



## Train

Antonio Carlos Falcão Petri 586692  
José Antônio dos Santos Júnior 586765  
José Vitor de Carvalho Aquino 609170  
Tiago Bonadio Badoco 586722

São Carlos  
Junho/2015

### **Autores:**

Antonio Carlos Falcão Petri ([falcaopetri@gmail.com](mailto:falcaopetri@gmail.com)) – Implementação do Jogo  
José Antônio dos Santos Júnior ([jusantosjr@hotmail.com](mailto:jusantosjr@hotmail.com)) – Desenvolvimento Gráfico  
José Vitor Aquino ([jvcaquino95@gmail.com](mailto:jvcaquino95@gmail.com)) – Implementação das Estruturas de Dados  
Tiago Bonadio Badoco ([tiago.badoco@gmail.com](mailto:tiago.badoco@gmail.com)) - Documentação

### **Funcionamento do Jogo:**

No jogo, tem-se como objetivo juntar os vagões de trem à locomotiva de acordo com a ordem determinada pela estação. Para isso, existe uma fila de vagões onde, em ordem, tendo como primeiro elemento o vagão 1 e como último elemento o vagão . A ordem que os vagões devem aparecer no final (ou seja, ligados à locomotiva) é determinada de forma aleatória e aparece na placa sobre a estação de trem.

Para montar as sequências os jogadores podem passar um trem diretamente ou então empilhar ele na plataforma de vagões. Só é possível retirar da plataforma de vagões o último vagão adicionado a ela. Para efetuar tais ações temos dois semáforos e um guindaste. O semáforo da esquerda permite os vagões irem para o centro do cenário onde podem ser levados a plataforma por meio do guindaste ou podem passar para se acoplar à locomotiva, isso se o semáforo da direita for acionado. Só é possível retirar um vagão da plataforma se não houver nenhum vagão posicionado no centro do cenário.

Nem todas as sequências propostas pela estação são factíveis. Caso o jogador se depare com uma sequência que não pode ser efetuada ele deve clicar no botão “Impossível montar a sequência” localizado no canto superior direito. Caso a sequência seja factível o jogador deve acoplar os vagões

adequadamente à locomotiva. Todo esse processo deve ser feito em 60 segundos sendo que, se o tempo estourar, é considerada uma derrota.

Esse jogo é uma leve adaptação da situação proposta no [ExercícioRails- 1062](#), descrito no URI Online Judge. De fato, a ideia de criar o TraIn veio durante a tentativa de resolução de tal exercício.

#### Controles:

Mouse – O mouse controla todas as interações com os menus do jogo, selecionando as opções desejadas.

Setas – Movimentam o guindaste.

“A” – Ativa o semáforo da esquerda, permitindo a ida dos vagões da fila para a posição central do cenário.

“D” – Ativa o semáforo da direita, permitindo o acoplamento dos vagões posicionados no centro do cenário.

