso1.width= 165;

var piso2 = new Bloco();
piso2.m = [1,0,0,1, 350, 100];
piso2.height= 30;
piso2.width = 1200;
```

```
var noPrincipal =  SceneGraph.root.add(piso1);
noPrincipal.add( piso2 );
```

Neste exemplo, adicionamos o objeto piso1 na raiz do grafo de cena, que retorna o nó que foi armazenado na variável noPrincipal. Em seguida adicionamos o objeto piso2 como filho de noPrincipal. Assim, se a matriz de transformação 2D do objeto piso1 for alterada, terá influência na posição de piso2. Por exemplo, se transladarmos piso1 de 100 pixels, seus filhos serão transladados também.

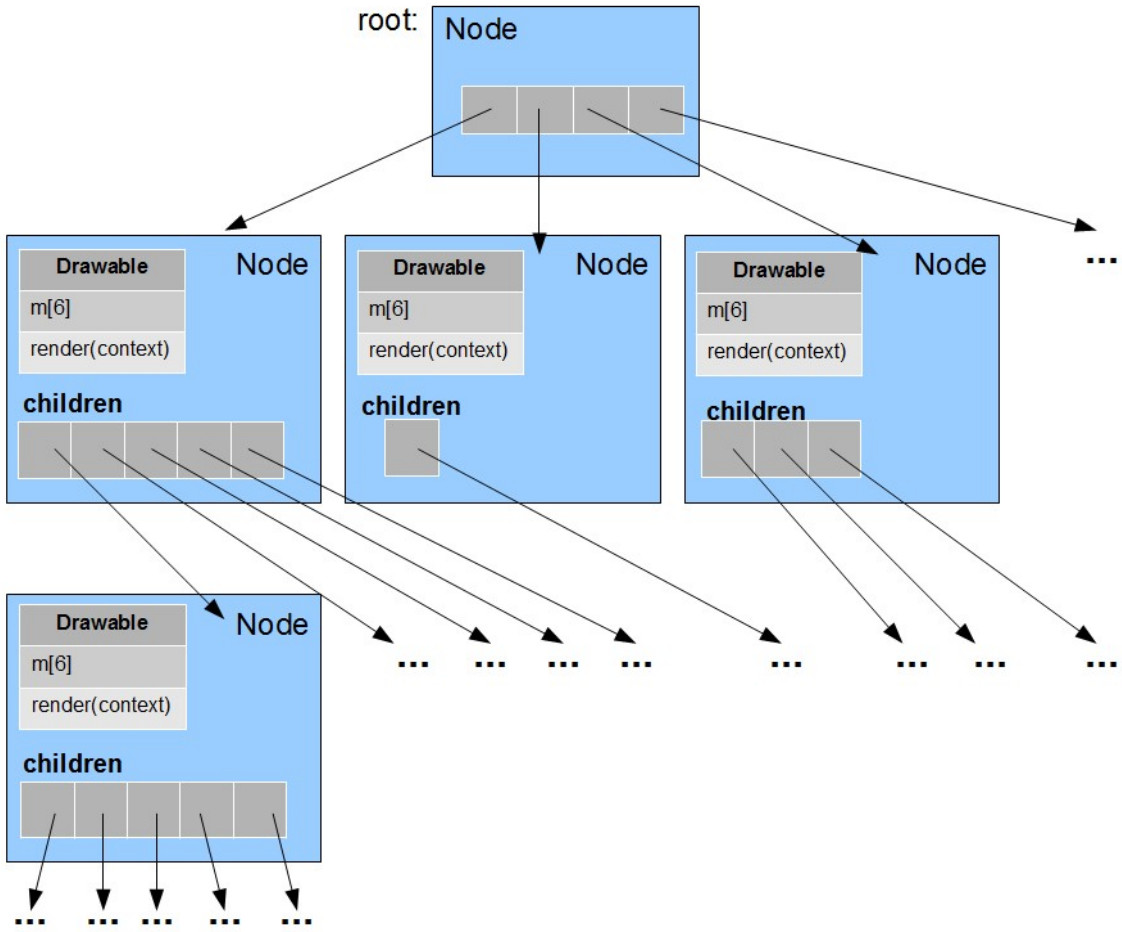

Figura 1: Representação gráfica da estrutura SceneGraph

Exemplo completo:

```
<html>
    <head>
   <title>Demo1</title>
   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
   <script type="text/javascript" src="../mawi/engine.js"></script>
   <script type="text/javascript" src="../mawi/core.js"></script>
    <script type="text/javascript" src="../mawi/physics.js"></script>
   <script type="text/javascript" src="../mawi/scenegraph.js"></script>

