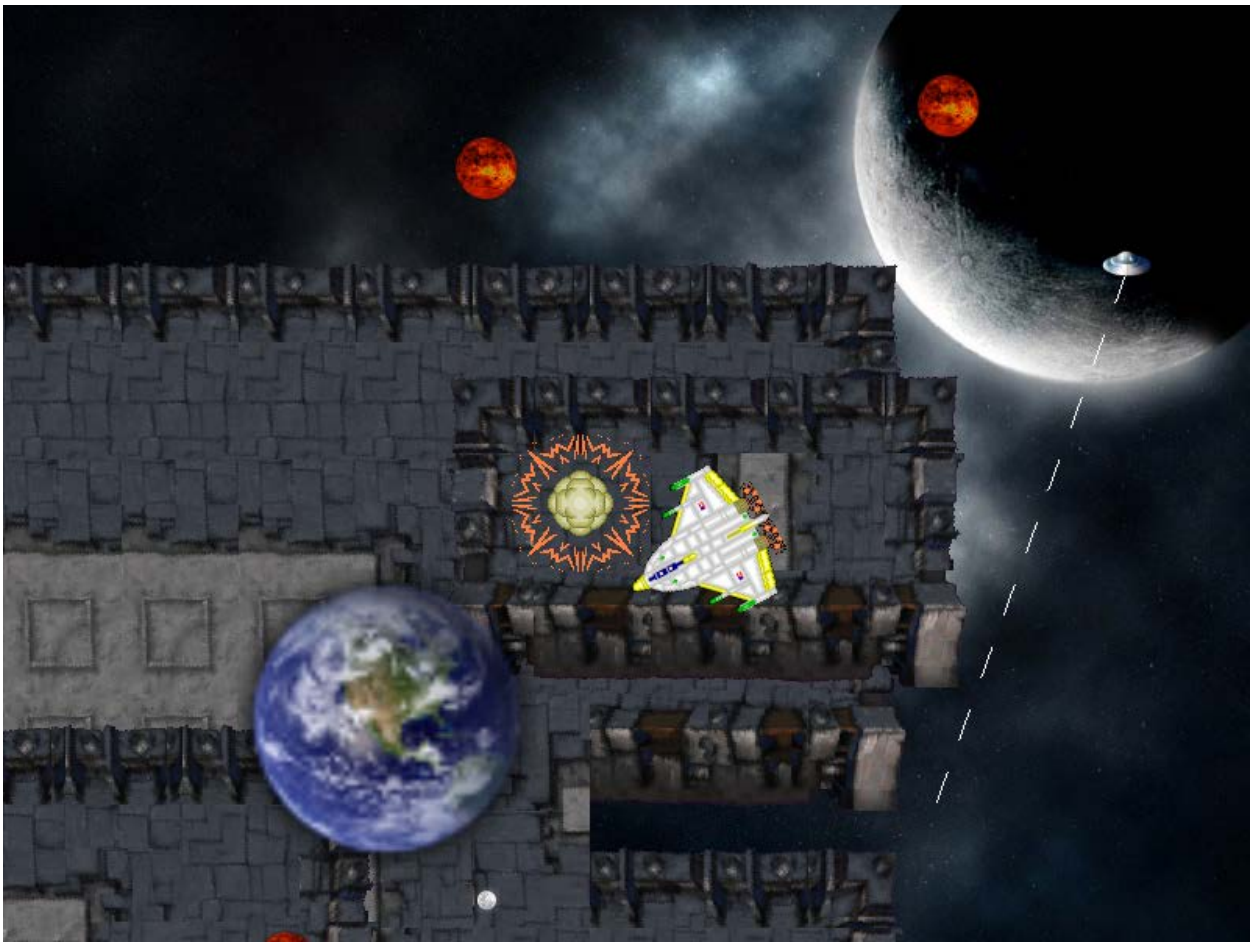
Introdução ao Desenvolvimento de Jogos – Turma A
Professora: Carla Denise Castanho (carlacastanho@cic.unb.br)
Monitores: Davi “Doom” Diniz (khenstot@gmail.com)
Lenonardo Guilherme (leonardo.guilherme@gmail.com)
Luiggi Monteiro Reffatti (lmreffatti@gmail.com)

Trabalho 3 – Animação, Movimento Acelerado, Rotação, Zoom e Colisão



Perdidos no espaço por anos e recentemente sendo caçados por alienígenas, os tripulantes da nave Hyperion vasculham desesperadamente o setor em busca de seu planeta natal, a Terra. Sem chance de combate com os inimigos, sua única esperança é que as armas Terrestres consigam derrotar esta ameaça.