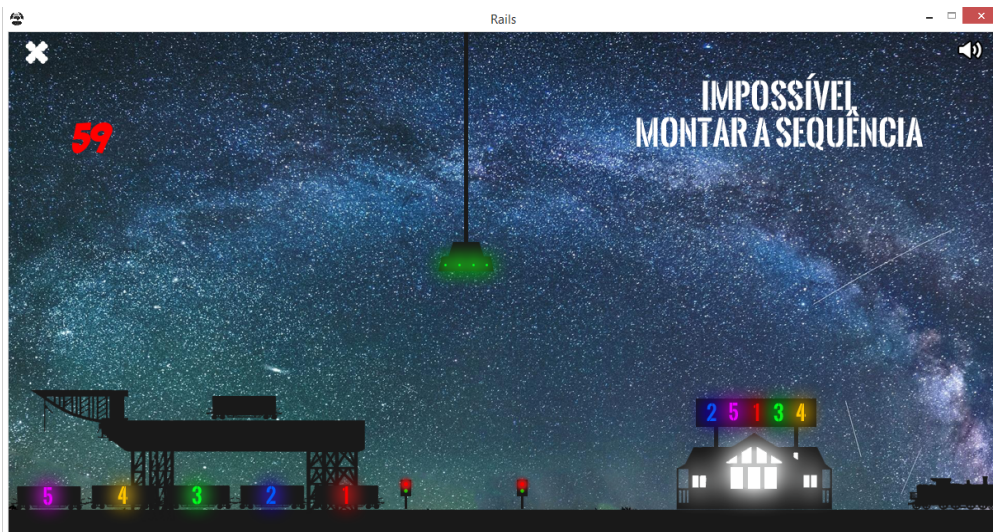


Imagem 1 - Tela do Jogo

#### Ferramentas:

- Codeblocks;
- Allegro 5.0;
- Photoshop CS5;
- Astah Professional;
- Fruity Loops Producer Edition 11;
- STL, C++ Standard Template Library.

#### TAD's

A implementação do jogo utilizada utiliza-se de 3 estruturas de dados para organizar os principais elementos do jogo. São elas: um deque encapsulado em uma estrutura Fila e em uma Pilha, e um Map.

É importante ressaltar que as duas estruturas implementadas utilizam o conceito de Templates, mantendo a camada de abstração entre a estrutura e os dados que ela armazena. Além disso, teve-se como parâmetro de implementação as estruturas da STL, referências em C++.

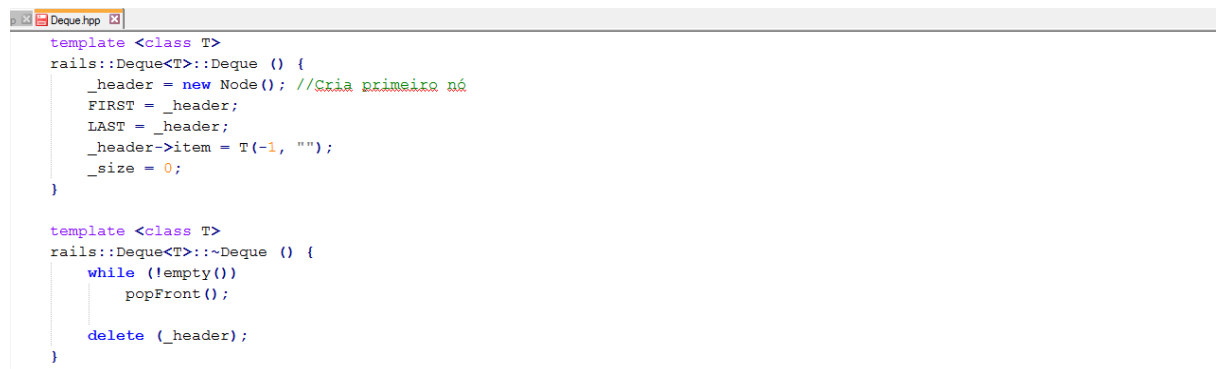
## Deque

**Double-ended queue** ou fila duplamente encadeada, funciona como um container que possui tamanho dinâmico e pode ser expandido ou contraído em ambas as extremidades, dessa forma possuindo métodos de inserção, exclusão e recuperação da informação em ambas as extremidades.

### Como o Deque é implementado?

O Deque é utilizado tanto para auxiliar na construção de outras estruturas utilizadas no jogo quanto como um tipo de Lista. Ela é implementada para inserir elementos de forma dinâmica e duplamente encadeado.

As funções de deque utilizadas e implementadas no jogo foram: Deque(), ~Deque(), pushFront(), pushBack(), popFront(), popBack(), front(), back(), empty(), at().



