XNA Sokoban 3D Tutorial

Índice

Introdução 3
Público Alvo 3
O Jogo 3
A versão 3D 3
Mãos à obra! 4
Arquivo de Níveis 4
Geração dos artefatos gráficos 5
Texturas 5
Modelos 3D 5
Telas 6
Implementando o Sokoban 3D com o XNA Game Studio Express.7
Criação do projeto 7
Importando artefatos no GSE 9
Game loop...10
Sokoban Game Loop 11
Como gerenciar as telas e suas transições? 11
Autores...28
Contato...28

Introdução

Este tutorial descreve como desenvolvemos uma versão do *Sokoban* em 3D utilizando o *Microsoft XNA*.

Queremos deixar claro que esse foi o nosso primeiro jogo em 3D. Nunca fizemos algo do tipo antes. Focamos mais a visualização de objetos 3D.

Público Alvo

Pessoas com conhecimento básico de *programação orientada a objetos*. Mínima noção do que venha a ser *Microsoft XNA*. Conhecimento básico de C# ou Java bastante desejável.

O Jogo

A idéia do *Sokoban* foi criada por um japonês chamado *Hiroyuki Imabayashi*, em 1980.

Ambienta-se em um estoque de fábrica com algumas caixas, e o personagem principal (*Sokoban*) deve colocar estas caixas nos lugares marcados.

Pode parecer fácil à principio, porém o Sokoban só conta como instrumento suas mãos: não há empilhadeiras no estoque e as caixas não tem puxadores e etc., assim, pode-se somente empurrar uma caixa por vez. O Sokoban não é forte o suficiente para empurrar duas ou mais caixas.

O jogo é simples assim mesmo e é relativamente fácil programá-lo. A dificuldade encontra-se em criar níveis interessantes, que requeiram boa capacidade de raciocínio por parte do jogador, equilibrando também o quesito diversão. O "trabalho sujo" de *Sokoban* fica nas mãos do *level designer*.

A versão 3D

Algumas questões gerais foram envolvidas no desenvolvimento desta versão 3D, como:

- Como criar os modelos 3D?
- Como importar estes modelos para o XNA?
- Como exibi-los na tela?
- Como gerenciar as telas e suas transições?
- Como receber a entrada do teclado / gamepad?
- Como gerenciar a colisão entre estes objetos?
- Como configurar um arquivo contendo os níveis do jogo?
- Como ler este arquivo de níveis e refletir seu conteúdo através dos objetos?

Com este tutorial responderemos estas perguntas que fizemos a nós mesmo antes de criar o XNA Sokoban 3D.

Mãos à obra!

Deixando um pouco de lado o *modo artístico, filosófico e blá-blá-blá* que um jogo é visto por alguns, vamos encarar por alguns instantes o ***XNA Sokoban 3D*** como ele é: um *software*. Isso mesmo. Tem gente que não admite que se diga tal coisa, mas no fundo o jogo eletrônico é um software escrito para rodar em um processador... Triste mas é a realidade, não é, gente? Bom, chega de choro e vamos lá...

Em linhas gerais: o *XNA Sokoban 3D* é um programa com interface configurável via arquivo XML, que por acaso é 3D. Cada nível do jogo pode ser considerado um passo de um *wizard*, que foi configurado pelo arquivo. Este XML contém o mapa dos níveis, e é possível adicionar quantos níveis quiser, sem precisar recompilar o código.

Arquivo de Níveis

A composição do mapa de um nível qualquer do Sokoban é realizada através de uma matriz, onde cada posição da matriz contém um símbolo que representa o objeto 3D que ocupará na tela, na posição correspondente.

Um arquivo XML contendo a configuração de todos os níveis do jogo tem o seguinte formato geral. Em seguida apresentamos um exemplo concreto.

```
<sokoban lines=”n” columns=”m”>
      <level characterLinePosition=”X1” characterColumnPosition=”Y1”>
            <data>(...)</data>
                  ...
            <data>(...)</data>
      </level>

      ...

      <level characterLinePosition=”Xn” characterColumnPosition=”Yn”>
            <data>(...)</data>
                  ...
            <data>(...)</data>
      </level>
</sokoban>
```

```
<sokoban lines="14" columns="20">
      <level characterLinePosition="8" characterColumnPosition="11">
            <data>____#####___________</data>
            <data>____#___#___________</data>
            <data>____#@__#___________</data>
            <data>__###__@##__________</data>
            <data>__#__@_@_#__________</data>
            <data>###_#_##_#___#######</data>
            <data>#___#_##_#####___++#</data>
            <data>#_@__@___________++#</data>
            <data>#####_###_#_##___++#</data>
            <data>____#_____##########</data>
            <data>____#######_________</data>
            <data>____________________</data>
            <data>____________________</data>
            <data>____________________</data>
      </level>
</sokoban>
```

Os símbolos utilizados são:

- _ = espaço em branco
- # = parede
- @ = caixa
- + = marcação-alvo

O programa interpreta cada símbolo e traduz para o objeto 3D correspondente dentro do jogo. A passagem de um nível para o outro só se dá quando todos os alvos são preenchidos por caixas pelo jogador.

Geração dos artefatos gráficos

Texturas

Contamos com a ajuda do aplicativo *Adobe Photoshop CS*. No nosso caso precisamos de 3 texturas: para os tijolos, para as caixas, e para as marcações dos alvos. Elas têm dimensão 50x50 pixels e estão no formato PNG.

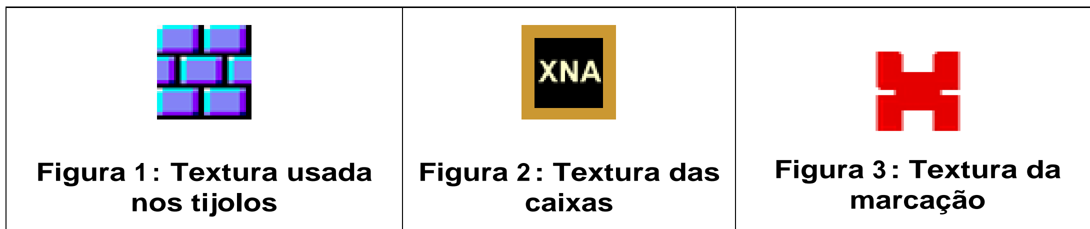

Figura 1: Textura usada nos tijolos | **Figura 2: Textura das caixas** | **Figura 3: Textura da marcação**

Modelos 3D

O jogo utiliza 3 modelos criados no *Discreet 3DS Max 7*: o módulo do piso (*dimensões 50x50*), um cubo (*dimensão 50*) e o boneco do *Sokoban* (*altura também 50*).