Os alienígenas, por sua vez, procuram por um material único em toda a galáxia, que acredita-se ser capaz de gerar energia infinita, e assim, salvar seu planeta da destruição. Supostamente todo ele foi usado na construção da maior Nave de transporte já feita, a Hyperion.

Hyperion: ASWD para mover e correr pela sua vida!

UFO: Mouse 3 coloca aquela posição na fila de comandos. Se a Hyperion for abordada a vitória é certa. Mouse 1 cria um planeta em um posição aleatória da tela. Dessa forma, podemos abater a Hyperion e roubar seus destroços!

1. Rotação e Zoom.

a. Classe SDLBase: rotoZoom e clip.

SDLBase
[...] + clip(original : SDL_Surface*, clip : SDL_Rect*) : SDL_Surface* + rotoZoom(surface : SDL_Surface*, angle : float, scalex : float = 1.0f, scaley : float = 1.0f) : SDL_Surface*

Agora vamos experimentar algumas funções da SDL_gfx. A primeira será rotação e zoom de uma Surface. Inclua o header:

```
#include <SDL/SDL_rotozoom.h>
```

A função para criar uma nova surface com rotação e/ou zoom é a seguinte:

```
SDL_Surface* rotozoomSurface(SDL_Surface *src, double angle, double zoom, int smooth)
```

Os atributos da função são:

- src – a surface que será rotacionada e/ou terá zoom aplicado.
- angle – o ângulo de rotação em graus.
- zoom – o zoom a ser aplicado. O valor 1 corresponde a imagem em seu tamanho original. Se o valor for menor, ela será diminuída, e se for maior será aumentada.
- smooth – aplica anti-aliasing se o valor for 1, não aplica se for 0.

A função retorna uma surface com rotação e/ou zoom.

Novos métodos da classe:

+ rotoZoom(surface : SDL_Surface*, angle : float, scalex : float = 1.0f, scaley : float = 1.0f) : SDL_Surface*

Conferir se os parâmetros passados são válidos (surface não nula, tamanhos > 0, etc). Chamar o método rotozoomSurface e retornar a surface resultante.

+ clip(original : SDL_Surface*, clip : SDL_Rect*) : SDL_Surface*

Checar se os parâmetros são válidos e então criar uma nova surface, através da função:

```
SDL_Surface *SDL_CreateRGBSurface(Uint32 flags, int width, int height, int depth, Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32 Amask);
```

A nova surface deve ter o tamanho do retangulo de clip, e deve ter o formato da surface da tela. Fazer então o clip da surface original e copiar para a nova surface, usando a função:

```
SDL_BlitSurface(SDL_Surface *src, SDL_Rect *srcrect, SDL_Surface *dst, SDL_Rect *dstrect);
```

b. Classe Sprite: Rotação, Zoom preparada para Animação

Sprite
[...] - surfaceRotoZoom : SDL_Surface* - angle : float - scaleX, scaleY : float
[...] + virtual update(dt : int) : void + rotoZoom(angle : float, scalex : float = 1.0f, scaley : float = 1.0f, force : bool = false) : SDL_Surface* + restore() : void + render(x : int, y : int) : void + clip(x : int, y : int, w : int, h : int) : void + getClip() : SDL_Rect

Novos atributos da classe:

- surfaceRotoZoom : SDL_Surface*
  Guarda a surface modificada por rotação e zoom.
- angle : float
  Guardam o ângulo da rotação da Sprite.
- scaleX, scaleY : float
  Guardam os valores de zoom em cada eixo, em relação à imagem original.

Não esquecer de inicializar esses atributos: a surface com NULL, o angulo com 0, e as escalas com 1.0.

Novos/Alterações nos métodos da classe:

+ virtual update(dt : int) : void
  Não deve fazer nada aqui, será implementado na Animation.

+ restore() : void
  Deleta a surfaceRotoZoom.

