Introdução ao XNA

Bruno Rabello[1]

Edson Mattos[1]

Bruno Evangelista[2]

Esteban Clua[1]

Resumo: Este tutorial cobre as características básicas sobre a plataforma de desenvolvimento de games criada pela Microsoft chamado XNA (XNA's Not Acronymed). Ela permite criar jogos para PCs com Windows e para o console Xbox 360. O XNA vem substituir o Managed DirectX, uma versão do DirectX para a plataforma dotNET. Todas as aplicações XNA são compiladas para código gerenciado (managed code, ou Microsoft Intermediate Language). Esse código é executado no Common Language Runtime (CLR), que é a máquina virtual da plataforma dotNET.

Abstract: This tutorial explore the basic characteristics for the game development plataform developed by Microsoft, called XNA (XNA´s Not Acronymed). XNA allows the creation of PC games, for Windows plataform and XBOX 360, for a console plataform. XNA aims to substitute the DirectX Manager, a version of DirectX for a .NET plataform. All the applications made in XNA are compiled in a managed code. This code is executed at the Common Language Runtime (CLR), which is the virtual machine of the .NET plataform.

1 Instituto de Computação, Universidade Federal Fluminense
2 Instituto de Informática, Universidade Federal de Minas Gerais

1 Introdução

Em 2006 a Microsoft lançou uma plataforma inédita para desenvolvimento de games e aplicações de computação gráfica tempo real, denominado XNA - XNA's Not Acronymed. Esta é a primeira API unificada para desenvolvimento em PCs e Consoles de Games.

Muitas empresas estão adotando esta plataforma para propósitos comerciais e muitos game engines tradicionais estão sendo portados para este ambiente. O sucesso da ferramenta pode ser comprovado pelo grande volume de fóruns, sites e comunidades que surgem espontaneamente pelo mundo. Além da sua grande qualidade técnica e a simplicidade da arquitetura proposta, o fato de possuir versões de distribuição gratuita garante que a enorme massa de desenvolvedores independentes voltem sua atenção para o XNA.

A confirmada capacidade de produzir aplicações sofisticadas em pouco tempo e com poucos recursos, torna o XNA também uma excelente plataforma para desenvolvimento de pesquisas, não apenas em entretenimento digital mas em outras áreas de interesse acadêmico, tais como inteligência Artificial, computação gráfica, redes e interface homem-máquina.

Em termos acadêmicos, acrescenta-se também ao seu potencial o fato de poder ser utilizado para ensino de diversas disciplinas de computação, o que pode ser comprovado ao observar diversas universidades prestigiosas ao redor de todo o mundo adotando esta estratégia.

O XNA vem substituir o Managed DirectX, uma versão do DirectX para a plataforma dotNET. Todas as aplicações XNA são compiladas para código gerenciado (managed code, ou Microsoft Intermediate Language). Esse código é executado no Common Language Runtime (CLR), que é a máquina virtual da plataforma dotNET.

2 Componentes do Microsoft XNA

O ambiente de desenvolvimento que o XNA provê envolve são os seguintes:

• Framework: XNA Framework, com as classes necessárias para se criar um jogo, que pode ser tanto 2D como 3D;

• Gerência de artefatos: XNA Content Pipeline, encarrega-se de gerenciar a conexão entre os recursos produzidos pelos artistas (modelos, texturas, sprites, etc.) e o código do programador;

• Integrated Development Environment (IDE): XNA Game Studio Express, baseado no Visual C# 2005 Express. (gratuito, versão para estudantes, hobbistas e desenvolvedores independentes) ou o XNA Game Studio Professional (para desenvolvedores profissionais);

• Ferramentas de específicas para áudio: XNA XACT e o XACT Audio Authoring Tool.

2.1 XNA Framework

Conjunto de classes especificadas e criadas especificamente para o desenvolvimento de games. Ele é dividido em sete namespaces, que são descritos a seguir:

• Microsoft.Xna.Framework.Audio: classes, estruturas e enumerações relacionadas à API do XACT, capaz de manipular arquivos gerados por ele.

• Microsoft.Xna.Framework.Content: são as classes para a lidar com o Content Pipeline em tempo de execução. Permite, dentre outras coisas, que se faça uso em tempo de execução de artefatos como imagens, modelos 3D, arquivos XML, etc. Os recursos a serem usados devem ter sido previamente importados ao projeto.

• Microsoft.Xna.Framework.Design: classes que permitem a conversão entre tipos de valores.

• Microsoft.Xna.Framework.Graphics: contém classes que provomem um acesso ao hardware gráfico para uso no jogo, permitindo uma abstração à utilização da API DirectX.

• Microsoft.Xna.Framework.Input: classes, estruturas e enumerações referentes à manipulação para entrada de dados como mouse, teclado e gamepad.

• Microsoft.Xna.Framework.Storage: classes que permitem a leitura e escrita de arquivos e a abstração de dispositivos de armazenamento.

• Microsoft.Xna.Framework: contém classes, interfaces, estruturas e enumerações consideradas básicas, como Game, GameComponent, BoundingFrustrum, GameTime, GraphicsDeviceManager, etc.

2.2 XNA Content Pipeline

Permite a conversão dos artefatos para managed code, que pode ser lido pela aplicação em tempo de execução. Para essa transformação acontecer, o arquivo passa por uma série de estágios até chegar ao produto final, que são as seguintes: Importer, Processor, Compiler e Loader.

• Importer: transforma o arquivo do formato original para um formato intermediário, que utiliza tags em XML.

• Processor: a partir do arquivo gerado pelo Importer, o Processor traduz e gera como saída código-fonte em C# com a descrição do artefato.

• Compiler: com o código-fonte do artefato gerado pelo Processor, o Compiler faz a compilação, gerando código no formato MSIL, que está pronto para ser usado em tempo de execução pela aplicação.

• Loader: carrega o artefato compilado quando requisitado.

O Content Pipeline é flexível o bastante para permitir que se escrevam Importers e Processors para qualquer tipo específico de arquivo não suportado pelo XNA. A sua API é bem documentada e encontra-se no help do XNA.

2.2 XNA Game Studio Express

IDE baseada no Visual C# 2005 Express. Mantém algumas facilidades do Visual C# 2005 como: syntax highlighting, tecnologia de code complete chamada IntelliSense, just-in time debugging, criação de starter kits (templates de projeto), interface agradável, intuitiva e fácil de usar.

O XNA Game Studio Express (XNA GSE) permite que se criem projetos de games tanto para Windows como para Xbox 360, com o mesmo código fonte, sem grandes alterações. Caso a escolha seja gerar código para Xbox 360, deve-se evitar a utilização de bibliotecas (DLL) com código não gerenciado (unmanaged code). Este console utiliza um sistema operacional específico para ele e um processador próprio, baseado no IBM Power PC com três cores. Assim, uma DLL compilada com código nativo não será entendida no ambiente do Xbox 360, causando erro.

3 Criando um Game simples: Sokoban 3D

Para introduzir ao uso do XNA, será criado um pequeno exemplo de uma versão 3D do jogo Sokoban. Com ele serão mostrados alguns passos na utilização da IDE, no uso de algumas classes do Framework e do Content Pipeline. Os passos básicos serão descritos de maneira genérica, para que possa ser utilizado em qualquer projeto.