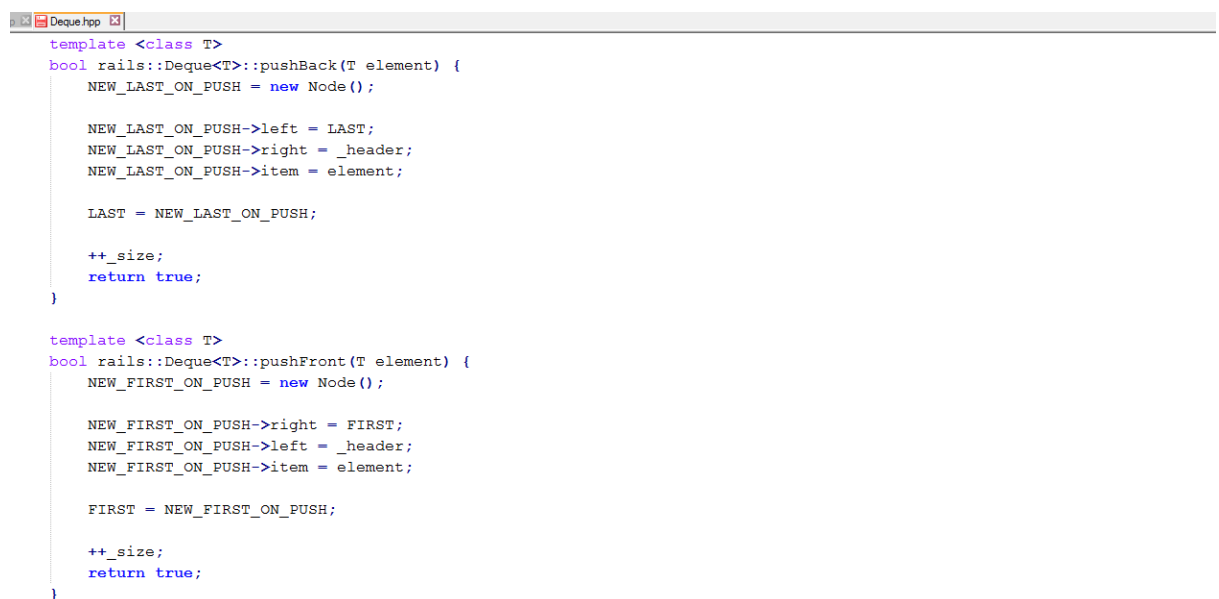
```
template <class T>
rails::Deque<T>::Deque () {
    _header = new Node(); //Cria primeiro nó
    FIRST = _header;
    LAST = _header;
    _header->item = T(-1, "");
    _size = 0;
}

template <class T>
rails::Deque<T>::~~Deque () {
    while (!empty())
        popFront();
    delete (_header);
}
```

Imagem 2 – Construtor e destrutor

O método construtor cria um novo nó em Header. Como a deque está sendo inicializado, o nó a direita(first) e o a esquerda(last) do Header será ele mesmo.