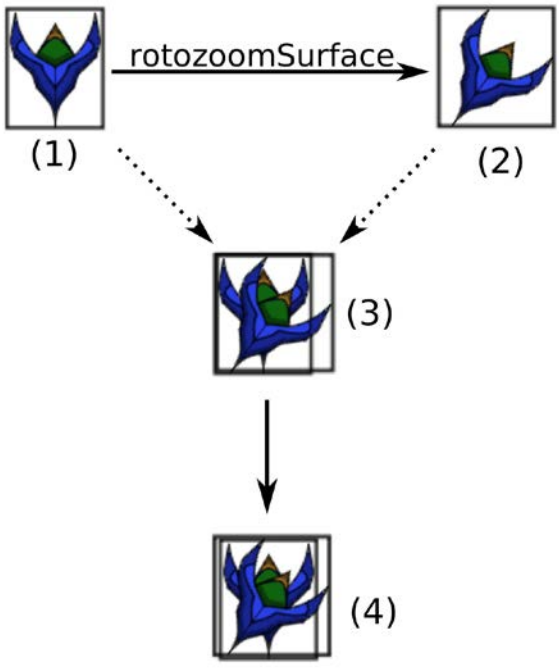
`+ rotoZoom(angle : float, scalex : float = 1.0f, scaley : float = 1.0f, force : bool = false) : SDL_Surface*`

- Checa se é necessário fazer alguma alteração na imagem, ou seja, se angle, scalex ou scaley são diferentes dos atuais. Caso sejam, é necessário rotacionar. Se force == true, a rotação também deve ser feita, independente dos parâmetros passados.
- Libera a surfaceRotoZoom anterior, caso houver.
- Atribui os parâmetros angulo, scalex e scaley.
- Confere se qualquer um dos valores do cliprect foi alterado, ou seja, se a surface usa clip. Caso use, é necessário "clipar" a surface antes de alterá-la. Use o método
  SDLBase::clip para clipar a surface original para a surface rotozoom.
  DICA: O método blit torna indefinidos os valores de w e h do Rect. Para resolver isso, basta fazer uma cópia do rect e passar essa copia para o método clip.
- Agora podemos alterar a surface. Basta usar o método
  SDLBase::rotoZoom com os parâmetros passados.

`+ render( x : int, y : int) : void`

Agora temos 2 surfaces diferentes em cada Sprite, temos que decider qual mostrar na tela. Para isso, basta checar se a surfaceRotoZoom existe. Se existir, mostramos ela, senão mostramos a normal.

Ao mostrar na tela a surfaceRotoZoom será necessário fazer um ajuste de posição, pois a rotação da imagem não é feita "pelo centro". Para isso basta calcular a diferença entre o ponto central do cliprect e o ponto central da surfaceRotoZoom e adicionar esse valor à posição na hora de mostrar na tela.



`+ clip( x : int, y : int, w : int, h : int) : void`

Caso o clip seja alterado e a surfaceRotoZoom não seja nula, devemos recalcular as transformações. Para isso, basta chamar o método rotoZoom com os parâmetros atuais e a variável force == true.

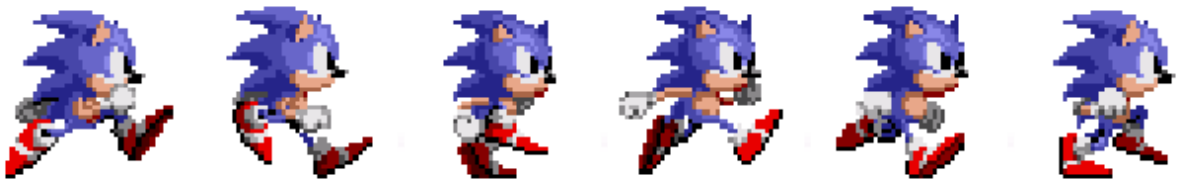
`+ getClip() : SDL_Rect`

Retorna o retângulo de clip da Sprite

2. Classe Animation: Gerando animações.

Animation : Sprite
- n_sprites : int - time : int - frameSkip : int - frameTime : int - clipRect : SDL_Rect
+ Animation(filename : string, frameTime : int, n_sprites : int) + update(dt : int) : void + setFrame(frame : int) : void + setFrameTime(time : int) : void + getframeTime() : int

O objeto Animation é o mesmo que a Sprite, só que animado. Sendo assim, ele herdará da classe Sprite. Basicamente, a Animation irá ler uma sprite sheet, ou seja, uma imagem com diversos quadros de uma animação, em intervalos de espaço iguais, e transformar em uma animação.
Um exemplo é esse que se segue:



Trata-se de um *sprite sheet* de animação do personagem Sonic the Hedgehog, do jogo Sonic the Hedgehog para Mega Drive, se movimentando (mantido aqui como referência em homenagem ao Ígoru ^^). Cada frame de animação é colocado a frente do outro e estes devem possuir o mesmo tamanho.
A idéia é que, em cada quadro de animação, um dos sprites apareça. Para fazer isso, lembre-se do método que foi utilizado no tileMap:

```
SDLBase::blitSurface(SDL_Surface* surface, SDL_Rect* clip, SDL_Rect* dst)
```

O segundo parâmetro (clip) é justamente um retângulo que diz qual a área da surface que será blitada. Ou seja, se ele for um srcrect {x = 0, y = 0, w = 45 e h = 45}, o primeiro sprite à esquerda será blitado na tela. Já com um srcrect { x = 46, y = 0, w = 45 e h = 45}, será o segundo sprite, da esquerda para a direita; e assim por diante. Naturalmente, todo esse processo deverá ser temporizado. Assim, temos a variável frameTime, que diz o quanto de tempo em milissegundos deve durar um frame. Quanto maior o tempo do frame, mais devagar ocorrerá a animação.

Sobre os atributos da classe:

- `n_sprites : int`
  Número de sprites na spritesheet.
- `time, frameSkip: int`
  São usados para calcular a sequência de animação.
- `frameTime : int`
  Define a velocidade da animação.
- `clipRect : SDL_Rect`
  Retângulo de clipping na posição da sprite atual.

Sobre os métodos da classe:

`+ Animation(filename : string, frameTime : int, n_sprites : int)`

- O construtor irá receber o nome do arquivo, o tempo do frame e o número de frames do sprite sheet.
- Primeiro, ele passa o nome do arquivo para o construtor do pai, que irá tratar de carregar a imagem.
- Então ele deve inicializar os atributos frameTime e n de acordo com os valores dos parâmetros passados.
- A seguir, ele determina os novos valores de rect.w e rect.h, que deverão ser os tamanhos (surface->w)/n e surface->h, respectivamente.
- Então ele determina os valores da clipRect, que é o retângulo que diz qual área da surface será blitada. Os valores x e y devem ser inicializados com 0 e os valores w e h devem ser inicializados com os mesmos valores de rect.w e rect.h, respectivamente.
- O atributo tempo deve ser inicializado com 0.

`+ update(dt : int) : void`

O método update deve ser responsável por identificar qual o valor de x de clipRect, ou seja, qual a área da surface será blitada e, por consequência, qual frame será desenhado na tela. Para isso, ele recebe dt. O algoritmo é o seguinte:

- Some dt ao tempo;
- Divida a variável do tempo pelo valor do tempo do frame, eo quociente da divisão deve ser atribuído à variável saltoFrame, que é justamente o número de frames que devem ser saltados.
  A explicação é a seguinte: consideremos que o tempo de frame seja 30. Se a variável de tempo estiver entre 30 e 59, significa que teremos de pular um frame. Agora se estiver entre 60 e 89, significa que teremos de pular dois frames.
- O resto da divisão deve ser atribuído a própria variável de tempo.
  A explicação é a seguinte: suponha que o tempo de frame seja 30. Se a variável de tempo estiver em 30, significa que saltoFrame deverá ser 1 e o tempo deve ser resetado para 0, que é justamente o resto da divisão. Se for 35, por exemplo, o saltoFrame será 1 e o tempo será resetado para 5.
  No caso de o tempo ser menor do que 30, como 20, por exemplo, o resto continuará sendo 20, então isso não é problema.

- O valor de clipRect.x deve ser incrementado em clipRect.w*saltoFrames, o que fará com que o srcrect mova para frente, seguindo para a próxima animação, saltando um número “saltoFrames” de vezes.
- Se clipRect.x >= surface->w, então significa que o srcrect passou do espaço da sheet. Então, devemos fazer clipRect.x = (clipRect.x – surface->w) % surface->w, ou seja, o valor de x de clipRect retorna para a posição apropriada.
- Por fim, fazemos o clip da imagem na posição calculada.

`+ setFrame(frame : int) : void`

OPCIONAL: método para definir o próximo sprite a ser renderizado.