3.1 Criação de um projeto no XNA Game Studio Express

A área de impressão é de 13cmX17cm sendo o tamanho da página definido como 16cm × 23cm. O texto deve ocupar a linha inteira e as figuras não podem ultrapassar as margens definidas. Procure preencher toda a área de impressão da página de modo não deixar mais do que 2 cm em branco no final de cada página. Faça seu maior esforço, movimentando figuras e texto, para completar cada página.

Ao criar um novo projeto, podem-se usar diversos templates, o que facilitarão no ajuste do ambiente para uma aplicação específica. Neste primeiro exemplo, inicia-se com o template do tipo “Windows Game” no MS XNA GSE. Para tal deve-se:

• Abrir o Game Studio Express;

• Clicar no menu File / New Project...

• Uma caixa de dialogo aparecerá

• Escolher a opção Windows Game no campo Templates;

• No campo Name, dar um nome para o projeto. No caso deste tutorial foi colocado "Sokoban3D";

• Ao clicar em OK o projeto acaba de ser criado no local indicado no campo Location.

Nesta etapa, o XNA GSE estará configurado para iniciar o desenvolvimento da aplicação. Tanto o ambiente de desenvolvimento como uma série de arquivos foram criados. Estes servirão como ponto de partida para a criação do jogo. Os dois arquivos mais importantes são o Program.cs e o Game.cs.

3.2 Analisando as classes recém-criadas: Program e Game1

```
namespace Sokoban3D
{
    static class Program
    {
        static void Main(string[] args)
        {
            using (Game1 game = new Game1())
            {
                game.Run();
            }
        }
    }
}
```

A listagem acima mostra a classe Program, que representa o ponto de entrada da aplicação. Por se tratar de uma aplicação .net, o game precisa de uma classe que contenha o método Main, que será a primeira a ser chamada pelo sistema quando for carregado.

Nesta classe foi instanciado um objeto do tipo Game. Pode-se renomear a classe e seu arquivo para atender as nossas necessidades da aplicação. Mais adiante será mostrado como fazer isso usando os recursos do XNA GSE. A classe Game irá conter o código principal com a lógica e renderização da aplicação. A chamada ao método Run consiste na execução desta classe, implicando na execução do loop de um game, que estará desmembrado na implementação da classe Game.

Game1 é uma classe derivada de Game, pertencente ao XNA Framework. Em qualquer jogo implementado no XNA deve-se ter pelo menos uma classe derivada de Game no, pertencente ao Microsoft.Xna.Framework. Esta classe contém métodos que são chamados automaticamente pelo sistema, compondo o game loop. À partir da implementação

destes métodos da classe Game será implementado o jogo, sobrescrevendo seus métodos para executar código desejado.

Deve-se destacar também para o conceito de namespace, presente em várias linguagens como C++, Java, etc. Namespace é o particionamento lógico das classes de um sistema. Todas as classes criadas daqui por diante num jogo estão sob o mesmo namespace. No caso deste exemplo inicial, Sokoban3D. O nome do namespace pode ser modificado e podem-se criar quantos namespaces se desejar.

O que o XNA GSE criou até este primeiro momento já pode ser compilado e executado, porém obviamente não pode ser considerado ainda um jogo... Ao executar-se a aplicação, tem-se como resultado uma tela azul simples, como mostrado na figura 1. Como dito anteriormente, já se pode considerar a aplicação como um esqueleto base para o desenvolvimento de uma aplicação maior.

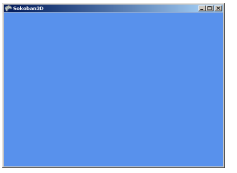

Figura 1: Game executando com o projeto recém criado.

3.3 Renomeando classes usando o recurso de Refactoring

O XNA Game Studio Express definiu automaticamente o nome das classes principais: Program e Game1. Recomenda-se neste momento renomear Game1 para SokobanGame:

• Clicar com o botão direito do mouse sobre o nome da classe. Um menu pop-up aparecerá, como na figura 2. Escolher Refactor / Rename. Uma nova caixa de diálogo aparecerá, como mostra a figura 3.

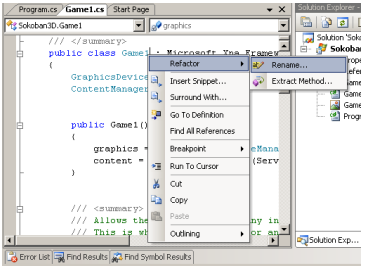

Figura 2: Usando o recurso de refactoring

• Nesta nova caixa de diálogo, deve-se colocar o novo nome da classe desejada. Neste caso será colocado o nome SokobanGame. Após fazer isto, clicar em OK.

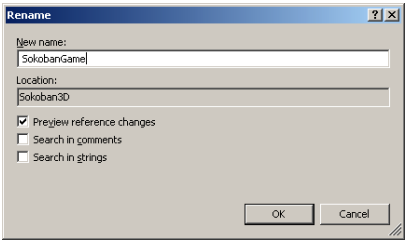

Figura 3: Escolhendo o novo nome da classe.

• Antes de efetivar o renomeio da classe, uma nova janela é exibida mostrando os lugares que a classe é referenciada ou usada, como ilustra a figura 6. Assim, é possível aplicar-se a modificação somente em partes do projeto. Neste caso deseja-se que todas as referências, tanto em Program como em Game1, as modificações sejam efetuadas.

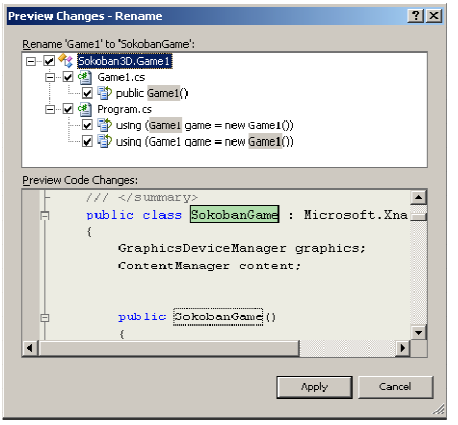

Figura 4: Confirmando o renomeio da classe.

3.4 Entendendo a classe Game

A seguir, na listagem abaixo, apresenta-se detalhes da classe SokobanGame:

```
namespace Sokoban3D
{
    public class SokobanGame : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        ContentManager content;

        public Game1()
        {
```

```
            graphics = new GraphicsDeviceManager(this);
            content = new ContentManager(Services);
        }

        protected override void Initialize()
        protected override void LoadGraphicsContent
                                    (bool loadAllContent)
        protected override void Update(GameTime gameTime)
        protected override void Draw(GameTime gameTime)
        protected override void UnloadGraphicsContent
                                    (bool unloadAllContent)
    }
}
```

Como já foi dito, SokobanGame sobrescreve alguns métodos presentes na classe Game. O XNA GSE criou stubs destes métodos automaticamente para poderem ser completados com o código apropriado para a aplicação específica. A seguir será explicado sobre o porquê desta classe ser a base de um game XNA.

3.4.1 Abstração do hardware gráfico

O atributo graphics presente em SokobanGame permite ter-se acesso ao hardware gráfico. É possível que se tenha mais de uma GPU presente no sistema, e a GraphicsDeviceManager, pertencente ao namespace Microsoft.Xna.Framework provê meios de se obter acesso a elas. Neste caso se está considerando apenas uma placa gráfica, portanto o atributo GraphicsDevice, pertencente à graphics, representa esta única GPU.

Através do atributo graphics se poderá obter informações relavantes para a aplicação, tais como o tamanho da janela, se a janela ela se encontra em full screen, qual suporte da GPU em relação à shading, etc.