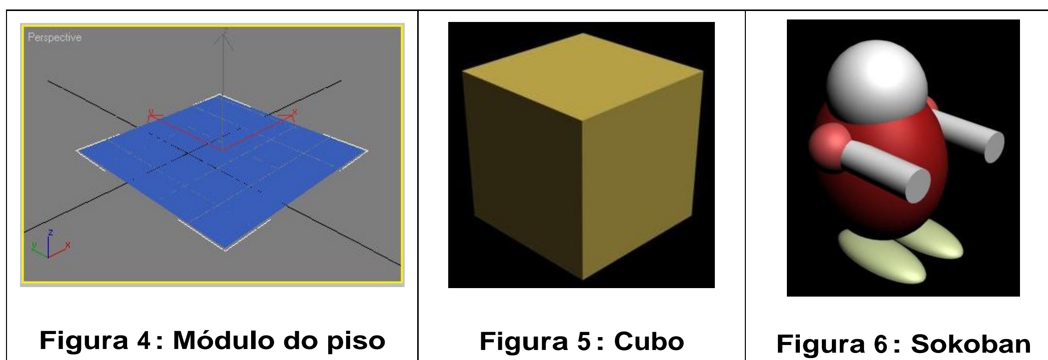

Figura 4: Módulo do piso | **Figura 5: Cubo** | **Figura 6: Sokoban**

O cubo mostrado acima é utilizado tanto para os tijolos quanto para as caixas do estoque. Para diferenciá-los, uma textura correspondente é aplicada sobre o modelo. Isso será mostrado mais adiante quando nos aprofundarmos em relação ao *XNA*.

Telas

Para **a primeira e terceira tela** utilizamos novamente o *Adobe Photoshop CS*. Já a segunda tela é o jogo em si, onde **importamos modelos e texturas** para o projeto e **gerenciamos sua exibição baseado na lógica do jogo**.

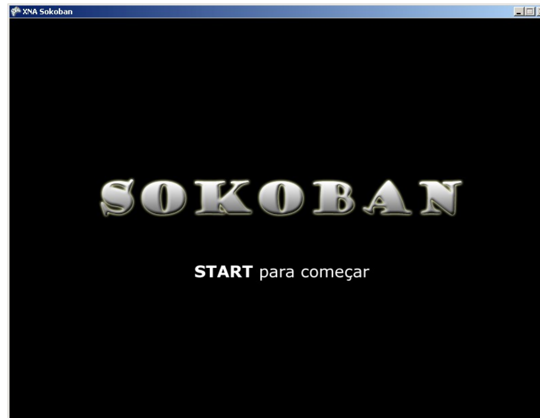

Figura 7: Tela de introdução do jogo

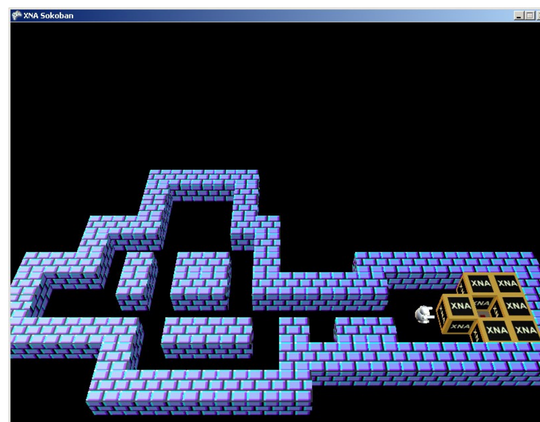

Figura 8: Sokoban 3D em ação

Figura 9: Tela de Congratulações

Implementando o Sokoban 3D com o XNA Game Studio Express

Com os artefatos criados, vamos a partir de agora nos ater a execução da parte de implementação do *Sokoban* com o ***Microsoft XNA Game Studio Express***.

Criação do projeto

Começamos criando um novo projeto do tipo "***Windows Game***" no *MS XNA GSE*. Para tal deve-se:

- Abrir o *Game Studio Express;*
- Clicar no menu *File / New Project...* ;
- Clicar na opção *Windows Game*. Isto significa que o código gerado rodará somente na plataforma *Windows*;
- No campo *Name* definir o nome do seu projeto. Aqui chamaremos de "*Sokoban3D*";
- No campo *Location* defina o caminho onde deseja hospedar seu projeto;
- Na combobox *Solution* deixe na opção "*Create New Solution*";
- Clique *OK*... Pronto! Seu novo projeto acaba de ser criado.

Se tudo correu bem até aqui, o GSE configurou seu ambiente de desenvolvimento e alguns arquivos foram criados, que servem como ponto de partida para começarmos a criação do nosso jogo. Os dois mais importantes são *Program.cs* e *Game.cs*.

A seguir explicaremos esses arquivos. Para quem sabe dotNET é básico mas vamos explicar só para dizer...

Program.cs

```
using System;

namespace Sokoban
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            using (SokobanGame game = new SokobanGame())
            {
                game.Run();
            }
        }
    }
}
```

Program é a "parte dotNET da coisa", a parte burocrática, a parte principal da sua aplicação. Qualquer jogo XNA tem como base uma **classe estática** que contém o método ***Main***, assim como qualquer aplicação dotNET. Este método Main é o primeiro a ser chamado quando seu jogo é posto em execução, e é dentro dele que criamos um objeto do tipo Game pertencente ao XNA Framework. Aí sim nossa aplicação se torna um jogo XNA! Isso que permite que criemos jogos tanto para *Windows* quanto para o *XBOX 360*.

Outra coisa que é para ser notada é o conceito de namespace, presente em várias linguagens como C++, Java e, claro, C#. Todas as classes criadas daqui por diante neste jogo estão sob o mesmo namespace, no nosso caso, `Sokoban`.

Game.cs

Vamos por partes... Nós renomeamos *Game* (*nome padrão que o XNA GSE nos forneceu*) para *SokobanGame*. Adicionamos alguns atributos e recheamos seus métodos com o que o XNA GSE nos forneceu.

```
namespace Sokoban
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class SokobanGame : Microsoft.Xna.Framework.Game
    {
        private static GraphicsDeviceManager graphics;
        private static ContentManager content;

        private static SpriteBatch batch;
        private Screen currentScreen;

        public SokobanGame()
        {
            graphics = new GraphicsDeviceManager(this);
            content = new ContentManager(Services);
        }

        //...
```

Veja que esta classe é a base do nosso jogo XNA. Ela contém o game loop, conceito que explicaremos mais adiante. Sobre ela que construiremos no jogo, definindo uma arquitetura adequada para resolver nossos problemas.