Para isso, basta ajustar clipRect.x para a posicao desejada (clipRect.w * frame) e fazer o clip.

`+ setFrameTime(time : int) : void`

OPCIONAL: Redefine o valor de frametime. Deve checar se o novo valor é <= 0, pois o valor deve ser no mínimo 1 (ou teremos uma divisão por 0).

`+ getframeTime() : int`

OPCIONAL: retorna o frametime

3. Classe GameObject: Implementando Colisão

GameObject
+ box : Rect
+ GameObject(x : float, y : float, w : float, h : float) [...] + collidesWith(GameObject * other) : void

Vamos agora implementar a colisão por caixas nos nosso gameObjects. Para isso, usaremos um Rect box, que guardará a posição (x,y) dos objetos e seu tamanho (w,h). Como no mínimo as posições DEVEM ser floats, sugerimos a criação de uma classe Rect no seguinte modelo:

```
class Rect
{
public:
	float x, y;
	float w, h;

	Rect(float x = 0, float y = 0, float w = 0, float h = 0) :
			x(x),y(y), w(w), h(h)
	{	}

};
```

Quem quiser pode implementar essa classe como um template:
http://www.cplusplus.com/doc/tutorial/templates/

Alteração nos atributos da classe:

`+ box : Rect`

Armazena a posicao e o tamanho do objeto.

Novos/ Alterações nos métodos da classe:

`+ GameObject(x : float, y : float, w : float, h : float)`

O constructor deve ser alterado para receber w e h.

Ou seja, todos os filhos devem passar para o gameObject seus respectivos tamanhos de w e h, de acordo com o tamanho da sprite.

IMPORTANTE:

Essa modificação por si só vai alterar todos os objetos filhos do gameObject (Terra, Lua, planet, redplanet, followerObject).

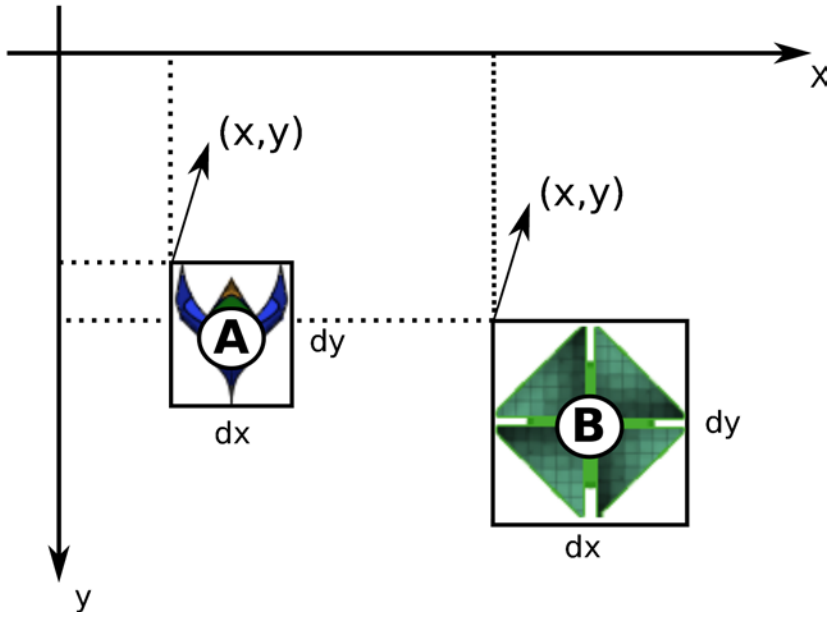
Aconselho fortemente que vocês corrijam todos problemas causados por essa troca de estrutura antes de proseguirem com o laboratório. Ou seja, como diz a boa prática de programação, NÃO CONTINUEM enquanto seu código não compilar!

PS.: Não usamos essa estrutura a priori pois nós (monitores) também não vimos a utilidade dela quando começamos a programar. Pequena falha de arquitetura nossa, eu diria.

`+ collidesWith(GameObject  * other) : void`

Método de colisão por caixas é um método que leva em consideração um retângulo que se faz em torno do objeto, observando seus píxeis. Se estes retângulos se interceptarem, então ocorreu colisão.

Como calcular o retângulo precisamente é um pouco complicado, pois torna-se necessário observar todos os píxeis da imagem, usaremos o retângulo do objeto (box) mesmo, calculado com base nas dimensões da imagem.