3.4.2 Acesso aos artefatos

A manipulação das imagens, modelos e efeitos (Texture2D, Model e Effect, respectivamente) se dá utilizando-se o atributo content, presente em SokobanGame.

Pode-se carregar este tipo de conteúdo através do seu método Load. Por exemplo, para carregar um modelo 3D, deve-se seguir o procedimento abaixo:

Model myModel = this.content.Load<Model>(@"nome_do_modelo");

3.4.3 Inicializando

O método Initialize será chamado automaticamente pelo sistema antes do jogo começar efetivamente. Ali devem ser feitas verificações, carregar conteúdo não gráfico (como arquivos XML), iniciar componentes e todas as tarefas preliminares ao laço principal do game.

3.4.4 Carregando conteúdo

A listagem da sessão 3.4.2 mostrou como se carrega um modelo 3D, porém ela deve estar implementada dentro de algum método da classe SokobanGame. O melhor lugar para isso será em LoadGraphicsContent, pois quando esse método for chamado pelo sistema, a GPU já foi analisada e está pronta para ser utilizada pela aplicação.

3.4.5 Atualizando a lógica do jogo

O método Update é chamado automaticamente pelo sistema de tempos em tempos, requisitando que se atualize a lógica do jogo.

3.4.6 Atualizando os gráficos na tela

Assim como o método Update, Draw também é chamado com certa regularidade pelo sistema. Essa regularidade pode ser definida no desenvolvimento da classe, sendo de tempo fixo ou variável.

3.4.7 Finalização

Antes de o jogo encerrar, pode ser necessário descartar o conteúdo carregado. Desta forma, UnloadGraphicsContent permite que se faça esse tipo de tarefa.

Perceba que conhecendo o tipo de código que vai dentro de cada método, só nos é preciso "rechear" SokobanGame, pois a janela já foi criada, a ponte para a GPU feita, o modo carregamento de conteúdo facilitado e o game loop em funcionamento.

3.5 Importando artefatos no GSE

Com o básico configurado, agora o projeto pode agregar os artefatos gráficos produzidos pelo artista, para uso da aplicação.

3.5.1 Criando uma nova pasta

Recomenda-se criar uma pasta no projeto para conter o material a ser usado e produzido:

No Solution Explorer, clicar com o botão direito do mouse sobre o nome do projeto. Um menu pop-up aparecerá, como mostrado na figura 5. Escolher a opção Add / New Folder.

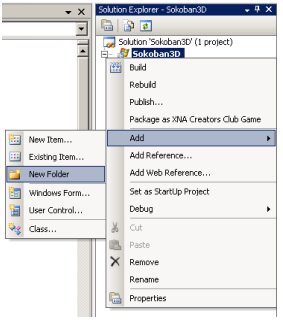

Figura 5: Criando uma nova pasta.

A nova pasta foi criada. Deve-se dar um nome à ela. Para tanto, Escolher Content, como mostrado na figura 6.

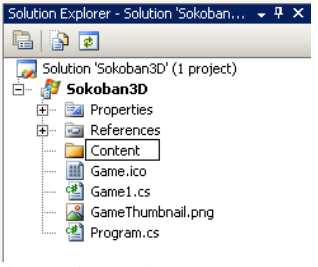

Figura 6: Dando nome à pasta.

3.5.2 Adicionando elementos para a pasta no projeto

Um game é composto por diversos elementos, além do seu código: texturas, sprites, modelos 3D, etc. Antes de usar estes elementos na programação da lógica do jogo, deve-se inseri-los numa pasta que contará estes dados. Para realizar esta tarefa, deve-se seguir o procedimento abaixo:

- No Solution Explorer, clicar com o botão direito do mouse sobre a pasta Content, conforme ilustrado na figura 7. Escolher Add / Existing Item...

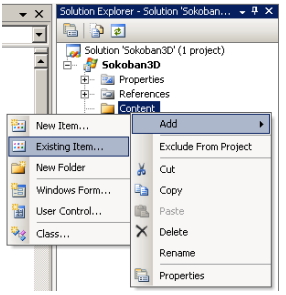

Figura 7: Adicionando novos itens a pasta.

- Na caixa de diálogo que apareceu, escolher a opção Content Pipeline no campo Files of type. Selecionar o arquivo desejado e clicar no botão Add. O arquivo será adicionado ao projeto (O exemplo apresenta o arquivo GameThumbnail.png, mas ele não será usado efetivamente).

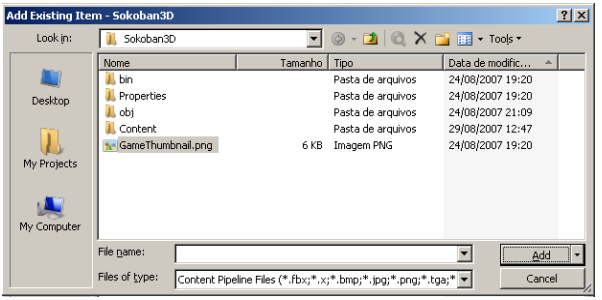

Figura 18: Adicionando a pasta de projeto um Item.

- Feito isto, o arquivo está pronto para ser usado via código, como na figura 9.

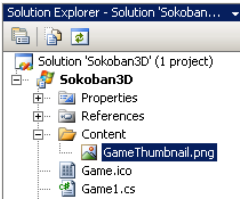

Figura 9: GameThumbnail.png no Solution Explorer.

3.5.3 Artefatos gráficos e o Content Pipeline

Como dito anteriormente, o XNA trabalha com um elemento chamado Content Pipeline. Após ter-se importado um arquivo para o projeto, o Content Pipeline será fundamental para a manipulação destes elementos, pois criará o elo de ligação com os arquivos e o código que os referencia.

Pode-se conhecer algumas propriedades relacionadas ao arquivo que foi importado da seguinte forma:

- No arquivo desejado, clicar com o botão direito do mouse. Um menu pop-up aparecerá, como na figura 10. Escolher Properties e as propriedades relacionadas ao arquivo aparecerão.

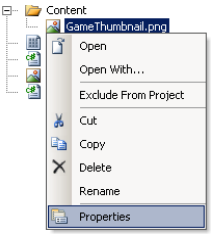

Figura 10: Exibindo as propriedades do arquivo.

- A figura 11 exibe a aba *Properties* do arquivo escolhido

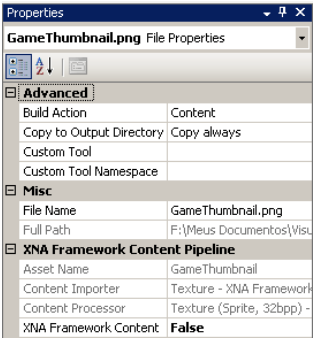

Figura 11: Aba Properties.

- Alguns campos apresentam informações relevantes em relação ao arquivo, observe-se algumas delas:
    - Advanced → Copy to Output Directory: como o nome diz, significa que o arquivo será copiado para o mesmo diretório em que a aplicação será

compilada. Isso permite o uso de referências relativas ao arquivo. Colocar a opção "Copy If Newer"

- XNA Framework Content Pipeline → Asset Name: o nome que servirá para referenciar ao artefato desde dentro da aplicação. Não precisa ser necessariamente o mesmo nome do arquivo, mas este é o nome que deve que ser usado quando for referenciado no código e não mais o nome do arquivo original.
- Content Importer e Content Processor: são os elementos dentro do Content Pipeline que manipularão o arquivo no momento do build. Para tipos de arquivos amplamente conhecidos, o XNA provê Importers e Processors default, mas é possível escrever para tipos desconhecidos. Não é do escopo deste tutorial tratar sobre este assunto, mas o XNA já possui diversas implementações prontas.