`SokobanGame` tem métodos que sobreescrevem o do seu pai, `Game`. Adiante explicaremos estes métodos e os atributos.

O que o XNA GSE nos forneceu já dá para compilar e rodar, porém não podemos considerar isso um jogo... Ele apresenta uma tela azul simples, mas como dito anteriormente, é um esqueleto base para começarmos a trabalhar.

Importando artefatos no GSE

Com o básico configurado, agora nosso projeto pode agregar os artefatos gráficos gerados pelo artista, para uso da aplicação. Para cada um deles, seguiremos os seguintes passos:

- No canto direito do GSE, há uma aba *Solution Explorer*. Se não tiver à mostra, expanda-a. Com ela você visualiza *todos os arquivos que pertencem ao seu projeto*.

- (*Opcional*) Criaremos uma pasta que conterá *somente os artefatos gráficos* (*imagens, modelos 3D, etc.*) para ficar um projeto mais "*limpo*". Para tal, clique com o botão direito sobre o nome do seu projeto, que está em negrito (*no caso **Sokoban3D***), selecione *Add / New Folder*. Dê um nome a ele, como "*Resources*", etc.

- Para adicionar o arquivo, clique com o botão direito sobre o nome do projeto em negrito exibido no *Solution Explorer* (*ou sobre a pasta recém criada, caso tenha seguido o passo anterior*). Vá em *Add / Existing Item...* e uma caixa de diálogo se abrirá para seleção do arquivo.

- *PAUSA PARA OS COMERCIAIS*: o XNA trabalha com o conceito de *Content Pipeline*. Em termos gerais, é um ente que serve para separar o trabalho do artista do trabalho do programador e facilitar sua vida. Quando você compila seu projeto, os artefatos gráficos são transformados em um formato intermediário que o XNA entende.

(continuando...)

- Nesta caixa de diálogo que apareceu, há uma combobox para seleção do tipo de arquivo. Selecione *Content Pipeline*. Escolha seu arquivo e clique OK.

- Com o arquivo selecionado, expanda a aba *Properties* existente abaixo de *Solution Explorer*. Ela exibe as propriedades do arquivo que você escolheu. Veja que algumas informações são importantes saber:

    - ***Copy to Output Directory*** em *Advanced*: como o nome diz, significa que seu arquivo será copiado para o mesmo diretório que a aplicação será compilada. Isso permite o uso de referência relativa ao seu arquivo. Coloque a opção "*Copy If Newer*"

    - ***Asset Name*** em *XNA Framework Content Pipeline*: o nome do seu artefato dentro da aplicação. Não precisa ser necessariamente o mesmo que o nome do arquivo, mas **esse é o nome que tem que ser usado quando for referenciado no seu código**.

    - ***Content Importer*** e ***Content Processor***: são os caras dentro do *Content Pipeline* que manipularão seu arquivo. Para tipos de arquivo amplamente conhecidos, o XNA provê *Importers* e *Processors default*, mas é possível escrever alguns para tipos desconhecidos. Não é do escopo deste tutorial tratar sobre este assunto, mas dê graças à Deus pelo *XNA* já vir com isso.

Pronto! Seu artefato gráfico já pode ser usado no código. Mais adiante mostraremos como o usamos de fato.

Game loop

Chegamos na parte *pra valer* da coisa. Aqui entra o conceito de *Game Loop*, que é uma seção do código que é executada desde o início até o fim e responsável pela gerencia central. Apesar de variar de plataforma para plataforma, o pseudocódigo que se segue serve para se ter uma idéia de um *Game Loop* típico:

```
enquanto (a aplicação não terminou)
      cheque a entrada do usuário;
      execute IA;
      atualize a lógica do jogo;
      desenhe os gráficos;
      toque sons;
fim-do-enquanto
```

No desenvolvimento de jogos, a escrita do game loop é considerada uma seção crítica do código, pois influencia alguns quesitos como: jogabilidade, atualização da tela (f*rames por segundo*), resposta do usuário, etc.

Felizmente no XNA isso já vem pronto de fábrica, trazendo consigo um modelo de *game loop* que será seguido por qualquer jogo. Você **não precisa se importar *quando* vai atualizar os dados nem *quando* vai atualizar a tela**, e sim *como* vai atualizá-los, permitindo o foco mais na lógica do seu jogo.

O *game loop* do XNA está contido em *Game.cs*, que descreve o objeto ***Game***. **Este objeto é o que entra em contato com o framework do XNA**. Ele contém métodos que serão chamados pelo framework automaticamente como:

- ***Initialize***: chamado assim que o jogo inicia e, como o nome diz, serve para inicializar algo não gráfico que você queira.

- ***LoadGraphicsContent***: idêntico ao Initialize, porém serve para inicializar os artigos gráficos. Ele é chamado assim que o XNA verifica que seu hardware é compatível e está pronto pra ser usado.

- ***Draw***: Chamado em intervalos de tempo regulares. Serve para atualizar o conteúdo gráfico do jogo (*posições de objetos e etc*).

- ***Update***: também chamado em intervalos de tempo regulares. Aqui deve ir seu código referente à atualização da lógica do jogo, captura de entrada do usuário, gerenciamento de colisão, etc.

- ***UnloadGraphicsContent***: momento de descarte dos objetos gráficos do jogo. Chamado antes do jogo ser encerrado.

O importante é saber que tipo de código vai em cada um destes métodos para não fazer do projeto uma zona!

Por exemplo, em um jogo do tipo *Pong*, o código de capturar as entradas do teclado e atualizar a posição das raquetes e bola iria em *Update*. Já o código de pegar o dado destas posições e atualizar os desenhos na tela iria em *Draw*.

Sokoban Game Loop

Por ser um jogo XNA o Sokoban também segue o game loop e assim criamos um objeto chamado ***SokobanGame***, derivado de Game (contido no namespace Microsoft.Xna.Framework.Game).

Nós poderíamos já começar a rechear os métodos com código pra desenhar, capturar teclas e etc., mas temos que começar a responder aquela seqüência de perguntas que fizemos anteriormente, no início do tutorial. A primeira pergunta que escolhemos para resolver foi:

Como gerenciar as telas e suas transições?

Poderíamos deixar tudo em ***SokobanGame*** mas, convenhamos, ficaria uma porcaria. Em vez de tentar bolar um esquema do zero, partimos para procurar soluções já existentes por aí. Decidimos usar o esquema feito no *XNA Spacewar Staterkit* para tal.

