# 1. City Saver

O jogo denominado como *City Saver* é um remake básico e rudimentar do jogo R-type. A história de City Saver passa em um cenário global prestes a ser destruído, onde cientistas descobriram um grande asteroide vindo para a terra e o destruíram em inúmeros pedaços. O dever do jogador é eliminar cada pequeno asteroide usando as armas da nave, desviando dos obstáculos rumo ao infinito.

# 2. Documentação

## 2.1. Requisitos

* MinG 4.7.0
* Allegro 5.0.10

## 2.2. Bibliotecas

Allegro5 e suas dependências (Imagens, áudio, fontes, etc), stdio, time, stdlib, math.

## 2.3. Lógica de implementação

O corpo principal do jogo foi implementado no arquivo principal, *main.c*, onde é carregado as instancias necessárias para rodar o jogo e a lógica principal. Todas as dependências foram divididas baseadas em tipos e alocadas dentro da pasta src:

* Configurações: configurações principais do jogo e da aplicação, são alocadas na pasta config.
* Data: arquivos para armazenamento de dados da aplicação, são alocadas na pasta data.
* Eventos: bibliotecas pessoais criadas para uso dos eventos do Allegro e do jogo, são alocadas na pasta events.
* Fontes: arquivos de fontes de texto para uso na aplicação, são alocadas na pasta fonts.
* Imagens: todo e qualquer tipo de imagem utilizado no jogo, são alocadas na pasta images.
* Objetos: todo e qualquer coisa na tela, sejam tratativas para o usuário ou objetos em si, são alocadas na pasta objects.
* Sons: todo e qualquer tipo de arquivo relacionado ao áudio do jogo, são alocadas na pasta sounds.
* Utilidades: toda e qualquer biblioteca usada para a manipulação de dados de forma personalizada ao longo da aplicação, são alocadas na pasta utilities.
* Extensões: toda e qualquer extensão de alguma dependência criada, para fins de estender e melhorar o funcionamento da mesma posteriormente, são alocadas dentro das pastas anteriores, quando usadas, dentro de uma pasta denominada extended.

Para carregar todas essas dependências de forma rápida e organizada, foi criado um arquivo dentro da pasta src, sem nenhuma sub-pasta, denominado autoload.h. Este arquivo é o único include dentro da main.c, onde o mesmo faz todos os includes necessários de dependências do projeto de forma automatizada e organizada.

Os includes do autoload são feitos de forma sequencial, onde:

* Inclui as bibliotecas do Allegro.
* Inclui as bibliotecas nativas usadas.
* Inclui as bibliotecas de utilidades.
* Inclui as bibliotecas de configurações.
* Inclui as bibliotecas de objetos e suas respectivas extensões.
* Inclui as bibliotecas de eventos e suas respectivas extensões.

### 2.3.1. Utilidades

#### 2.3.1.1 arquivo.h

Funções relacionadas a manipulação de arquivos.

**bool escreveInteiroArquivo(const char \* localizacao, int valor);** : recebendo a localização do arquivo o valor a ser escrito dentro do mesmo, sobrescreve o conteúdo do arquivo por esse valor, retornando true, se foi feio com sucesso, e false, se deu algo errado.

**int leInteiroArquivo(const char \*localizacao);** : recebendo a localização de um arquivo contendo um inteiro, retornará o valor do inteiro se for lido com sucesso ou zero se algo der errado na leitura.

#### 2.3.1.2. string.h

Funções relacionadas a manipulação de strings.

**char \*concat(const char \*s1, const char \*s2);** : recebendo duas strings, concatena a primeira com a segunda e retorna a string concatenada.

#### 2.3.1.3. graus.h

Funções relacionadas a manipulação de graus.

**double to\_radians(double degrees);** : converte um valor em graus para radianos

#### 2.3.1.3. bitmap.h

Funções relacionadas a manipulação de bitmaps.

**ALLEGRO\_BITMAP \*load\_bitmap\_at\_size(const char \*filename, int w, int h);** : recebendo a localização do arquivo, junto com as dimensões de largura e altura desejadas, retorna um bitmap do alegro no tamanho informado.

#### 2.3.1.4. rand.h

Funções relacionadas a manipulação de números aleatórios.

**int randInt(int n);** : Retorna um inteiro aleatório entre 0 e n-1.

**int randIntMinMax(int min, int max);** : Retorna um inteiro aleatório entre o valor mínimo e máximo informado.

**double randDouble();** : retorna um double aleatório entre 0 e 1.

**double randDoubleMinMax(double min, double max);** : retorna um double aleatório entre o valor mínimo e máximo informado.

### 2.3.2. Configurações

Funções e inicializações de configurações primordiais para o funcionamento da aplicação.