4 Usando Imagens, Modelos e Efeitos

Como foi apresentado na seção Importando artefatos no GSE, viu-se que é relativamente simples o processo de importar estes elementos. Agora será discutido como exibi-los numa aplicação. As duas seções a seguir mostram como carregar e exibir uma imagem 2D e um modelo 3D. Será visto que o modo de carregamento entre os dois é semelhante, porém o modo de desenhar é distintos para ambos.

4.1 Exibindo imagens 2D

Para exemplificar o processo de lidar com uma figura será usando um arquivo de imagem como mostrado na figura 12.

Figura 12: texturaCaixa.png

Considere-se que este arquivo já foi adicionada ao projeto e o nome que lhe foi atribuído é texturaCaixa, dentro de uma pasta chamada Content. Este processo adicionou um atributo ao SokobanGame,mas precisa ser criado um objeto que guardará uma referência para a imagem carregada, como pode ser visto na listagem abaixo

```
public class SokobanGame : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    ContentManager content;
```

```
        //Novos atributos
        Texture2D imagem;
      SpriteBatch batch;
```

Além disso, colocou-se mais um atributo chamado batch, que serve para desenhar imagens 2D na tela.

Para carregar a imagem e inicializar o modo de desenho na tela, será utilizado o método LoadGraphicsContent, da seguinte forma:

```
protected override void LoadGraphicsContent(bool loadAllContent)
{
    if (loadAllContent)
    {
        batch = new SpriteBatch(graphics.GraphicsDevice);
        imagem                                                    =
content.Load<Texture2D>(@"Content\texturaCaixa");
    }
```

Foi criado um objeto do tipo SpriteBatch, pertencente ao namespace Microsoft.Xna.Framework.Graphics. Sua contrução requer que seja passado como parâmetro qual GPU ele deve usar para desenhar a imagem. No exemplo em questão há apenas um hardware dedicado, o que não traz maiores complicações.

O próximo passo será desenhar a imagem na tela. Como é um código relativo à renderização, deverá estar inserido no método Draw, como mostra a listagem abaixo:

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

    // TODO: Add your drawing code here
    this.batch.Begin();
    this.batch.Draw(this.imagem, Vector2.Zero, Color.White);
  this.batch.End();
}
```

Foram passados ao método Draw de batch um total de 3 parâmetros, a saber: a imagem a ser exibida, sua posição na tela e sua cor. No caso, Color.White significa que ela será desenhada em sua cor original. Pode-se desenhar quantas imagens se desejar, bem como adicionar várias linhas entre Begin e End para cada imagem em questão.

Caso seja chamado o método Draw de batch fora de um bloco delimitado por Begin e End, a exceção InvalidOperationException é lançada. Pode-se compor quantos blocos de SpriteBatch se desejar mas isto pode incorrer em um queda de performance.

A figura 13 exibe a saída do programa, exibindo a imagem na posição indicada.

Figura 13: Exibindo a imagem.

4.2 Exibindo Modelos 3D

Para exibir modelos 3D, o código para carregar é parecido ao apresentado para as imagens. Será utilizado como exemplo um modelo 3D contido no arquivo sokoban.fbx, como exibido na figura 14. Ele foi importado ao projeto e colocado na pasta Content. Colocou-se o nome de sokoban, que será usado no código quando se desejar referenciá-lo.

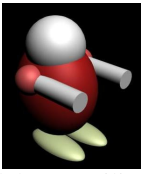

Figura 14: Modelo 3D utilizado no exemplo

Para receber o modelo, foi adicionado um atributo à SokobanGame chamado sokoban, do tipo Model, pertencente ao namespace Microsoft.Xna.Framework.Graphics.

```
public class SokobanGame : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    ContentManager content;

    //Novos atributos
    Model sokoban;

    Vector3 cameraPosition = Vector3.Zero;
```

```
Matrix view = Matrix.Identity;
Matrix projection = Matrix.Identity;
Matrix world = Matrix.Identity;
```

A listagem acima mostra os atributos da classe SokobanGame, alguns já descritos, como graphics, content e sokoban além dos novos. A seguir a explicação sobre os 3 novos atributos: view, projection e cameraPosition.

A matriz que foi chamada de view é responsável pela abstração da câmera no mundo 3D. Sua criação implica na definição da sua posição, direção do alvo e do up Vector, vetor que determina o eixo vertical da câmera.

Já a matriz projection define o cone de visão da câmera (frustum). Os parâmetros passados são os seguintes:

Near plane: distância entre a posição da câmera e a base menor do cone.

Far plane: distância entre a câmera e a base maior do cone.

Field of View: angulo de abertura do cone de visão

Aspect Ratio: fator de escala

Por último, deve-se definir a matriz que define as transformações que o modelo deve sofrer, como rotação, translação ou escala. Neste exemplo não será feita nenhuma alteração, pois será exibido com as configurações default. Por esta razão a matriz da listagem 7 recebe o valor de uma matriz identidade, ou seja, uma matriz que não influencia no cálculo das transformações. A figura 15 ilustra o conceito apresentado:

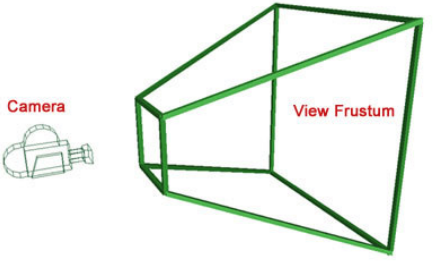

Figura 15: Cone de visão da camera (frustum).

Com isso já é possível carregar o modelo e definir valores às matrizes. A listagem a seguir mostra como definir isto no XNA.

```
protected override void LoadGraphicsContent(bool
                                            loadAllContent)
  {
      if (loadAllContent)
```

```
    {
        GraphicsDevice device = graphics.GraphicsDevice;
        sokoban = content.Load<Model>(@"Content\sokoban");

        float fieldOfView = MathHelper.ToRadians(45);
        float aspectRatio = device.Viewport.Width /
                            device.Viewport.Height;
        float nearPlane = 1.0f;
        float farPlane = 10000.0f;

        cameraPosition = new Vector3(300.0f, 300.0f, 300.0f);

        projection =
           Matrix.CreatePerspectiveFieldOfView(fieldOfView,
                  aspectRatio, nearPlane, farPlane);
        view = Matrix.CreateLookAt(cameraPosition,
                  Vector3.Zero, Vector3.Up);
    }
```

Neste caso, o modelo é composto por várias partes (cabeça, braços, corpos, pés). O conceito de modelo presente no XNA é um pouco diferente do que se costuma ver nos softwares de modelagem 3D, como Maya ou 3DS Max. Aqui, o modelo é o arquivo todo, e os modelos dentro do arquivo são os chamados de meshes, sendo que cada um tem seu nome, sua cor, texturas, etc.

Uma vez carregado o modelo, deseja-se efetivamente plotá-lo na tela. A listagem abaixo mostra como isto é realizado.

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

    foreach (ModelMesh mesh in this.sokoban.Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
            effect.EnableDefaultLighting();
            effect.View = this.view;
            effect.Projection = this.projection;
            effect.World = this.world;
        }

        mesh.Draw();
    }
```

O laço externo percorre todas as partes que compõem o modelo. Para cada uma delas, aplicam-se os efeitos designados. Estes efeitos são na maioria das vezes os shaders que serão carregados para a GPU, correspondentes ao tratamento do material e iluminação para a parte da malha em questão. Neste caso não foi definido nenhum efeito, e portanto foi usado o BasicEffect (default) ao invés de Effect.