A princípio nós já fizemos o design das telas e as regras de interação com o usuário. Deste modo:

- **1ª tela**: (inicial) aceitará a tecla "Enter" do teclado. Ao acontecer isto, ela será trocada pela segunda tela;

- **2ª tela**: (Sokoban) aceitará as teclas direcionais (setas) do teclado e cliques do mouse. A tela somente mudará para a terceira ao ganhar o jogo.

- **3ª tela**: (congratulações) aceitará a tecla "Space" do teclado. Ao acontecer isto, ela será trocada pela primeira tela e o jogo recomeça.

A idéia geral deste esquema é agir como um **seletor de canais** de uma TV, assim, cada tela é um "canal" que é selecionado através de um "seletor". Na verdade o jogo será composto por estados que são definidos através de um **tipo enumerado do C#**: o *Enum*. As telas serão compostas como se segue.

Para compor as janelas, nos baseamos num *design pattern* chamado **Composite,** onde tratamos uma coleção de objetos como se eles fossem iguais. Isso permite maior flexibilidade para gerenciar um número grande de janelas. Apesar de termos apenas 3, não seria difícil adicionarmos mais.

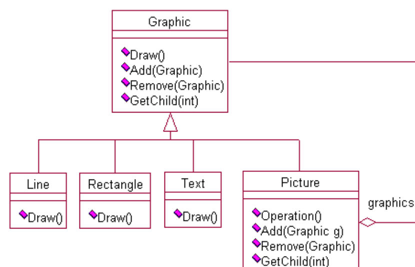

Figura 10: Composite design pattern

A imagem acima mostra a cara dos "canais" da nossa "TV". Devemos guarda uma referência no código principal para chamarmos os seus métodos quando precisarmos. Vamos introduzir código na classe SokobanGame.cs adicionando uma propriedade chamada *currentScreen*. Abaixo um esboço do que será no código:

```
public class SokobanGame : Microsoft.Xna.Framework.Game
{
	//outras propriedades

	private Screen currentScreen;

	public SokobanGame()
	{
		//inicialização
	}

	//outros métodos
}
```

Para compor o enum, vamos criar mais um arquivo para o nosso projeto. Ele na verdade conterá todos os tipos enumerados do código e estará sob o mesmo namespace dos outros, que é *Sokoban3D*:

- No canto direito do GSE, há uma aba *Solution Explorer*. Se não tiver à mostra, expanda-a.

- Para criar um novo arquivo .CS, clique com o botão direito sobre o nome do projeto em negrito exibido no *Solution Explorer*. Vá em *Add / New Item...* e uma caixa de diálogo se abrirá. Selecione "Class" e dê o nome "Enum.cs" para ele. Clique OK.

Veja que uma janela com o código gerado aparecerá. Uma classe chamada "*Enum*" foi criada dentro do namespace Sokoban3D, mas não a queremos por isso vamos deletar a parte do código que a define e inserir o código do enum que fica como assim:

```
namespace Sokoban
{
    /// <summary>
    /// Estados possíveis do Sokoban
    /// </summary>
    public enum GameState
    {
        None,

        /// <summary>
        /// Apresenta a tela inicial
        /// </summary>
        Intro,

        /// <summary>
        /// Momento onde o Jogo é exibido
        /// </summary>
        Playing,
```

```
        /// <summary>
        /// Tela de encerramento
        /// </summary>
        Congratulations
    }
    //outros enums...
}
```

Como o nome dela diz, ela diz qual estado o jogo está. Se há mudança de estado, há mudança de tela. Assim o core do jogo atua como uma *máquina de estados*. Vamos compor o código responsável pela mudança das telas.

Temos que inserir um novo método para isso. Ele receberá como parâmetro o novo estado e fará com que a referência *currentScreen* aponte pra outra tela, mas antes ela tem que chamar o método "Shutdown" da tela atual. Veja o código abaixo:

```
public void ChangeState(GameState newState)
{
    if (newState == GameState.Intro)
    {
        if (currentScreen != null)
        {
            currentScreen.Shutdown();
        }

        currentScreen = (Screen)new IntroScreen(this);
    }
    else if (newState == GameState.Playing)
    {
        if (currentScreen != null)
        {
            currentScreen.Shutdown();
        }

        currentScreen = (Screen)new SokobanScreen(this);
    }
    else if (newState == GameState.Congratulations)
    {
        if (currentScreen != null)
        {
            currentScreen.Shutdown();
        }

        currentScreen = (Screen)new CongratulationsScreen(this);
    }
}
```

Já temos o código para mudar as telas, mas... que telas? Temos que agora que criar 3 classes que herdem de *Screen* suas propriedades: *IntroScreen*, *SokobanScreen*, *CongratulationsScreen*, e começar a rechear respectivamente seus métodos *Render* e *Update* com códigos apropriados. Crie os arquivos para estas classes para prosseguirmos no tutorial.

Com os arquivos já criados, discutiremos o que vai em cada tela:

- IntroScreen
- CongratulationsScreen
- SokobanScreen

- IntroScreen

Como já dito, aqui só precisamos exibir uma imagem simples e capturar a entrada do teclado para passar para a próxima tela.

```
public class IntroScreen : Screen
{
    private Texture2D logo;
    private KeyboardState oldKeyState;
```

Criamos dois atributos para ela: *logo* e *oldKeyState*. O primeiro é uma referência para a imagem que carregamos no método LoadResources. O segundo serve como um atributo auxiliar na captura de teclas.

```
private void LoadResources()
{
    logo = SokobanGame.Content.Load<Texture2D>(@"Resources/Sokoban");
}
```

Como já foi dito, SokobanGame é a parte principal do jogo, e entra em contato com o XNA Framework. Acessamos então um atributo dele chamado Content, que é do tipo ContentManager, provido pelo XNA. Ele é responsável pela gerência dos artefatos gráficos da aplicação.

Assim, requisitamos o carregamento da imagem da tela (*neste caso, do tipo* Texture2D). Veja que passamos como parâmetro o nome da imagem ("*Sokoban*"), porém atente-se para o fato de que ela se encontra em uma pasta chamada "*Resources*" **criada por você dentro do projeto** (*isto não é um caminho real no disco*).

Com o método criado, devemos criar uma chamada para ele dentro do construtor da class, além de inicializar o atributo *oldKeyState* como se segue:

```
public IntroScreen(Game game) : base(game)
{
    this.oldKeyState = Keyboard.GetState();
    LoadResources();
}
```

Bem, estamos prontos para começar a rechear os métodos Render e Update de IntroScreen. Eles serão chamados externamente por SokobanGame.