O método destrutor, pensando na reusabilidade e portabilidade, utiliza a função popFront() para liberar a memória dos elementos do deque. Quando restar apenas o nó Header, este é deletado como os outros nós.



```
template <class T>
bool rails::Deque<T>::pushBack(T element) {
    NEW_LAST_ON_PUSH = new Node();

    NEW_LAST_ON_PUSH->left = LAST;
    NEW_LAST_ON_PUSH->right = _header;
    NEW_LAST_ON_PUSH->item = element;

    LAST = NEW_LAST_ON_PUSH;

    ++_size;
    return true;
}

template <class T>
bool rails::Deque<T>::pushFront(T element) {
    NEW_FIRST_ON_PUSH = new Node();

    NEW_FIRST_ON_PUSH->right = FIRST;
    NEW_FIRST_ON_PUSH->left = _header;
    NEW_FIRST_ON_PUSH->item = element;

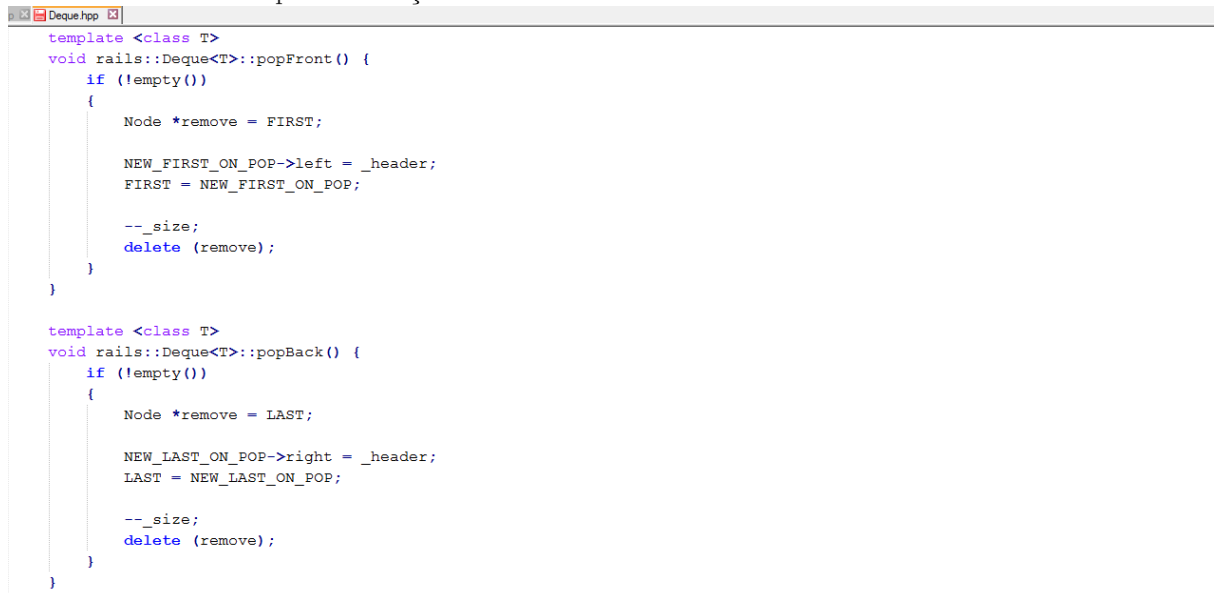
    FIRST = NEW_FIRST_ON_PUSH;

    ++_size;
    return true;
}
```

Imagem 3 – pushBack, pushFront

Pensando que a estrutura Deque é duplamente encadeada, foram criados métodos pushBack() e pushFront(), onde podem ser utilizados para inserir valores tanto no início (pushFront), quanto no final da estrutura (pushBack).

É utilizado dois define's para a implementação dessas funções, eles serão responsáveis por apontar início e fim do deque na inserção.



```
template <class T>
void rails::Deque<T>::popFront() {
    if (!empty())
    {
        Node *remove = FIRST;

        NEW_FIRST_ON_POP->left = _header;
        FIRST = NEW_FIRST_ON_POP;

        --_size;
        delete (remove);
    }
}

template <class T>
void rails::Deque<T>::popBack() {
    if (!empty())
    {
        Node *remove = LAST;

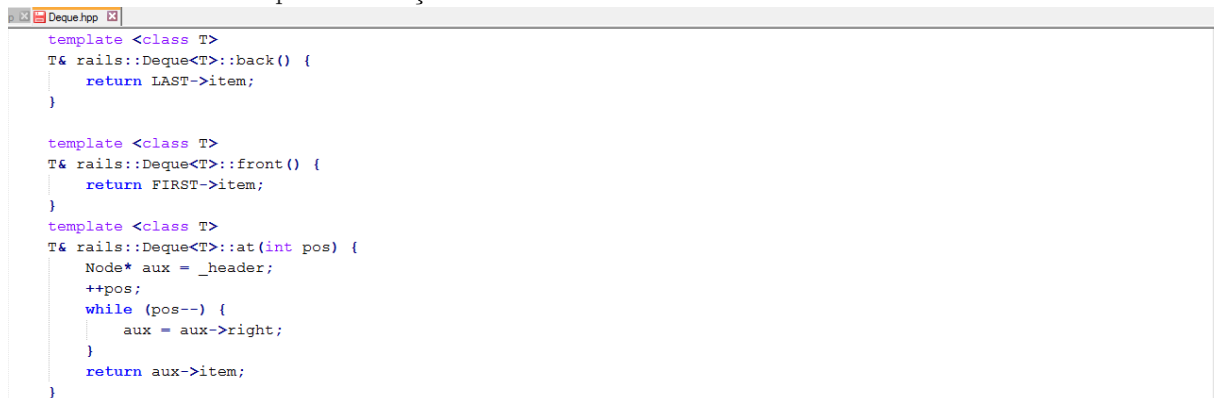
        NEW_LAST_ON_POP->right = _header;
        LAST = NEW_LAST_ON_POP;

        --_size;
        delete (remove);
    }
}
```