A figura 15 mostra o resultado obtido ao se executar o código desenvolvido até mo momento.

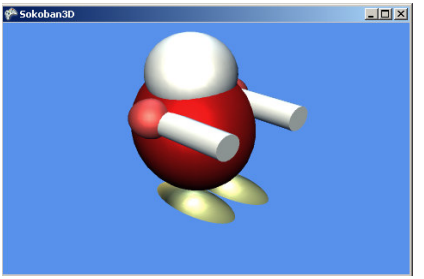

Figura 16: Modelo do boneco renderizado usando XNA.

5 Criando telas para a aplicação

Seria possível inserir todo o código da aplicação na classe SokobanGame. Porém em termos de design de software isto não seria uma solução correta. A solução de organização que se propõem baseia-se na arquitetura do template do XNA, chamado de Spacewar Starterkit.

Assim, será criado um exemplo simples, composto por 3 telas. As regras de interação com o usuário são as seguintes:

1ª tela: (inicial) aceitará a tecla “Enter” do teclado. Ao acontecer isto, ela será trocada pela segunda tela (figura 17);

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 (figura 18);

3ª tela: (congratulações) aceitará a tecla “Space” do teclado. Ao acontecer isto, ela será trocada pela primeira tela e o jogo recomeça (figura 19).

Figura 17: Primeira tela.

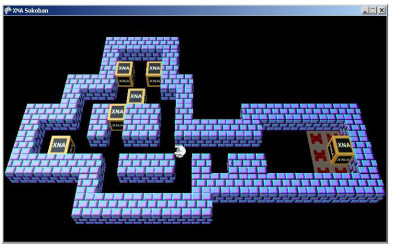

Figura 18: Segunda tela.

Figura 19: Terceira tela.

A idéia geral deste esquema é que cada tela terá sua própria classe, descrevendo o seu comportamento. Além disso, será criada uma classe abstrata que conterá propriedades e métodos comuns a todas elas. Mais adiante será explicado como fazer para saber qual é a tela atual. A estrutura proposta permite que se possam inserir mais telas, de maneira limpa e sem afetar as já existentes.

5.1 Criando e adicionando novas classes ao projeto

A seguir será criada uma classe abstrata chamada Screen, que posteriormente será adicionada ao projeto. Para isto deve-se seguir os seguintes passos:

- Clicar com o botão direito do mouse sobre o nome do projeto. Um menu pop-up aparecerá como na figura. Escolher Add / New Item...

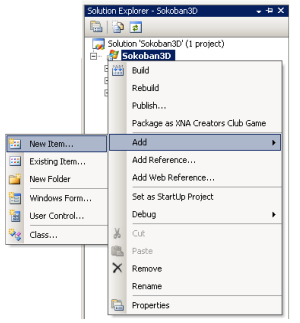

Figura 20: Adicionando novas classes

- A figura 21 mostra a caixa de diálogo que pede para escolher o tipo de item que se deseja criar. Selecionar Class, dando o nome de Screen.cs e clicando logo em seguida em Add.

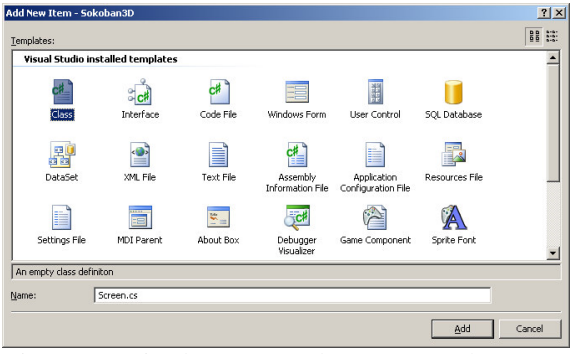

Figura 21: criando e nomeando uma nova classe

- A classe recém-criada foi adicionada ao Solution Explorer e seu código básico já inserido no XNA GSE. Deve-se defini-la como sendo abstrata e colocar os atributos e métodos que devem ser comuns a todas as telas. A listagem abaixo dá a idéia de como deve ficar a classe

```
abstract public class Screen
{
    private SpriteBatch batch = null;
    private Game game = null;
    public Game GameInstance { get { return game; } }
    public SpriteBatch SpriteBatch { get { return batch; } }
```

```
        public Screen(Game game)
        {
            this.game = game;

            if (game != null)
            {
                IGraphicsDeviceService graphicsService =
                (IGraphicsDeviceService)
game.Services.GetService(typeof(IGraphicsDeviceService));
batch = new SpriteBatch(graphicsService.GraphicsDevice);
            }
        }


        public virtual GameState Update(GameTime gameTime)
        {
            return GameState.None;
        }
        public virtual void Render() { }
        public virtual void Shutdown()
        {
            if (batch != null)
            {
                batch.Dispose();
                batch = null;
            }
        }
    }
```

Deve-se guardar uma referência ao código principal para poder chamar os seus métodos quando forem necessários. Será introduzido código na classe SokobanGame adicionando-se uma propriedade chamada currentScreen. Abaixo um esboço do que será este código:

```
    public class SokobanGame : Microsoft.Xna.Framework.Game
    {
        //outras propriedades
          private Screen currentScreen;
          public SokobanGame()
          {
              //inicialização
          }
```

```
        //outros métodos
  }
```

5.2 Criando enumerações

Para compor uma enumeração, será necessário criar mais um arquivo para o projeto. Ele na verdade conterá todos os tipos enumerados do código e estará sob o mesmo namespace dos outros, que é Sokoban3D. Os passos para criar uma enumeração são idênticos ao da criação de uma classe. Foi dado o nome de Enum.cs para este exemplo. Ao criar o stub da classe, a palavra reservada class presente no código será trocada pela palavra enum.

Neste exemplo, esta enumeração será criada com o propósito de representar os estados do jogo e por isto foi dado o nome de GameState à enum.

A listagem a seguir mostra como ficou arquivo.

```
  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 diz, esta estrutura terá a responsabilidade de dizer em qual estado o jogo se encontra. Se há mudanças de estado, há mudanças de tela. Assim o núcleo do jogo atua como uma máquina de estados. A seguir será mostrado como compor o código responsável pela mudança das telas.

5.3 Criando métodos auxiliares no SokobanGame

Deve-se inserir um novo método para que seja responsável pela mudança de estados do jogo. Este método receberá como parâmetro o novo estado e fará com que a referência currentScreen da classe principal aponte pra outra tela, chamando antes o método "Shutdown" da tela atual. A listagem abaixo ilustra esta implementação:

```
  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);
      }
}
```

5.4 Utilizando herança

Já existe no exemplo em questão o código para mudar as telas, porém ainda falta definir como e quais são estas telas. Para tanto, serão criadas 3 classes que herdam de Screen suas propriedades: IntroScreen, SokobanScreen, CongratulationsScreen. Posteriormente, será necessário implementar seus respectivos métodos Render e Update com códigos apropriados.

Para exemplificação na utilização de herança em C#, será usado o método IntroScreen:

```
   public class IntroScreen : Screen
```

Os dois pontos após o nome da classe seguidos do nome de uma classe/interface caracterizam uma herança .

Com os arquivos já criados, será discutido o que será colocado em cada tela:

- IntroScreen
- CongratulationsScreen
- SokobanScreen

5.4.1 IntroScreen

Como já dito, aqui só será necessário 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;
```

Cria-se a seguir dois atributos para ela: logo e oldKeyState. O primeiro é a imagem de boas vindas, que será exibida na primeira tela. O segundo serve como um atributo auxiliar na captura de teclas. A seguir será carregada a imagem para seu uso dentro do método LoadResources:

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

Cria-se então uma chamada para este método dentro do construtor da classe, além de inicializar o atributo oldKeyState como se segue:

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

Neste estágio já se pode começar a implementar os métodos Render e Update de IntroScreen. Eles serão chamados externamente por SokobanGame.