No método Render, **tudo que temos que fazer é criar um código para exibir a imagem carregada para a memória**. Para isso precisamos do objeto SpriteBatch, também provido pelo XNA, responsável pela renderização de objetos 2D. Como nossa classe herda várias propriedades de Screen, SpriteBatch vem de "brinde" pra gente. A criação de Screen abstrata com propriedades comuns a uma tela é de grande utilidade para nós.

Então segue o código para finalmente exibir a imagem inicial na tela:

```
public override void Render()
{
    base.SpriteBatch.Begin();
    base.SpriteBatch.Draw(logo, new Vector2(75f, 50f), Color.White);
    base.SpriteBatch.End();

    base.Render();
}
```

Com a linha `base.SpriteBatch.Begin();` avisamos ao `SpriteBatch` que vamos começar a desenhar objetos 2D na tela através do comando `Draw` dele. É possível inserir vários comandos `Draw` entre `Begin` e `End` para cara imagem que queiramos exibir, **mas não antes e nem depois senão a aplicação cai!**

Perceba que o método recebe 3 parâmetros:

- `logo:` a nossa imagem já previamente carregada na memória
- `new Vector2(75f, 50f):` posição na tela
- `Color.White:` a cor que será pintada sobre a imagem. Colocamos `White` pois queremos a cor original da imagem.

Devemos agora rechear `Update` de `IntroScreen` com código para capturar os eventos do teclado, porém ele é chamado em intervalos de tempo regulares, assim como `Render`. Por isso criamos o atributo *`oldKeyState`*, que verifica o estado da tecla da última vez que `Update` foi chamado, como segue o código abaixo:

```
public override GameState Update(TimeSpan time, TimeSpan elapsedTime)
{
    KeyboardState keyState = Keyboard.GetState();

    if (keyState.IsKeyUp(Keys.Enter) && oldKeyState.IsKeyDown(Keys.Enter))
    {
        return GameState.Playing;
    }

    oldKeyState = keyState;

    return GameState.None;
}
```

Na verdade este código simula um evento do tipo `Release` no XNA.
Repare que ele retorna um estado do jogo. `SokobanGame` monitora a tela atual capturando os valores de retorno das telas para fazer a mudança.

- CongratulationsScreen

Idêntico à IntroScreen, só há a diferença da imagem 2D que deve ser carregada.

- SokobanScreen

A tela do jogo em si. Aqui carregamos os modelos 3D e o código é um pouco mais "elaborado" em relação às outras telas: mais atributos, métodos LoadResources, Render e Update mais "gordinhos" e o uso de métodos auxiliares. Além disso, utilizamos uma classe auxiliar para carregar o arquivo de configuração contendo os níveis do Sokoban.

Segue um resumo do que acontece nesta seção do programa:

- Os modelos são carregados dos arquivos de origem e postos na memória;
- As texturas são carregadas dos arquivos e postos na memória;
- O arquivo XML contendo a descrição de todos os níveis do jogo é carregado na memória;
- O programa consulta a matriz que descreve o 1° nível para plotar os objetos 3D;
- Para cada objeto a ser plotado, verifique sua posição, aplique rotação (*se necessário*) e a textura correspondente com efeitos (*se necessário*);
- Receba a entrada do teclado, trate as colisões, atualize as posições;
- Atualize a tela refletindo as novas posições...;
- Caso o jogador passe para um novo nível, atualize a matriz do nível;
- Atualize a tela refletindo as novas posições..., etc., etc., etc...;

Em *SokobanScreen*, contamos com os seguinte métodos:

- `private void LoadResources()`
- `public override void Render()`
- `public void RenderModel(Model m, Texture2D modelTexture, Vector3 mPosition, float scale, float mRotation)`
- `public override GameState Update(TimeSpan time, TimeSpan elapsedTime)`
- `private void CharacterMove(CharacterMovements vertical, CharacterMovements horizontal)`
- `private void RestartCurrentLevel()`

Estes métodos são explicados um a um mais a frente neste tutorial.

Os atributos da classe são como se segue:

```
//1a parte
private const float aspectRatio = 800.0f / 600.0f;
private const float offsetMultiplier = 120.0f;
private const float offsetX = -750.0f;
private const float offsetY = -250.0f;
private const int timeToKeyVerification = 5 * 16;
private float accumulator = 0;

//2a parte
private KeyboardState oldKeyState;
private MouseState oldMouseState;

private Vector3 camera = new Vector3(180.0f, 1100.0f, 150.0f);

//Graphics Assets
private Model modelPersonagem;
private Model plane;
private Model bloco;

private Texture2D caixaTextura;
private Texture2D tijoloTextura;
private Texture2D xTextura;

//3ª parte
private Vector3 modelPosition = Vector3.Zero;
private float modelRotation = 0.0f;

//Game logic
private Character personagem;

private int goldenBoxes = 0;
private int[,] currentLevelMap;
```

1ª parte: parâmetros para o posicionamento do mundo 3D dentro da janela

- `aspectRatio`: razão que descreve a dimensão da janela. Necessário como parâmetro da renderização 3D.
- `offsetMultiplier`: dada uma posição de um objeto referêncial, quanto o próximo objeto estará distante dele
- `offsetX` e `offsetY`: distancia do topo esquerdo do nível ao centro do mundo 3D.
- `timeToKeyVerification`: delay para verificação das teclas
- `accumulator`: acumula o total de tempo passado desde a última verificação

2ª parte**:** atributos que contém a ultima tecla digitada e o último botão usado para o clique. Além disso `camera` contém a posição inicial da câmera no mundo 3D, que não é fixa e pode ser controlado com o mouse.

Graphics Assets: atributos que guardam os modelos e suas texturas

3ª parte: atributos que guardam a posição (real) e a rotação em torno do se eixo do modelo do boneco *Sokoban*.

Game logic: `personagem` contém a posição relativa do boneco; `goldenBoxes` representa a quantidade de caixas já postas nos alvos; e `currentLevelMap` é a matriz-cópia do nível atual, muito usada para colisão dos objetos.