Imagem 4 – popBack, popFront

Seguindo o conceito de estrutura duplamente encadeada, foram criados métodos popBack() e popFront(), onde podem ser utilizados para remover valores tanto no início (popFront), quanto no final da estrutura (popBack).

É utilizado dois define's para a implementação dessas funções, eles serão responsáveis por apontar início e fim do deque na remoção.



```
template <class T>
T& rails::Deque<T>::back() {
    return LAST->item;
}

template <class T>
T& rails::Deque<T>::front() {
    return FIRST->item;
}

template <class T>
T& rails::Deque<T>::at(int pos) {
    Node* aux = _header;
    ++pos;
    while (pos-- > 0) {
        aux = aux->right;
    }
    return aux->item;
}
```

Imagem 5 – back, front, e at

Back() e Front() são implementadas utilizando os defines first e last, para setar o início e o fim da estrutura, essas funções retornam o valor do primeiro elemento (front), e último (back).

Imagem 6 – erase, size

```

Deque.hpp
template <class T>
void rails::Deque<T>::erase(int pos) {
    Node* headerCopy = _header;

    Node* aux = _header;
    ++pos;
    while (pos--) {
        aux = aux->right;
    }

    _header = aux->left;
    popFront();

    _header = headerCopy;
}

template <class T>
unsigned int rails::Deque<T>::size() {
    return _size;
}

template <class T>
bool rails::Deque<T>::empty() {
    if (FIRST == _header)
        return true;
    else
        return false;
}

```

### Como a Fila é implementada e utilizada?

A Fila é utilizada para armazenar os vagões de 1 a 5 em ordem crescente. Ela é implementada para inserir elementos de forma dinâmica e encadeada. Assim, surge a utilização do Deque, onde todas as funções necessárias para manipulação de uma fila já foram implementadas.

As funções de fila utilizadas e implementadas no jogo foram: Queue(), ~Queue(), push(), pop(), front(), e empty().

```

Stack.hpp  Queue.hpp
template <class T>
rails::Queue<T>::Queue() {
}

template <class T>
rails::Queue<T>::~~Queue() {
}

template <class T>
void rails::Queue<T>::pop() {
    _deque.popFront();
}

template <class T>
bool rails::Queue<T>::push(T element) {
    return _deque.pushBack(element);
}

template <class T>
T& rails::Queue<T>::front() {
    return _deque.front();
}

template <class T>
bool rails::Queue<T>::empty() {
    return _deque.empty();
}

```

Imagem 7 – Construtor, destrutor, push, front, e empty

### Como a Pilha é implementada e utilizada?

A Pilha é utilizada para armazenar os vagões que são tirados da fila inicial. Ela é implementada para inserir elementos de forma dinâmica e encadeada. Assim, surge a utilização do Deque, onde todas as funções necessárias para manipulação de uma pilha já foram implementadas.