No método Render, tudo que se deve fazer é criar um código para exibir a imagem carregada para a memória. Para isto torna-se necessário novamente o objeto SpriteBatch, também provido pelo XNA, responsável pela renderização de objetos 2D. Como a classe herda várias propriedades de Screen, SpriteBatch já está disponível neste momento. A criação de Screen abstrata com propriedades comuns a uma tela é de grande utilidade.

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();
}
```

Deve-se agora implementar o método Update de IntroScreen com código para capturar os eventos do teclado. Este será chamado em intervalos de tempo regulares, assim como Render. Por isto foi criado 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. Observe-se que ele retorna um estado do jogo. SokobanGame monitora a tela atual capturando os valores de retorno das telas para fazer a mudança.

5.4.2 CongratulationsScreen

É Idêntico à IntroScreen, apenas com diferença da imagem 2D que deve ser carregada, que é o arquivo CongratulationsScreen.png, adicionado ao projeto na pasta Content.

5.4.3 SokobanScreen

Esta classe corresponde a tela do jogo em si. Aqui serão carregados os modelos 3D e o código é um pouco mais "elaborado" em relação às outras telas: mais atributos e métodos, uma vez que é o local onde reside a implementação da lógica do jogo. Além disso, será utilizada uma classe auxiliar para carregar o arquivo de configuração contendo os níveis do Sokoban.

Segue um resumo do que está presente nesta seção do programa:

- Os modelos são carregados dos arquivos de origem e colocados na memória;
- As texturas são carregadas dos arquivos para a 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 1º nível para plotar os objetos 3D;
- Para cada objeto a ser plotado, verifica-se sua posição, aplica-se a rotação (se necessário) e a textura correspondente, com os respectivos efeitos, se necessário;
- Recebe a entrada do teclado, trata as colisões, atualiza as posições;
- Atualiza a tela, com as novas posições sendo refletidas;
- Caso o jogador passe para um novo nível, atualiza a matriz do nível;
- Atualiza a tela refletindo as novas posições...

Além dos métodos herdados de Screen, na classe SokobanScreen, conta-se também com os seguinte métodos:

- `public void RenderModel(Model m, Texture2D modelTexture, Vector3 mPosition, float scale, float mRotation)`
- `private void CharacterMove(CharacterMovements vertical, CharacterMovements horizontal)`
- `private void RestartCurrentLevel()`

Estes métodos são explicados um a um mais adiante. Os atributos da classe são:

```
//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 referencial, este parâmetro diz quanto o próximo objeto estará distante dele
- offsetX e offsetY: distância 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, o atributo 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 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.

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 ao 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 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 já foi apresentado na etapa de visualização de um modelo 3D. Ele contém vários conceitos de visualização 3D envolvidos que serão explicados a seguir.

```
  Matrix[] transforms = new Matrix[m.Bones.Count];
  m.CopyAbsoluteBoneTransformsTo(transforms);

  // Desenho do modelo, composto por diversas malhas
  foreach (ModelMesh mesh in m.Meshes)
  {
      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;
          }
      }

      // Desenha a malha com o efeito apropriado
      mesh.Draw();
}
```

Cada transformação deve ter uma matriz própria para ela, porém elas podem ser combinadas e o XNA permite isto. Veja no exemplo uma linha que implementa isto:

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

Esta combinação é feita através de multiplicações de matrizes conforme o conceito de matrizes afim.

Para definir um Frustrum, é necessário definir os seguintes parâmetros:

- Angulo de abertura do cone
- Near plane: a menor base do cone, ou seja, a partir de quê distância, partindo da câmera, os objetos vão ser considerados visíveis para a renderização
- Far plane: a maior base do cone, ou seja, até onde, a partir do near plane, os objetos serão considerados para a renderização.

No XNA isto é definido da seguinte forma:

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

É possível aplicar efeitos extras sobre os Meshes dos modelos usando arquivos de configuração escritos em HLSL (High Level Shader Language), como será visto na sessão 6, porém neste instante apenas será usado o efeito padrão BasicEffect.

- SokobanScreen.Update: Aqui será implementada a lógica do jogo propriamente dita. Ao apresentar o método de Render, viu-se como atualizar a tela. Neste método será feita a atualização de todo o restante, levando em conta a entrada do usuário.

a. 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. Posteriormente compara-se com o estado anterior para ver quanto ele se moveu. Com esta informação, atualiza-se a posição da câmera no mundo 3D.

b. Carregar próximo nível

```
  //Deve-se 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.

c. 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 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 seja mais plausível. Quando isto ocorre, verificam-se as teclas pressionadas e, conforme for, move-se o boneco para a posição indicada e o gira-se 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 pela sua posição, escrita na matriz. A "colisão" é verificada pelas células adjacentes à sua célula.

O método recebe como parâmetro 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];
```

O código acima representa a posição adjacente ao boneco no mundo 3D do nível, ou seja, nearPosition representa a posição que tem a maior probabilidade de ocorrer uma colisão.

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

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

```
  //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. Isto é necessário para obter-se algumas informações, como por exemplo saber se é possível o boneco mover uma caixa para algum lugar. greenBoxChecking e goldBoxChecking indica se há alguma caixa à frente do boneco.