- SokobanScreen.LoadResources

Carregamento dos artefatos gráficos; configuração do nível;

```
//1ª parte
modelPersonagem = SokobanGame.Content.Load<Model>(@"Resources/sokobanBoneco");
plane = SokobanGame.Content.Load<Model>(@"Resources/plane");
bloco = SokobanGame.Content.Load<Model>(@"Resources/Block");

caixaTextura = SokobanGame.Content.Load<Texture2D>(@"Resources/texturaCaixa");
tijoloTextura=SokobanGame.Content.Load<Texture2D>(@"Resources/texturaTijolo");
xTextura = SokobanGame.Content.Load<Texture2D>(@"Resources/texturaX");

//2ª parte
Levels.LoadMapsFromFile();

//3a parte
currentLevelMap = Levels.getCurrentLevelMap();
personagem.position = Levels.getInitialCharacterPosition();

//4a parte
modelPosition = new Vector3(
(offsetX + (Levels.getInitialCharacterPosition()).column * offsetMultiplier),
0.0f,
(offsetY + (Levels.getInitialCharacterPosition()).line * offsetMultiplier));
```

1ª parte: carregamento dos modelos e texturas. Veja que o modo de carregamento dos modelos é idêntico ao carregamento das texturas, mudando apenas o parâmetro `Model`.

2ª parte: a classe estática `Levels` é responsável pelo carregamento do arquivo XML para a memória, transformando as descrições em matrizes. Ela contém o método `LoadMapsFromFile` que ordena que isso aconteça.

3ª parte: com o arquivo já carregado, é possível requisitar as informações necessárias como `currentLevelMap` e a posição inicial relativa do boneco no nível correspondente.

4ª parte: com a posição relativa conhecida, é necessário conhecer sua posição real no mundo 3D. O cálculo é feito sabendo-se o marco zero do nosso mundo 3D (`offsetX` e `offsetY`)

- SokobanScreen.Render

```
//desenha o tabuleiro
int imageIndex;
Vector3 position;

modelPosition = new Vector3(
                   offsetX + (personagem.position.column * offsetMultiplier),
                   0.0f,
                   offsetY + (personagem.position.line * offsetMultiplier));

//para cada celula do mapa...
for (int i = 0; i != Levels.getLines(); i++)
{
    for (int j = 0; j != Levels.getColumns(); j++)
    {
        imageIndex = currentLevelMap[i, j];
        position = new Vector3(offsetX  + (j * offsetMultiplier),
                               0.0f,
                               offsetY + (i * offsetMultiplier));

        switch (imageIndex)
        {
            case ((int)SpriteIndex.None):
                break;

            case ((int)SpriteIndex.Brick):
                RenderModel(bloco, tijoloTextura,
                            position, 0.4f, 0.0f);
                break;

            case ((int)SpriteIndex.GreenBox):
            case ((int)SpriteIndex.GoldBox):
                RenderModel(bloco, caixaTextura,
                            position, 0.4f, 0.0f);
                break;

            case ((int)SpriteIndex.X):
                RenderModel(plane, xTextura, position, 0.4f, 0.0f);
                break;
        }
    }
}

RenderModel(modelPersonagem, null, modelPosition, 0.4f, modelRotation);
```

Cada célula do mapa do nível é traduzido em uma posição real no mundo 3D. O cálculo é feito a partir do marco zero do tabuleiro + posição relativa do objeto. A matriz é revista e atualizada no método *Update*.

- SokobanScreen.RenderModel

Este código para renderização de um modelo 3D foi retirado do help do MS XNA GSE e utilizado aqui. Ele contém vários conceitos de visualização 3D envolvidos que serão explicados a seguir.

```
// Copy any parent transforms.
Matrix[] transforms = new Matrix[m.Bones.Count];
m.CopyAbsoluteBoneTransformsTo(transforms);

// Draw the model. A model can have multiple meshes, so loop.
foreach (ModelMesh mesh in m.Meshes)
{
    // This is where the mesh orientation is set, as well as our camera and
projection.
    foreach (BasicEffect basicEffect in mesh.Effects)
    {
        basicEffect.EnableDefaultLighting();
        basicEffect.World = transforms[mesh.ParentBone.Index] *
                        Matrix.CreateRotationY(mRotation) *
                        Matrix.CreateTranslation(mPosition) *
                        Matrix.CreateScale( scale );

        basicEffect.View = Matrix.CreateLookAt(camera, new
                           Vector3(camera.X, 0.0f, 0.0f), Vector3.Up);
        basicEffect.Projection = Matrix.CreatePerspectiveFieldOfView(
MathHelper.ToRadians(45.0f), aspectRatio, 1.0f, 5000.0f);

        if (modelTexture != null)
        {
            basicEffect.Texture = modelTexture;
            basicEffect.TextureEnabled = true;
        }
    }

    // Draw the mesh, using the effects set above.
    mesh.Draw();
}
```

Vamos lá... A seguir vem o básico que está por trás do código descrito acima. Se você já conhece isso, pode pular. Ele está dividido em passos numerados.

1) O objetivo final deste método é plotar um modelo 3D na tela. Pois saiba que a superfície dele é composta por triângulos que, juntos, o compõem dando o formato desejado. (Mesh)

A seguir há um exemplo de um modelo de um rosto humano. Repare de que modo ele é composto.

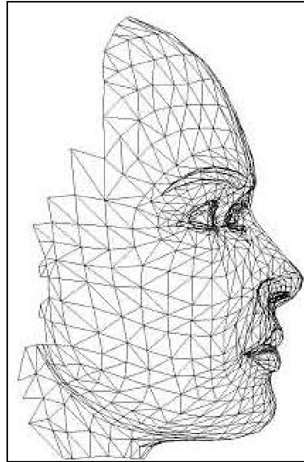

Figura 11: exemplo de modelo 3D

2) Existe um conceito chamado *Frustrum*, que é o cone de visão da câmera.