Para calcular a colisão entre dois retângulos, faremos uma série de testes para verificar que eles **não** interceptam, cobrindo todas as possibilidades. Se todos estes testes falharem, significa que ocorreu a intercepção.

Sejam dois retângulos A = (X_A, Y_A, dx_A, dy_A) e B = (X_B, Y_B, dx_B, dy_B).
$diff_X = X_A - X_B$ e $diff_Y = Y_A - Y_B$.

O algoritmo para calcular intercepção entre os retângulos A e B é o seguinte:

Se $diff_X > dx_B$ ou $-diff_X > dx_A$, retorna falso.
Se $diff_Y > dy_B$ ou $-diff_Y > dy_A$, retorna falso.
Se falhar em todos estes testes, então retorna verdadeiro.

4. Classe AccObject: Movimento Acelerado.

AccObject : GameObject
animation : Animation # speed : float # acceleration : float # rotation : float + hitPoints : int
+ AccObject(animation : Animation, x : float, y : float, w : float, h : float, hitPoints : int) + render(camera : float, camraY : float) : void + update(dt : int) : int

Vamos agora implementar um objeto que usa tudo que nós implementamos até agora e ainda se movimenta com movimento acelerado.

Sobre os atributos da classe:

`# animation : Animation`
Animação usada pelo objeto.
`# speed : float`
Velocidade atual do objeto
`# acceleration : float`
Aceleração do objeto
`# rotation : float`
Ângulo que indica a direção do objeto
`+ hitPoints : int`
Vida do objeto

Sobre os métodos da classe:

`+ AccObject(animation : Animation, x : float, y : float, w : float, h : float, hitPoints : int)`
O construtor da classe. Inicializa a animação, a box (x,y,w,h) de acordo com os parâmetros do construtor. box.w e box.h devem ser inicializados de acordo com getClip().w e getClip.h() da sprite/animation.
Alem disso, speed, acceleration e rotation devem ser inicializadas com 0, e hitPoints com hitPoints.

`+ render(camera : float, camraY : float) : void`
Renderiza a animation. Simples assim.

`+ update(dt : int) : int`

Para que possamos implementar movimento acelerado no nosso objeto, lembremos de algumas fórmulas de física:

$$S = S_0 + v\,\Delta t \quad \text{(I)}$$, onde

S = espaço final	(m)
S_0 = espaço inicial	(m)
v = velocidade	(m/s)
Δt = diferença de tempo	(s)

Adaptando-a ao universo dos games, teríamos:

S = posição do frame atual	(pixel)
S_0 = posição do frame anterior	(pixel)
v = velocidade	(pixel/ms)
Δt = tempo entre um frame e outro	(ms)

Em um movimento acelerado, a velocidade teria um valor variável, o qual é determinada por:

$$v = v_0 + a\,\Delta t \quad \text{(II)}$$, onde

v = velocidade final	(m/s)
v_0 = velocidade inicial	(m/s)
a = aceleração	(m/s^2)
Δt = diferença de tempo	(s)

No universo dos games, temos:

v = velocidade do frame atual	(pixel/ms)
v_0 = velocidade do frame anterior	(pixel/ms)
a = aceleração	($pixel/ms^2$)
Δt = tempo entre um frame e outro	(ms) = 1

Um objeto que se move por movimento acelerado tem, naturalmente, os atributos:

`float x, y, vx, vy, ax e ay;`

Seu método update será dividido em dois passos.

a. **Primeiro devemos calcular a nova posição do objeto.**
   Para um caso genérico, o algoritmo seria:

```
Decide-se quais os valores de ax e ay, ou de vx e vy;
Atualiza-se o valor da velocidade e da posição, nessa ordem, ou seja:
vx = vx + ax*dt;
vy = vy + ay*dt;
x = x+ vx*dt;
y = y + vy*dt;
```

No nosso caso específico, temos que:

- Checar o input do teclado, como fazíamos antigamente com a Terra.
- Caso seja pressionado ‘a’ e ‘d’, a rotação do objeto deve ser alterada, com uma velocidade angular arbitrária porém sempre dependente de dt (usamos dt/10 no exemplo).
- Caso seja pressioando ‘w’ ou ‘h’, a aceleração deve ser definida com outro valor arbitrário sempre dependente de dt (usamos 3*dt/1000 no exemplo).
- Calculamos então a velocidade do objeto, usando a fórmula citada acima.
- Por fim temos que a nova posição do objeto será determinada pelo ângulo do objeto. Para isso, lembrando um pouco de trigonometria, usaremos as seguintes fórmulas:

```
x += v *  cos((rotation -90) * π/180)
y += v * -sin((rotation -90) * π/180)
```

b. Então devemos atualizar sua animação:

Simples: atualizar a animação (update(dt)) e atualizar sua rotação (rotoZoom(rotation)).