As funções de fila utilizadas e implementadas no jogo foram: Stack(), ~Stack(), push(), pop(), top(), e empty().

```

Stack.hpp Queue.hpp
template <class T>
rails::Stack<T>::Stack() {
}
template <class T>
rails::Stack<T>::~~Stack() {
}

template <class T>
void rails::Stack<T>::pop() {
    _deque.popFront();
}

template <class T>
bool rails::Stack<T>::push (T element){
    return _deque.pushFront(element);
}

template <class T>
T& rails::Stack<T>::top() {
    return _deque.front();
}

template <class T>
bool rails::Stack<T>::empty() {
    return _deque.empty();
}

```

Imagem 8 – Construtor, destrutor, push, front, e empty

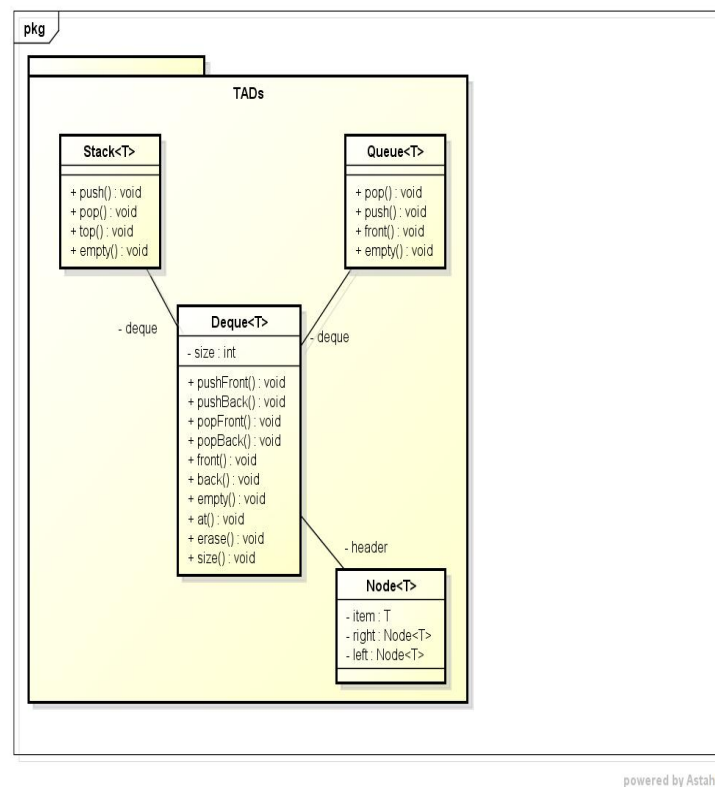


Imagem 9 – Diagrama de Classes: TADs

### Como o Mapa é utilizado?

Um Mapa é uma estrutura de dados que permite o armazenamento de elementos “indexados” por uma chave, normalmente única. Externamente, e para fins didáticos, se parece com um vetor que, ao invés de ter índices numéricos e acessar o elemento “na posição I”, permite ter qualquer tipo de objeto como índice e acessa o elemento “com a chave K”.

Internamente, entretanto, os mapas possuem comportamentos diferentes. Eles costumam ordenar seus dados em relação à chave e, para manter eficiência nas operações de inserção, consulta e remoção, são implementados com árvores binárias balanceadas.

Na implementação desse jogo, foi utilizado o `std::map`, um container da STL. Dessa forma, pode-se definir identificadores às imagens utilizadas e manipuladas pelo jogo. São utilizados dois maps. Um relaciona o identificador da imagem ao endereço (relative-path) de seu arquivo no sistema. O outro, relaciona um identificador (o mesmo, por comodidade e coerência) à um objeto da Classe `IDrawing`.