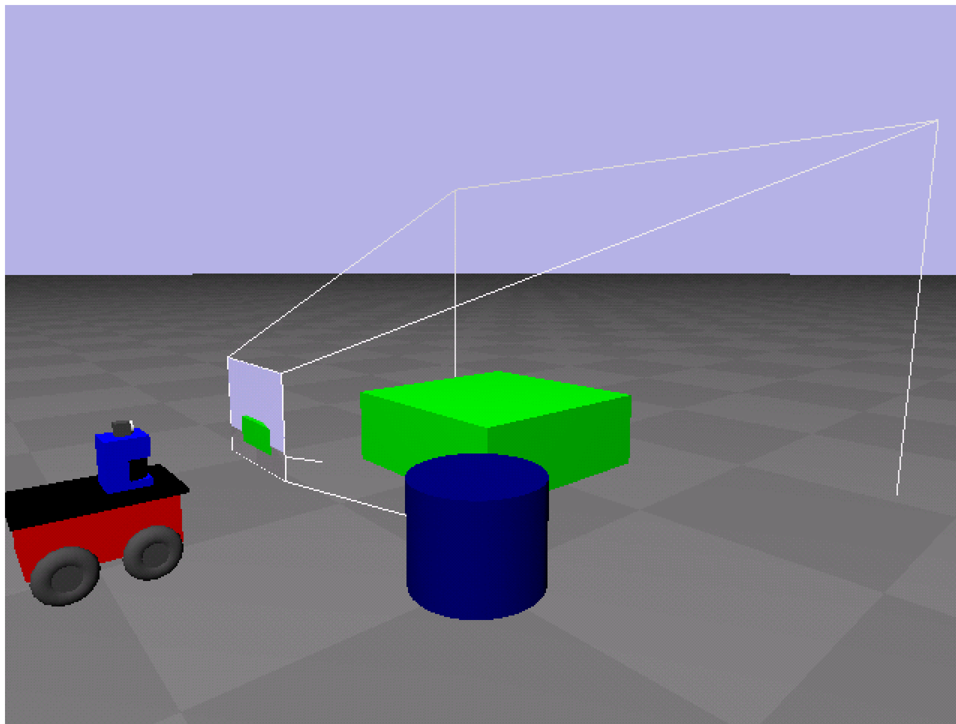

Figura 12: cone de visão da camera

Nesta imagem você está vendo os objetos em posições do seu mundo 3D. Porém há um inconveniente: seu monitor é 2D. Então aqui entra o truque: ele precisa **Projetar** (*praticamente "chapar"*) os objetos na sua tela, **mantendo a ilusão de profundidade**. E isso é feito para cada objeto que se encontra dentro do cone que você definiu. *No exemplo acima a câmera é representada pelo carrinho vermelho e a caixa verde encontra-se dentro do cone, então ele vai ser renderizado, ao contrário do cilindro azul.*

A câmera é o **representante do seu olho no mundo 3D**. Esse "*olho*" precisa:

- De uma posição no mundo 3D; (1)
- Para qual direção ele tá apontando; (2)
- Se esse olho tá de pé, deitado, etc. (3)

No XNA vc define isso com o comando

```
basicEffect.View = Matrix.CreateLookAt(camera,
                                       new Vector3(camera.X, 0.0f, 0.0f),
                                       Vector3.Up);
```

3) Cada objeto vc pode aplicar uma *Transformação* nele. Você pode:

- Mudar a posição dele (*Translation*);
- Rotacioná-lo em torno de um eixo (*RotationX, RotationY, RotationZ*);
- Mudar seu tamanho (*Scale*)

Pena que o XNA não tenha um comando do tipo *Model.setScale()* ou *Model.setRotationX*, etc. É preciso aplicar esta transformação em cada *Mesh* do seu modelo.

Para aplicar uma tranformação do tipo *Translation*, *Rotation* ou *Scale*, é utilizada uma Matrix 4x4, porém **cada transformação deve ter uma matriz própria para ela**. Abaixo segue um exemplo de uma matriz de transformação sendo aplicada em um ponto de um modelo 3D qualquer.

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} T_{00} & T_{01} & T_{02} & T_{00} \\ T_{10} & T_{11} & T_{12} & T_{13} \\ T_{20} & T_{21} & T_{22} & T_{23} \\ T_{30} & T_{31} & T_{32} & T_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Figura 13: matriz de transformação

Já foi dito *cada transformação deve ter uma matriz própria para ela,* porém elas podem ser combinadas e o XNA permite isso. Veja no exemplo uma linha que faz isso:

```
basicEffect.World = transforms[mesh.ParentBone.Index] *
                    Matrix.CreateRotationY(mRotation) *
                    Matrix.CreateTranslation(mPosition) *
                    Matrix.CreateScale( scale );
```

Ele combina as matrizes através da multiplicação.

5) Para definir um *Frustrum,* é necessário definir os seguintes parâmetros:

- Angulo de abertura do cone
- ***Near plane***: a "*base pequena*" do cone, ou seja, a partir de que distância, partindo da câmera, os objetos vão ser considerados para renderização
- ***Far plane***: a "*base grande*" do cone, ou seja, até onde, a partir do *near plane*, os objetos serão considerados para a renderização.

No XNA isso é definido com a linha:

```
basicEffect.Projection = Matrix.CreatePerspectiveFieldOfView(
MathHelper.ToRadians(45.0f), aspectRatio, 1.0f, 5000.0f);
```

6) Tudo que foi dito é aplicado para cada *Mesh* de todos os modelos 3D que se encontram dentro do *Frustrum*.

É possível aplicar efeitos extras sobre os *Meshes* dos modelos usando arquivos de configuração escritos em HLSL (*High Level Shader Language*), porém estamos não estamos usando nada disso, somente coisas básicas. Por isso o *BasicEffect*.

- SokobanScreen.Update

Hora da lógica do jogo! Já cobrimos a parte de renderização do jogo. Lá vimos que consultamos algumas estruturas de dados para atualizar a tela. Neste método manipulamos estas estruturas através da entrada do usuário.

Por partes...

1) Movimento de câmera.

```
//Move a camera
MouseState mouseState = Mouse.GetState();

if (mouseState.LeftButton == ButtonState.Pressed)
{
    camera.X += mouseState.X - oldMouseState.X;
    camera.Z += mouseState.Y - oldMouseState.Y;
}

camera.Y += mouseState.ScrollWheelValue -
oldMouseState.ScrollWheelValue;
```

Captura-se o estado do mouse. Comparamos com o estado anterior para vermos quanto ele se moveu. Com esse dado, atualizamos a posição da câmera no mundo 3D.

2) Carregar próximo nível

```
//É hora de ir ao próximo nível?
if (goldenBoxes >= Levels.getTotalOfAimSpaces())
{
    bool state = Levels.increaseCurrentLevel();

    //O jogo ainda não acabou?
    if (state)
    {
        RestartCurrentLevel();
    }
    else
    {
        return GameState.Congratulations;
    }
}
```

O número de caixas que já foram postas nos lugares marcados é dado por `goldenBoxes`. Como é sabido, cada nível tem seu número de caixas próprio. Caso o jogador tenha alcançado o objetivo do nível, o próximo é carregado. Se todos os níveis foram concluídos, o jogo acaba.