   <script type="text/javascript">
        window.onload = function(){
```

```
            Engine.init("canvasId", innerWidth*0.50, innerHeight*0.50);
            var piso1 = new Bloco();
            piso1.m= [1,0,0,1,30,  175];
            piso1.height= 30;
            piso1.width= 30;

            var piso2 = new Bloco();
            piso2.m = [1,0,0,1, 250, 100];
            piso2.height= 30;
            piso2.width = 120;

            var noPrincipal =  SceneGraph.root.add(piso1);
            noPrincipal.add( piso2 );
            Engine.run();
        }
    </script>

   <style type="text/css">
        #canvasId{
            background-color: black;
        }
   </style>
</head>
   <body>
      <canvas id="canvasId"></canvas>
   </body>
</html>
```

Ilustração 2: Imagem Gerada pelo código Demo1

Adicionando Eventos

É possível adicionar captura de eventos ao canvas ou à janela normalmente, da forma mais tradicional. Porém a função Engine::processInput() pode ser usada para processar os dados de entrada do usuário. A função processInput() é invocada dentro do gameloop, ficando sincronizada com a renderização ou processamento de colisão.
Exemplo:

```
 window.onkeydown = function(event){
```

```
      controlDownFunction(event);
};
   window.onkeyup = function(event){
      controlUpFunction(event);
};
```

ou

```
Engine.processInput = handleKeys;
```

Animações

Animações utilizando efeitos já implementados pelo framework ou desenhos que usam formas primitivas, arcos, retas ou curvas de bezier, podem ser realizadas dentro do método render() de cada objeto renderizável (que possuir as propriedades de um Drawable).
Geralmente o objeto renderizável possui um ou mais atributos que controlam a animação, sua duração dentre outros parâmetros.

Exemplo:

```
var sol = {
   step: 0.1,
   frame: 0,
   limit: 0.6,
   m :  [1,0,0,1,150,0],
   height: 60,
   width: 60,
    render: function(ctx){
         ctx.save();
        ctx.shadowColor = "yellow";
         ctx.shadowOffsetX = 0;
        ctx.shadowOffsetY = 5;
         ctx.shadowBlur = 20;
        ctx.beginPath();
        ctx.arc(sol.m[4], 0 , sol.width + sol.frame, 0, 2 * Math.PI, false);
         if(sol.frame == sol.limit || sol.frame == sol.limit*-1){
              sol.step = sol.step*-1;
         }
         sol.frame += sol.step;
         ctx.fillStyle = "yellow";
         ctx.fill();
         ctx.lineWidth = 3;
         ctx.strokeStyle = "orange";
         ctx.stroke();
         ctx.restore();
    }
};
sol.prototype = new Drawable();
```

Para Adicionar o sol a raiz da cena, basta executar: SceneGraph.root.add(sol);

Ilustração 3: Exemplo de animação usando primitivas no método render()

Animação usando Sprites

Um sprite é uma única imagem que será incorporada a uma cena maior de forma que seja parte daquela cena. É uma forma popular de criar cenas mais complexas, pois é possível manipular cada sprite separadamente. Isto permite um controle maior sobre como a cena é renderizada e até como os jogadores interagem com a cena. Alguns jogos podem ter centenas de sprites. Diante disso, carregar cada sprite pode ser custoso, por isso muitos jogos utilizam os chamados spritesheets.
Quando vários sprites são colocados em uma única imagem, chamamos esta imagem de spritesheet. Spritesheets são usadas para melhorar o processo de busca e exibição das imagens na tela. É mais fácil requisitar uma única imagem e mostrar apenas parte dela do que buscar várias imagens e exibi-las. Isso é ainda mais perceptível em uma página web, onde cada imagem é buscada por uma requisição HTTP.
Em suma, animação usando Spritesheet nada mais é do que carregar um spritesheet e mudar qual sprite é renderizado em rápidas sucessões para dar a ilusão de movimento, assim como um desenho animado.

No MAWI as animações baseadas em *spritesheets* são realizadas pela classe CAnimation. A ideia básica é ir alternando entre essas poses, com intervalos definidos pelo programador , dando a impressão de movimento. É possível que cada objeto do motor possua uma variedade de animações e dependendo de seu estado, ative a animação adequada (ver exemplo jacaré). Também é possível mesclar diferentes tipos de animações em um só ator.

CAnimation

- horizontalSprites: indica se os spritesheets contém sequências de animação na horizontal(default) ou na vertica.
- width: largura do sprite. O motor permite que seja diferente do tamanho da figura.
- height: altura do spritel. O motor permite que seja diferente do tamanho da figura.
- frames: Matriz, com três colunas: A primeira coluna é a Imagem(objeto Image Javascript). A segunda e terceira colunas possuem um vetor de mesmo tamanho. O primeiro vetor aramzena os intervalos de tempo que devem ser esperados antes de trocar para o próximo sprite do spritesheet. O segundo vetor é o índice de cada sprite dentro do spritesheet que será utilizado em cada intervalo definido no primeiro vetor.

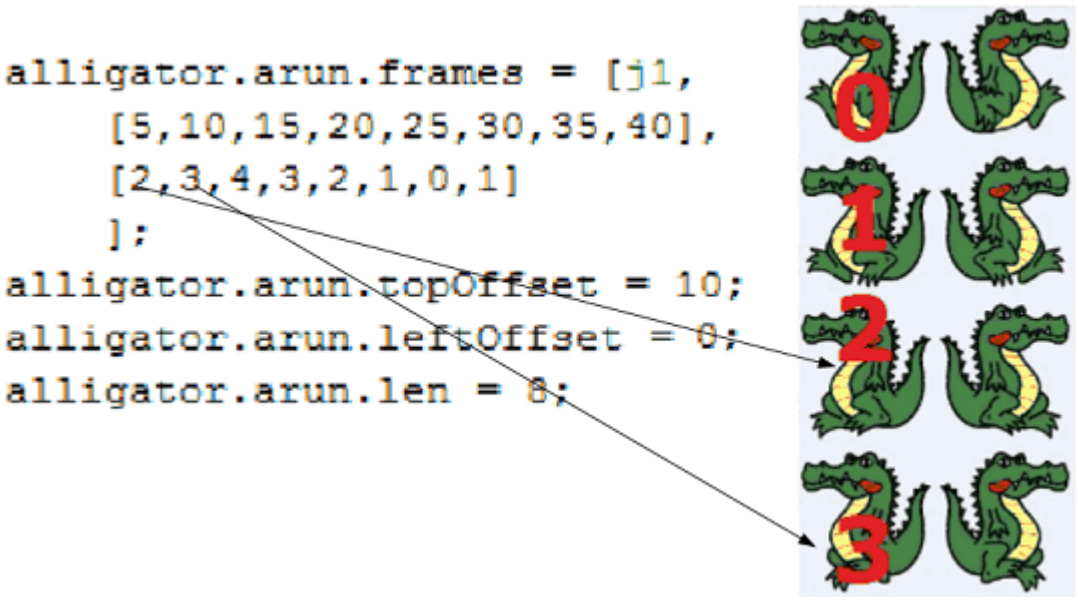

Ilustração 4: Exemplo do funcionamento de um objeto CAnimation

No exemplo da ilustração 3, temos um spritesheet vertical. J1 é um objeto do tipo Image (figura a direita) A animação definida possuirá 8 sprites. Inicialmente o sprite de índice 02 será mostrado; após 5 frames, o sprite será trocado pelo sprite 03; Após mais cinco frames(10), o sprite 03 será trocado pelo sprite 4 e assim sucessivamente.

- Cont: Define se a animação deve ser repetida indefinidamente após percorrer toda a sequência de sprites. Após último sprite a animação recomeça do início.
- run(ctx, obj): Executa uma animação. É importante observar que o objeto animado deve ter uma propriedade chamada *left*. Esta propriedade diz se o ator está virado para esquerda ou direita. O motor considera que os spritesheets contém duplas de sequências de animação: A primeira para o ator voltado para esquerda e a segunda para o ator voltado para direita(espelho). Desta forma basta mudar a propriedade left do objeto que a animação será trocada automaticamente. Na figura anterior, a imagem de jacaré possui duas colunas: a primeira é para a propriedade left=true e a segunda para left=false. Se houver apenas uma posição possível para o ator, left deve ser true.
- update(obj): O parâmetro obj é o objeto sendo animado. Ao final de cada chamada do método run(ctx,obj), é invocado o método update. Neste método são atualizados os ticks, que verificam se o spritesheet deve ser trocado. Além disso é invocado o método callback(this, obj, unfinished) se este foi fornecido pelo usuário. Isto pode servir , por exmeplo, para criar outro objeto ou mudar a animação dependendo do estado do objeto. O parâmetro unfinished indica se a animação ainda não foi executada por completo ao menos uma vez.
- width = largura do sprite
- height = altura do sprite
- name = Um nome qualquer para a animação. Pode ser usado por exemplo para saber se determinada animação já está sendo executada.
- callback = função opcional fornecida pelo usuário que pode receber três parâmetros: o CAnimation sendo executado, objeto sendo animado e um booleano indicando se a animação continua;

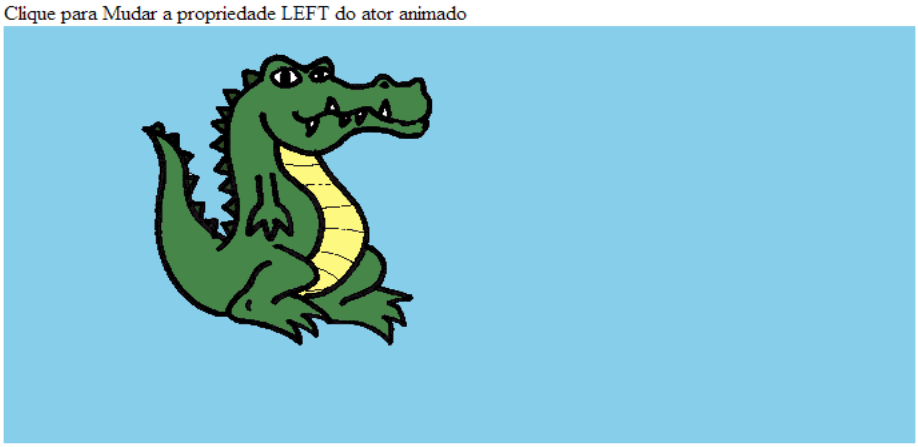

Ilustração 5: Exemplo de animação usando sprites e a propriedade left

Por exemplo, podemos criar um nó do nosso grafo de cena representando o ator e criar como seus filhos um objeto que seja possua uma animação por sprites e um outro que possua animação por renderização de primitivas.

Sons

Sons são tocados através do objeto Audio introduzido no HTML5. Os formatos suportados mais comuns são .mp3 e .ogg (há ainda alguma variação no suporte entre os navegadores, sendo que o ideal é oferecer recursos nos dois formatos).
Para tocar um som em determinado momento:

```
var som = identificaNavegador() == "firefox" ? new Audio("jump.ogg") : new Audio("jump.mp3");
/* objeto audio e volume (0 a 1)*/
Engine.playSound(som, 1.0)
```

Música de Fundo:

Há a possibilidade de definir uma música de fundo, tocada repetidamente pelo motor:

```
var sound = identificaNavegador() == "firefox" ? new Audio("intro.ogg") : new Audio("intro.mp3");
Engine.music = sound;
```

Após o Engine.run() é possível ajustar outros atributos da música de fudo como volume (entre 0 e 1) e repetição:

```
Engine.music.volume = 1;
Engine.music.loop = true;
```

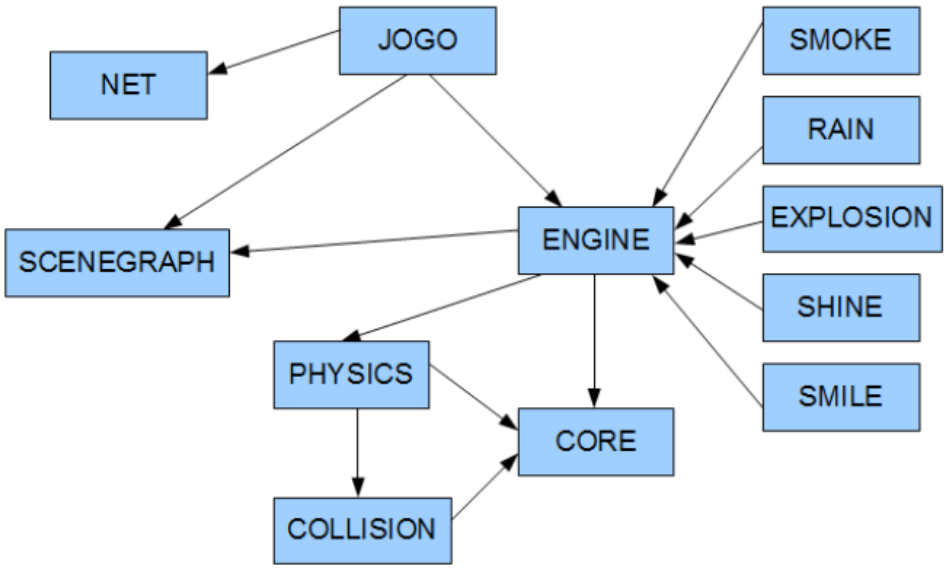

Figura 6: Arquitetura de grafo acíclico das bibliotecas do MAWI