```
  //se boneco está 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));
```

As linhas acima são responsáveis para colher a informação do ambiente: “é possível mover a caixa”? A condição a seguir verifica isto:

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

A listagem a seguir atualiza 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 serviria 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 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. Isto acontece quando o usuário tecla "R" em um nível qualquer. Esta operação também é necessária caso o jogador erre ou fique preso por conta própria no estoque.

6 Shaders no XNA

6.1 Introdução

Um tópico que se tornou muito importante no desenvolvimento de jogos nos últimos anos foi a programação e desenvolvimento de shaders. Shaders são pequenos programas que são executados no hardware gráfico programável (GPU – Graphics Processing Unit), e permitem modificar alguns estágios do pipeline de renderização. Antes da introdução dos shaders, as APIs gráficas como DirectX e OpenGL, possuíam um pipeline de renderização completamente fixo. Isso fazia com que todos os jogos utilizassem um mesmo processo de renderização, sendo possível apenas modificar parâmetros do mesmo. Por exemplo, para

iluminar uma cena, era possível escolher o tipo de fontes de luz (direcional, holofote e omnidirecional), e configurar os seus parâmetros. No entanto, o número de fontes de luz disponível era bastante limitado e definido pela API, além disso, não era possível criar novas fontes de luzes, nem modificar o algoritmo de iluminação que era utilizado. A flexibilidade de se programar a GPU tornou possível adicionar aos jogos efeitos visuais que antes só eram vistos em filmes. Esses efeitos geralmente eram criados utilizando ferramentas como RenderMan [HANRAHAN], que apesar de possuir uma linguagem de shaders não tira proveito dos hardwares gráficos programáveis e não pode ser utilizada para aplicações de tempo real. A Figura 22 exibe os vários estágios de um pipeline de renderização.

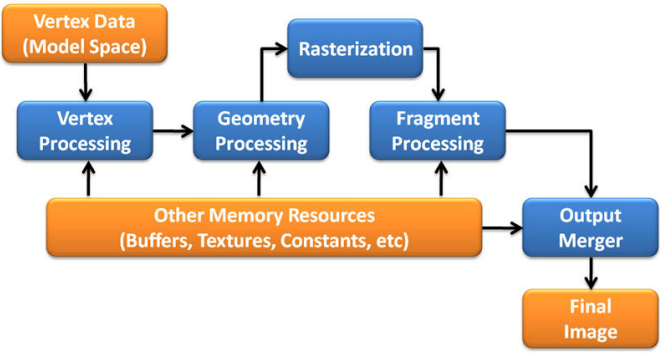

Figura 22: Pipeline de renderização

A entrada principal do pipeline da Figura 24 são os vértices da malha do modelo, onde cada vértice pode conter informações de posição, cor, normal, coordenada de textura, dentre outros. Vários estágios do pipeline têm acesso a outros recursos de memória, como constantes passadas pela aplicação e texturas. A saída final do pipeline é uma imagem na forma de uma matriz de pixels, que é exibida ao usuário.

7 Programação de Shaders

O primeiro passo para programar shaders é escolher uma API gráfica. Neste tutorial estaremos utilizando a API gráfica do XNA, que utiliza como base a API gráfica do DirectX. É importante notar que o XNA possui apenas o pipeline de renderização programável. Isso ocorre devido ao hardware gráfico do XBox 360 não possuir um pipeline fixo, sendo necessário o uso de shaders. Apesar de não possuir o pipeline programável o XNA possui a classe Helper BasicEffect que implementam uma pequena parte do pipeline fixo de renderização utilizando shaders.

Após escolher a API gráfica, deve-se escolher uma linguagem de shader, entre as linguagens suportadas pela API. Os shaders podem ser programados em uma linguagem

assembly de baixo nível, ou utilizando linguagens de mais alto nível similares a linguagem C. Algumas das linguagens de alto nível existentes são HLSL (Microsoft), Cg (nVidia) e GLSL (3DFX). O DirectX utiliza como padrão a linguagem HLSL, enquanto o OpenGL utiliza como padrão a linguagem GLSL. Já a linguagem Cg, que possui uma sintaxe muito similar ao HLSL, pode ser utilizada por ambas as APIs, DirectX e OpenGL.

O XNA possui suporte a shaders através de efeitos, criados utilizando a linguagem HLSL. Efeitos são programas que encapsulam várias técnicas de renderização, utilizando shaders de processamento de vértices, geometrias e pixels. Por exemplo, um efeito pode ter técnicas para renderizar um objeto com uma cor uniforme, utilizando uma textura, ou utilizando várias texturas sobrepostas. Na aplicação a técnica utilizada para renderizar o objeto pode ser escolhida de acordo com o desempenho do computador utilizado.

As diferentes técnicas dentro de um efeito podem compartilhar um mesmo processamento de vértices, geometrias ou pixels, sendo possível reutilizar as funções de processamento criadas em diferentes técnicas. Outra vantagem do uso dos efeitos é que eles permitem a configuração dos estados do pipeline de renderização, como Alpha Blending (Mistura de cores baseado na componente Alpha), teste de profundidade e outros.

7.1 Criando efeito

Nesta seção será criado um efeito de iluminação por pixel utilizando a equação de Phong apresentada, que será utilizado na renderização dos objetos de uma cena. A Tabela 1 apresenta algumas das funções de linguagem HLSL, utilizadas na construção do efeito.

dot	Calcula o produto escalar entre dois vetores.
mul	Realiza multiplicação entre vetores e matrizes.
nor	Normaliza um vetor.
refle	Reflete um vetor incidente a uma normal.
satur	Satura os valores entre 0.0 e 1.0

Tabela 1: Funções do HLSL utilizadas no efeito criado.

Quando se cria um efeito a primeira coisa que deve ser declarado são os dados que serão recebidos da aplicação. Estes dados são declarados como variáveis globais no efeito. A listagem 31 apresenta as variáveis utilizadas pelo efeito.

```
// Matrix
// ------------------------------------------------
float4x4 matW : World;
float4x4 matV : View;
float4x4 matVI : ViewInverse;
float4x4 matWV : WorldView;
float4x4 matWVP : WorldViewProjection;

// Ambient light
// ------------------------------------------------
float3 ambientLightColor;

// Light 0
// ------------------------------------------------
float3 lightPosition;
float3 lightColor;

// Material
// ------------------------------------------------
float3 materialColor;
float materialKd;
float materialKs;
float materialShininess;
```

As matrizes utilizadas pelo efeito são as matrizes de mundo, visão e projeção. Note-se que algumas combinações dessas matrizes, como matWV (Mundo * Visão) e matWVP (Mundo * Visão * Projeção), também são declaradas para não precisarem ser calculadas no efeito. Em seguida declara-se a cor da componente ambiente de luz da cena, a posição e cor de uma fonte de luz omnidirecional e as propriedades do material da superfície. Todas essas informações são utilizadas no cálculo de iluminação.

Deve-se declarar também uma estrutura com os dados dos vértices da malha do modelo que serão enviados ao processamento de vértices. E uma outra estrutura, que define os dados de saída do processamento de vértice, que serão utilizados como entrada do processamento de pixels. A listagem abaixo apresenta as estruturas criadas.

```
// Application to Vertex
// ------------------------------------------------
struct a2v
{
    float4 position   : POSITION;
    float3 normal     : NORMAL;
};

// Vertex to Fragment
// ------------------------------------------------
struct v2f
{
    float4 hposition  : POSITION;
    float3 normal     : TEXCOORD1;
    float3 lightVec   : TEXCOORD2;
    float3 eyeVec     : TEXCOORD4;
};
```

A estrutura a2v define os dados de entrada do processamento de vértices, e a estrutura v2f define os dados de saída do processamento de vértice. Os dados de entrada são a posição do vértice e seu vetor normal, e os dados de saída são a posição final do vértice e os vetores de normal, luz e visão.

Após declarar as variáveis e estruturas utilizadas pelo efeito pode-se criar o código do processamento de vértices e pixels. A seguir é apresentado o código da função de processamento de vértices criada. Observe-se que a entrada da função é a estrutura a2v, e a saída a estrutura v2f.

```
v2f LightingVS(a2v IN)
{
    v2f OUT;

    // Transform vertex and normal
    OUT.hposition = mul(IN.position, matWVP);
    OUT.normal = mul(IN.normal, matWV);

    // Calculate light and eye vectors
    float4 worldPosition = mul(IN.position, matW);

    float3 eyePosition = matVI[3].xyz;
    OUT.eyeVec = mul(eyePosition - worldPosition, matV);
    OUT.lightVec = mul(lightPosition - worldPosition, matV);

    return OUT;
}
```

No processamento de vértices a posição final do vértice é calculada, e a matriz transformada de acordo com as matrizes de mundo e visão. Em seguida são calculados os vetores que apontam do vértice para a luz e para o observador da cena. Repare-se que no final do processamento todos estes vetores se encontram no espaço de visão. O código a seguir apresenta a função utilizada para o processamento de pixels. A seguir é apresentado o código da função de processamento de pixels.