Dessa forma, reúne-se todas as imagens do sistema em uma única estrutura de dados, que assegura velocidade na manipulação desses objetos. Ex:

```
systemImages["name"].getBitmap();
```

### Arquitetura do Software:

A disposição geral do jogo baseia-se de criação de uma `RailsGUI`, que encapsula tanto as regras do jogo (`Game`) quanto a interpretação gráfica dada a ele. Para tal, criou-se o conceito de Telas (`Screens`), que possuem um sub-conjunto de imagens desenháveis.

Dessa forma, não é necessário identificar qual Tela atual está sendo exibida, pois todas possuem a mesma interface (de métodos). A Tela do Jogo (`PlayScreen`), por possuir uma lógica de funcionamento diferenciada, é uma especialização de uma `Screen`.

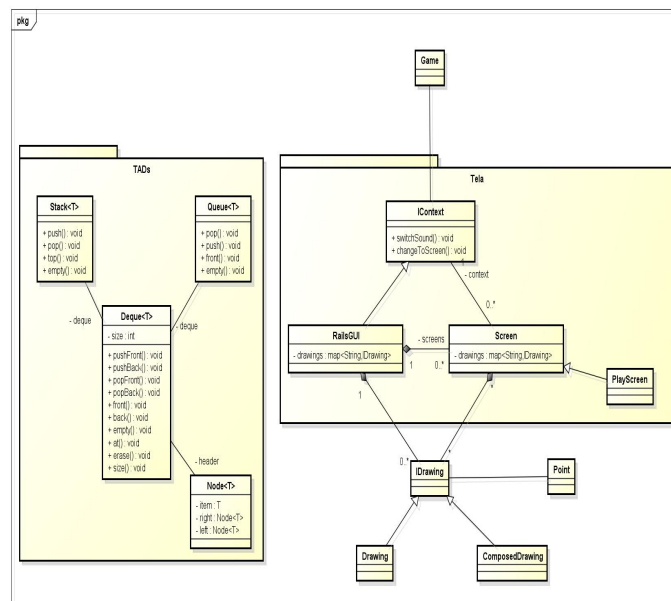


Imagem 10 – Diagrama de Classes: Visão simplificada do Jogo

### Observações e Comentários:

Essa seção reúne pequenas informações quanto ao processo de desenvolvimento do jogo.

- O Jogo em sua versão atual é dependente da dll allegro-5.0.10-monolith-mt.dll. Vários esforços foram realizados buscando eliminar essa necessidade. A ideia central é fazer um static-linkage com as bibliotecas \*-static-mt.dll disponibilizadas pelo Allegro. Entretanto, não foi encontrado nenhum caso de sucesso. Outra alternativa seria utilizar programas injetores de dlls em executáveis. Testou-se esse procedimento com o programa PE-Inject, mas sem sucesso. Exemplo de insucesso no static-linkage: <https://www.allegro.cc/forums/thread/613981>;
- Para atribuir um ícone ao executável, seguiu-se o procedimento descrito em <https://www.allegro.cc/forums/thread/614268>;
- Foi utilizado o addon PhysicsFS do Allegro, uma camada de abstração do PhysicsFS, que por sua vez, permite acesso à vários tipos de arquivos. Dessa forma, pode-se ler arquivos zipados pelos mesmos métodos de acesso de arquivos do Allegro. Veja: <https://www.allegro.cc/manual/5/physfs.html>;
- Utilizou-se a função setResourceArchive() no arquivo TheLastTooFast.cpp. Tal função está descrita em <https://www.allegro.cc/forums/thread/614268>, e faz com que o working-directory da aplicação seja mudado para a pasta em que se encontra o executável. Assim, pode-se utilizar relative-paths para referenciar os arquivos usados pelo jogo;
- A Arquitetura de Screens proposta não foi suficiente para a representação abstrata de uma Tela, tendo-se que criar artifícios internos para a correta manipulação dos elementos. De toda forma, ela pode ser revista e refinada, possivelmente, enquadrando-a em um Design Pattern.