#### 2.3.2.1. config.h

**Declaração de variáveis globais**: linha 01 até linha 196.

**int iniciarAllegro();** : função onde instala e inicia todos os módulos do Allegro. Local também onde é instanciado os valores de variáveis globais que usam bitmaps, fontes, etc.

**int finalizaAllegro();** : realiza os procedimentos de fim de jogo do Allegro, como finalizar localizadores de eventos (timer, display, fila de eventos).

### 2.3.3. Objetos

Funções e inicializações dos objetos para o jogo e suas respectivas tratativas.

#### 2.3.3.1. background.h

Funções e tratativas relacionadas ao fundo do jogo.

**struct Background** : Struct onde são armazenadas os bitmaps do fundo e seus clones, posições de ambos, tipos, etc.

**void iniciarBackground()**; : inicia um array de struct de backgrounds, definindo suas posições, imagens e bitmaps.

**void defineVelocidadeBaseadoNoTipo();** : define a velocidade do fundo baseado no tipo da instancia da struct. O tipo diz respeito a qual imagem é.

**void finalizaBackground();** finaliza os bitmaps de background.

**void calculaBackground();** : função que calcula a movimentação do fundo.

**void redesenhaBackground();** : função onde redesenha o fundo animado do jogo baseado nas atualizações calculadas.

#### 2.3.3.2. pontuacao.h

Funções e tratativas relacionadas a pontuação e ao recorde do jogo.

**void redesenhaPontuacao();** : redesenha a pontuação na tela baseado nos novos pontos.

**void resetaPontuacao();** : calcula se o usuário bateu o recorde, resetando a pontuação recorde do usuário e salvando-o a no arquivo caso a pontuação tenha batido o recorde.

#### 2.3.3.3. bloco.h

Funções e tratativas relacionadas ao bloco.

**struct Bloco** : estrutura onde contém todos os dados do bloco, como bitmap, posições em x e y, largura e altura, velocidade, etc.

**void iniciaBloco();** : inicia o bloco e suas características de forma aleatórias, mas baseado na altura mínima e máxima, largura mínima e máxima, de forma randômica entre esses valores.

**void finalizaBloco();** : destrói o bitmap do bloco.

#### 2.3.3.4. extended/explosaoAsteroides.h

Funções e tratativas relacionadas às explosões.

**struct Explosao** : estrutura das explosões, contém dados com status de ativo ou inativo, posições em x, y e a variável de controle de animação das explosões criadas.

**void redesenhaExplosaoAsteroides();** : Verifica se alguma instancia de explosão está ativa, caso esteja, inicia ou continua a animação da mesma, redesenhando-a de acordo com o passar dos frames.

**void iniciarExplosoes();** : incia o array de estrutura de explosões, no qual seta todos os valores como valores padrões.

**void criaExplosao(double x, double y);** : cria uma explosão nas posições x e y informadas.

**void limpaExplosao(int indice);** : limpa o índice de uma explosão, retornando-a para seus valores padrões

#### 2.3.3.5. asteroid.h

Funções e tratativas relacionadas aos asteroides.

**struct Asteroid** : estrutura dos asteroides, onde contém informações como controle de colisão, tipo do asteroide, velocidade, largura, altura, posições x e y, status, etc.

**void iniciaAsteroids();** : inicia o desenho dos asteroides baseado no tipo calculado.

**void calculaAsteroids();** : gera os valores dos asteroides, aleatoriamente e baseado no tipo aleatório definido.

**void redesenhaAsteroids();** : redesenha os asteroides na tela, baseando-se nos cálculos de movimentação, cálculos de colisão, baseando-se no tipo e status do asteroide, identificando se o mesmo precisa ser reposicionado ou não.

**void movimentacaoAsteroid(int indice, int tipo);** : calcula a movimentação do asteroide, baseada no tipo e velocidade.

**void reposicionaAsteroid(int indice);** : reposiciona o asteroide, verificando se o mesmo saiu o suficientemente da tela, considerando se o mesmo irá colidir com algum asteroide já existente ou com algum bloco ao ser reposicionado.

**bool verificaSeVaiColidirComAlgumAsteroid(int indice);** : faz o cálculo probabilístico de colisão de asteroide com asteroide e asteroide com bloco. Verifica se os asteroides já não colidiram entre si ou o asteroide com o bloco já não colidiu entre si, verifica também a colisão futura com blocos e asteroides. Foi utilizada função horária do espaço do movimento retilíneo uniforme (MU em física) para calcular possíveis colisões futuras de blocos e asteroides, devido a diferença de velocidade entre ambos e entre si. Tudo isso foi combinado a um timer e quantidade de tentativas. Caso após reposicionar o asteroide 5 vezes o mesmo continue colidindo ou com probabilidade de colidir futuramente, configura um timer para tentar novamente e passar para o próximo, evitando loops infinitos de tentativas.