5. Alterações no GameManager: Hora de transformar isso num jogo!

IMPORTANTE:

Todas as alterações no GameManager aqui descritas são sugestões que terminarão implementando o exemplo mostrado em sala. Sinta-se livre para implementar uma mecânica diferente no seu trabalho, contanto que você demonstre todas as funcionalidades implementadas (**zoom, rotação, Animação, Rotação + Animação, Colisão e movimento acelerado**).

Por exemplo, ao invés da nave chegar na Terra e vencer, o UFO poderia chegar na terra e destruí-la, assim o objetivo da Ship seria matar o UFO para que ele não destrua a Terra.

Antes de alterar o Gamemanager, temos que remover o comportamento do objeto Earth. Para isso, apenas esvazie seu método update: a Terra não precisa fazer nada (por enquanto).

Agora sim, na classe GameManager, instancie um AccObject chamado ship e sua devida animação e crie uma animação chamada boom. Declare também 2 floats, boomX e boomY;

\+ GameManager()

- Acrescente seus devidos includes
- Declare os objetos no header.inicialize-os no construtor da GameManager e destrua-os no destrutor. Defina a vida da Ship como 20. Inicialize boomX e boomY com 0;
- Assim que você carregar a sprite da terra, antes de construir o objeto earth, aplique nela um zoom de 3x.
- Agora defia a posição inicial da terra como uma posição aleatória, entre -1000 e 3000.
- Defina as posições iniciais da Ship e do UFO bem distantes, algo como 400px de distância.

\- processEvents()

- Altere a tecla de criação de planetas (dentro de process events) para mouse1.
- Remova ou desative o dano nos planetas através de input.
- Acrecente um check para ver se o UFO != NULL antes de tentar enfileirar posições.
- Remova ou desative todo e qualquer controle de câmera via input.

\+ update(int dt)

- Teste se Ship e UFO são válidos , e então atualize a posição da câmera., fazendo com que a câmera fique centralizada na Ship todo o tempo.
- Teste se Ship e UFO são válidos e dê seus respectivos updates.
- Dê o update na boom.
- Por fim, chame o método checkCollision(dt). Veja descrição abaixo.

Crie um método: privado checkCollision(int dt); Este método vai ser o responsável por tratar todas as colisões do jogo, logo, por toda a lógica de vitória e derrota.

- checkCollision(int dt)
  - Todo o teste de de checagem de colisão deve ser desabilitado caso ou o UFO ou a Ship não exitam. Ou seja: Se UFO e Ship são válidos:
    - Remova ou desative o método checkPlanets.
    - Testa a colisão da Ship contra cada um dos planetas. Caso seja true, altere a posição de boomX e boomY para para a posição do planeta e remova o planeta do vector. Além disso, diminua os hitPoints da Ship em 1.
    - Teste a colisão da Ship com o UFO. Caso seja true, defina os hitPoints da Ship como 0.
    - Teste os hitpoints da Ship. Se for <= 0, o UFO ganhou.
      - Defina a posição de boomX e boomY como a posição da Ship.
      - Delete a Ship, e torne seu ponteiro nulo.
      - De um update na boom.
      - Return;
    - Teste a colisão da Ship com a Earth. Se for true, a Ship ganhou.
      - Defina a posição de boomX e boomY como a posição do UFO.
      - Delete o UFO, e torne seu ponteiro nulo.
      - De um update na boom.
      - Return;

+ render()
  - Teste se o UFO é válido e renderize tanto se ele quanto seu caminho.
  - Teste se a Ship é válida e renderize-a.
  - Teste se a posição da boom é != (0,0), e então renderize-a na posição boomX e boomY;

6. Extras... pra quem ainda está vivo

- 0,25 pt - Animação de boost na Ship.
- 0,50 pt - Animação de virada na Ship.
- 0,75 pt - Implementar atrito (fazer a velocidade da nave tender a 0 quando não há aceleração)