3) Movimento do boneco

```
//Capta as teclas para o movimento do boneco
KeyboardState keyboardState = Keyboard.GetState();

accumulator += elapsedTime.Milliseconds;

if (accumulator == timeToKeyVerification)
{
    if (keyboardState.IsKeyDown(Keys.Left))
    {
        CharacterMove(CharacterMovements.None, CharacterMovements.Left);
        modelRotation = MathHelper.ToRadians(270.0f);
    }
    else if (keyboardState.IsKeyDown(Keys.Right))
    {
        CharacterMove(CharacterMovements.None, CharacterMovements.Right);
        modelRotation = MathHelper.ToRadians(90.0f);
    }
    else if (keyboardState.IsKeyDown(Keys.Up))
    {
        CharacterMove(CharacterMovements.Up, CharacterMovements.None);
        modelRotation = MathHelper.ToRadians(180.0f);
    }
    else if (keyboardState.IsKeyDown(Keys.Down))
    {
        CharacterMove(CharacterMovements.Down, CharacterMovements.None);
        modelRotation = MathHelper.ToRadians(0.0f);
    }
    else if (keyboardState.IsKeyDown(Keys.R))
    {
        RestartCurrentLevel();
    }

    accumulator = 0;
}
```

O método `Update` é chamado automaticamente pelo XNA, através de `SokobanGame` em um intervalo de tempo muito curto. Assim, mesmo que o usuário tecle algum botão muito rápido, para o `Update` ele ainda está com ele pressionado, agindo conforme o caso.

A variável `accumulator` atua como um *flag* para o *delay* desejado, permitindo que o tempo de resposta fique mais plausível. Quando isso ocorre, verificamos as teclas pressionadas e, conforme for, movemos o boneco para a posição indicada e o giramos para a direção correta fazendo uso do método auxiliar `CharacterMove`.

- SokobanScreen.CharacterMove

Move o boneco pelo ambiente, verificando a colisão dele com outros objetos. O boneco na verdade é representado na matriz por sua posição dentro dela. A "colisão" é verificada pelas células adjacentes à célula dele. Vamos por partes...

O método recebe como parâmetro *o quanto ele deve se mover* na direção vertical ou na direção horizontal. Isso é indicado por `offsetLine` e `offsetColumn`.

```
int offsetLine = (int) vertical;
int offsetColumn = (int) horizontal;
//Posicao à frente do jogador
int nearPosition = currentLevelMap[personagem.position.line + offsetLine,
                                   personagem.position.column + offsetColumn];
```

No código acima, representa a posição adjacente ao boneco no mundo 3D do nível, ou seja, `nearPosition` representa a posição que tem *o maior potencial de haver uma colisão*.

```
if (nearPosition != (int)SpriteIndex.Brick)
```

A linha acima evita cálculos desnecessários de colisão e movimento. Se o boneco está cara a cara com uma parede, não nada que possa ser feito. A seguir o bloco que é executado, caso a condição seja verdadeira... mas apresentaremos por partes.

```
//Duas posicões à frente do jogador
int farPosition = currentLevelMap[personagem.position.line + 2 *
offsetLine, personagem.position.column + 2 * offsetColumn];

bool greenBoxChecking = (nearPosition == (int)SpriteIndex.GreenBox);
bool goldBoxChecking = (nearPosition == (int)SpriteIndex.GoldBox);
```

Se `nearPosition` representa a posição adjacente ao boneco, `farPosition` representa o que há após esta posição. Isso é necessário para saber se é possível o boneco mover uma caixa para algum lugar, por exemplo. `greenBoxChecking` e `goldBoxChecking` indica se há alguma caixa à frente do boneco.

```
//se estou cara-a-cara com uma caixa... (verde || dourada)
if (greenBoxChecking || goldBoxChecking) //CONDICAO #1
```

O bloco de código que é executado se a condição acima é verdadeira é mostrado a seguir (em partes):

```
bool emptySpaceChecking = ((farPosition == (int)SpriteIndex.None));
bool aimSpaceChecking = ((farPosition == (int)SpriteIndex.X));
```

O a linhas acima querem colher a informação do ambiente: "é possível mover a caixa"? A condição a seguir verifica isso:

```
if (emptySpaceChecking || aimSpaceChecking) //CONDICAO #2
```

As linhas abaixo atualizam na matriz a posição do boneco e das caixas, ou seja, aqui *o boneco moveu um caixa para algum lugar*. Além disso, a contagem das caixas nos alvos também é verificada. O método `Update` vê as mudanças na matriz para atualizar a tela.

```
//Mantenha-a verde ou transforme-a em dourada (pois atingiu um alvo)
currentLevelMap[ personagem.position.line + 2 * offsetLine,
personagem.position.column + 2 * offsetColumn ] =
(int)((emptySpaceChecking) ? (SpriteIndex.GreenBox) :
(SpriteIndex.GoldBox));

currentLevelMap[ personagem.position.line + offsetLine,
personagem.position.column + offsetColumn] = (int)((goldBoxChecking) ?
(SpriteIndex.X) : (SpriteIndex.None));

personagem.position.line += offsetLine;
personagem.position.column += offsetColumn;

if (greenBoxChecking && aimSpaceChecking)
{
    goldenBoxes++;
}
else if (goldBoxChecking && emptySpaceChecking)
{
    goldenBoxes--;
}
```

O "senão" da **CONDICAO #1**, que servia para verificar se havia algo em frente ao boneco é resolvido aqui, ou seja, *uma simples movimentação do boneco livre pelo cenário*:

```
//se estou cara-a-cara com nada, mova
else
{
    personagem.position.line += offsetLine;
    personagem.position.column += offsetColumn;
}
```

- SokobanScreen.RestartCurrentLevel

Este método auxiliar é chamado por de caso o usuário queira recomeçar um nível: a quantidade de caixas já alcançadas é zerada, o personagem e as caixas voltam a posição inicial do nível correspondente e, claro, a matriz que descreve o nível volta ao seu início.

Isso acontece quando o usuário tecla "R" em um nível qualquer. Necessário caso o jogador erre ou fique preso por conta própria no estoque. Cabe dizer aqui que é uma facilidade oferecida por nós nesta versão do jogo.

```
goldenBoxes = 0;
personagem.position = Levels.getInitialCharacterPosition();
modelPosition = new
Vector3(((Levels.getInitialCharacterPosition()).column *
offsetMultiplier + offsetX),
                              0.0f,

((Levels.getInitialCharacterPosition()).line * offsetMultiplier +
offsetY));
currentLevelMap = Levels.getCurrentLevelMap();
```

Autores

O desenvolvimento deste *Sokoban 3D* foi parte integrante do processo de aprendizado do *Microsoft XNA* de Bruno Rabello e Edson Mattos em 2007. Eles são estudantes de Ciência da Computação da Universidade Federal Fluminense, *Niterói – RJ*, e escolheram o *Microsoft XNA* como base para o desenvolvimento do Trabalho de Conclusão de Curso, orientado pelo prof. Esteban Gonzalez Clua.

Contato

Bruno Rabello: uffbruno@gmail.com

Edson Mattos: edsonmattos@gmail.com