**void verificaColisaoComBloco(int indice);** : verifica possível colisão dos asteroides entre si e os asteroides com o bloco, criando uma explosão caso os mesmo tiverem colidido.

**void finalizaAsteroids();** : destróis os bitmaps dos asteróides.

#### 2.3.3.6 extended/calculosBloco.h

Funções e tratativas relacionadas aos cálculos e movimentações de blocos.

**void redesenhaBloco()**; : redesenha o bloco na tela baseado no calculo de seu movimento e no seu status.

**void calculaMovimentoBloco();** : calcula o movimento do bloco, baseando-se em sua velocidade e status, verificando se o mesmo está na tela.

**void reiniciaBloco()**; : reinicia o bloco com valores padrões e aleatórios.

**bool calculaColisaoBlocoAsteroid();** : faz o cálculo probabilístico de colisão de bloco com asteroide. Verifica se o bloco já não colidiu com algum asteroide, verifica também a colisão futura com blocos e asteroides. Foi utilizada função horária do espaço do movimento retilíneo uniforme (MU em física) para calcular possíveis colisões futuras de blocos e asteroides, devido a diferença de velocidade entre ambos e entre si. Tudo isso foi combinado a um timer e quantidade de tentativas. Caso após reposicionar o asteroide 5 vezes o mesmo continue colidindo ou com probabilidade de colidir futuramente, configura um timer para tentar novamente e passar para o próximo, evitando loops infinitos de tentativas.

**void verificaSeOBlocoEstaNaTela();** : verifica se o bloco ainda está na tela, caso o mesmo não estiver na tela, será feito a tentativa de reiniciar o mesmo.

#### 2.3.3.7. extended/gameOverReinicia.h

Funções e tratativas relacionadas aos cálculos de passagem de tempo após o jogador perder e reiniciar o jogo.

**bool verificaGameOverReinicia();** : Verifica se o jogador está no tempo pós reinicial, se sim, incrementa o contador de tempo. Quando o contador de tempo chegar no limite, mudara a flag de tempo pós reinicial para falso. Imprescindível para impedir tiros enquanto o jogo ainda não se reiniciou por completo.

#### 2.3.3.8. tiro.h

Funções e tratativas relacionadas aos cálculos, movimentação e interações dos tiros.

**struct Tiro** : estrutura do tiro, onde armazena as informações como colisão, se está ativado ou não, posição x e y, largura, altura, velocidade, etc.

**void disparaTiro();** : quando chamada, verificará quanto tempo o jogador apertou o espaço, verificará o tipo de tiro respectivo e o disparará.

**void finalizaTiro();** : finaliza os bitmaps e os samples do tiro.

**void teclasTiro(int tecla, int tipoEvento);** : identifica a tecla apertada de acordo com o tipo do evento de tecla, calculando assim, quanto tempo a tecla de tiro (espaço) foi pressionada e ativando a flag de tiro quando a tecla de espaço for solta.

**void calculaMovimentoTiro();** : calcula a movimentação do tiro baseando-se se o mesmo está na tela, se ele já colidiu com algum objeto e movimentando-o se o mesmo não colidiu e está ativado, incrementando suas posições ou decrementando-o as baseando-se na posição da nave ao atirar o mesmo e sua categoria (simples ou avançado).

**void redesenhaTiro();** : redesenha tiro baseado nas animações, se precisa atirar um tiro novo ou não, baseando-se também na movimentação, inclinação ou se o tiro colidiu com algo, iniciando assim uma explosão.

**void mudaAnimacaoTiro(int i);** : muda a animação do tiro ao passar do tempo, garantindo um ar melhor de movimentação do mesmo.

**void verificaSeOTiroEstaNaTela(int i);** : verifica se o tiro ainda está na tela, caso não estiver, desativa o mesmo.

**int verificaSeTodosOsTirosEstaoDesativados();** : verifica se todos os tiros estão desativados, se sim, retorna 1, caso contrário, retorna 0.

**void limpaTiros();** : limpa todos os tiros e a fila de tiros com os valores padrões.

**void colisaoTiros(int j);** : verifica se o tiro colidiu com algum objeto na tela, adicionando pontuações e/ou removendo objetos que colidiram com o tiro na tela segundo restrições.

**void animacaoExplosao(int i);** : cria a animação de explosão do tiro e a reproduz.

**void finalizaExplosao();** : destrói os bitmaps e samples de explosões.

**void limpaTiro(int i);** : limpa o valor de um tiro com valores padrões.

**void animacaoTiroIniciando();** : cria a animação do tiro sendo carregando a medida que o usuário segura a tecla de espaço.