```
float4 LightingPS(v2f IN): COLOR0
{
    // Normalize all input vectors
    float3 normal = normalize(IN.normal);
    float3 lightVec = normalize(IN.lightVec);
    float3 eyeVec = normalize(IN.eyeVec);

    // Calculate reflected light
    float3 reflectLight = reflect(-lightVec, normal);

    // Calculate diffuse and specular intensity
    float diffuseInt = saturate(dot(normal, lightVec));
    float specularInt = saturate(dot(eyeVec, reflectLight));
    specularInt = pow(specularInt, materialShininess);

    // Modulate by the light color
    float3 diffuseColor = diffuseInt * materialKd * lightColor;
    float3 specularColor = specularInt * materialKs * lightColor;

    // Final output color
    float4 finalColor;
        finalColor.rgb = specularColor + materialColor *
        (ambientLightColor + diffuseColor);
    finalColor.a = 1.0f;

    return finalColor;
}
```

No processamento de pixels, a equação de Phong apresentada na Seção 6.1 é calculada para cada pixel e a cor final do pixel é gerada. Por fim, deve-se criar uma técnica que defina qual processamento de vértice e pixel se deve utilizar. A seguir é apresentado o código da técnica para iluminação criada.

```
technique Lighting
{
    pass p0
    {
        VertexShader = compile vs_2_0 LightingVS();
        PixelShader = compile ps_2_0 LightingPS();
    }
}
```

7.2 Implementação no XNA

Após o efeito ser criado, o mesmo já está pronto para ser carregado e usado pelo XNA. O pipeline de conteúdo do XNA fica responsável em compilar o efeito e gerar um binário em um formato otimizado para ser utilizado no Windows ou no XBox 360. O efeito gerado pelo pipeline de conteúdo pode ser facilmente carregado utilizando o gerenciador de conteúdo do XNA (ContentManager). O gerenciado de conteúdo também será utilizado para

carregar o modelo de uma chaleira, que será renderizado utilizando o efeito criado. A seguir serão apresentados 6 passos de como criar um novo projeto no XNA e renderizar uma chaleira utilizando o efeito criado.

Passo 1 – Criar o projeto e adicionar os assets.

Primeiro deve-se criar um novo projeto para a aplicação e adicionar os artefatos necessários no projeto. Os artefatos utilizados serão um modelo de chaleira no formato .X (DirectX File), e o efeito criado no formato .FX (Effect).

Passo 2 – Declarar os objetos que serão utilizados.

Em seguida deve-se declarar os objetos que serão utilizados no programa como variáveis globais.: Matrix cameraView, cameraProjection; Model model; Effect lightEffect;

As variáveis cameraView e cameraProjection serão utilizados para armazenar as matrizes de visão e projeção da câmera. A variável model e lightEffect serão utilizados para armazenar o modelo da chaleira e o efeito criado.

Passo 3 – Configurar a câmera

Dentro do método Initialize do framework pode-se configurar a câmera.

```
protected override void Initialize()
{
    cameraView = Matrix.CreateLookAt(
     new Vector3(3.0f, 2.0f, 4.0f),
     Vector3.Zero,
     new Vector3(0.0f, 1.0f, 0.0f));

    cameraProjection = Matrix.CreatePerspectiveFieldOfView(
     MathHelper.ToRadians(45.0f), 1.0f, 0.01f, 1000.0f);

    base.Initialize();
}
```

A câmera está na posição (3, 2, 4), observando a posição (0, 0, 0), e o eixo (0, 1, 0) define a orientação da câmera.

Passo 4 – Carregar e configurar o efeito

O efeito do modelo deve ser carregado utilizando o gerenciador de conteúdo dentro do método LoadGraphicsContent.

```
 protected override void LoadGraphicsContent(bool
                         loadAllContent)
 {
     if (loadAllContent)
     {
         // Load the effect
         lightEffect = content.Load<Effect>("lightEffect");

         // Configure the matrices

lightEffect.Parameters["matW"].SetValue(Matrix.Identity);
         lightEffect.Parameters["matV"].SetValue(cameraView);
         lightEffect.Parameters["matVI"].SetValue(
                    Matrix.Invert(cameraView));
         lightEffect.Parameters["matWV"].SetValue(cameraView);
         lightEffect.Parameters["matWVP"].SetValue(
                    cameraView * cameraProjection);

         // Configure the light parameters
         lightEffect.Parameters["ambientLightColor"].SetValue(
                    new Vector3(0.2f, 0.2f, 0.2f));
         lightEffect.Parameters["lightPosition"].SetValue(
                    new Vector3(10.0f, 10.0f, 10.0f));

lightEffect.Parameters["lightColor"].SetValue(Vector3.One);

         // Configure the material parameters
         lightEffect.Parameters["materialColor"].SetValue(
                    new Vector3(1.0f, 1.0f, 0.0f));
         lightEffect.Parameters["materialKd"].SetValue(0.7f);
         lightEffect.Parameters["materialKs"].SetValue(0.3f);

lightEffect.Parameters["materialShininess"].SetValue(32.0f);
     }
 }
```

Para carregar o modelo utiliza-se o método Load da classe ContentManager, especifica-se o tipo de artefato que está carregando “Effect”, e por último passa-se o nome do efeito sem extensão. Após carregar o efeito, os seus parâmetros (variáveis globais declaradas no efeito), podem ser acessadas através do atributo Parameters da classe Effect.

No efeito, se está definindo a posição da luz como (10, 10, 10) e sua cor como branca. Em seguida define-se a cor do material como amarelo e seus coeficientes de reflexão.

Passo 5 – Carregar o modelo e modificar o efeito utilizado

O modelo da chaleira deve ser carregado da mesma maneira que o efeito, dentro do método LoadGraphicsContent.

```
protected override void LoadGraphicsContent(bool
                        loadAllContent)
   {
       if (loadAllContent)
       {
           // Load the effect
           // ... ... ...

           // Load model
           model = content.Load<Model>("teapot");

           // Change the model mesh effect
           model.Meshes[0].MeshParts[0].Effect = lightEffect;

       }
   }
```

Quando o modelo é carregado o mesmo já possui um efeito para renderização. Este efeito pode ter sido exportado junto com o modelo, ou criado pelo pipeline de conteúdo. Após carregar o modelo troca-se o efeito utilizado na renderização da sua malha pelo efeito carregado.

Passo 6 – Desenhar o modelo

O último passo necessário é desenhar o modelo carregado. Para isso basta chamar o método Draw de cada malha do modelo.

```
   protected override void Draw(GameTime gameTime)
   {
       graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

       foreach (ModelMesh mesh in model.Meshes)
           mesh.Draw();

       base.Draw(gameTime);
   }
```

8 Referências

[1] Blinn, J., Models of light reflection for computer synthesized pictures, Proceedings of the 4th annual conference on Computer graphics and interactive techniques, p.192-198, 1977.

[2] Blythe, D., The Direct3D 10 system. In proceedings of SIGGRAPH 2006, 724-734.

[3] Foley, J., van Dam, A., Feiner, S., and Hughes, J. Computer Graphics: Principles and Practice, 2nd edition. Addison Wesley, 1997.

[4] Hanrahan, P., and Lawson, J. 1990. A language for shading and lighting calculations. In proceedings of SIGGRAPH 90, 289-298.

[5] Möller, T., and Haines, E., Real-Time Rendering. 2ª ed. A. K. Peters, 2002.