GUIA PRÁTICO PARA INICIANTES – MICROSOFT® XNA

Por: Fernando Birck (Fergo)

SUMÁRIO

1. INTRODUÇÃO

Em Novembro de 2006, a Microsoft trouxe ao mundo a sua nova plataforma de programação gráfica, intitulada Microsoft XNA Game Studio. Essa plataforma funciona como um meio de conexão entre as APIs do DirectX e o programador, possuindo uma série de funcionalidades e rotinas previamente compiladas, facilitando ao máximo o trabalho com efeitos e geometria espacial.

Um exemplo dessa funcionalidade seria a capacidade nativa de carregar modelos, sons e texturas com apenas uma linha, sem precisar se preocupar em entender e decifrar o formato dos arquivos, como ocorreria ao se trabalhar diretamente com o DirectX.
Essas características facilitam bastante a criação de um pequeno jogo, sendo voltadas mais para os entusiastas e iniciantes nesse ramo e que não estão tão preocupados em deixar o jogo 100% otimizado, mas sim em poder ver rapidamente os resultados de sua criação.

A linguagem padrão escolhida pela Microsoft foi o C# (C Sharp), mas como o processo utiliza o .NET Framework, qualquer linguagem .NET é capaz de rodar o XNA, bastando apenas alterar a sintaxe. Neste tutorial vamos trabalhar com o próprio C#, já que é a linguagem “oficial” do XNA GS.

Toda a plataforma do XNA é distribuída gratuitamente pela Microsoft, sendo necessário apenas possuir o C# Express e o XNA Game Studio instalados. Veja no próximo capítulo as ferramentas necessárias.

2. APLICATIVOS NECESSÁRIOS

- Microsoft Visual C# 2005 Express Edition
  - IDE de programação para o C#.
  - http://msdn.microsoft.com/vstudio/express/downloads/

- Visual C# 2005 Service Pack 1
  - Correções para o C# 2005.
  - http://download.microsoft.com/download/7/7/3/7737290f-98e8-45bf-9075-85cc6ae34bf1/VS80sp1-KB926749-X86-INTL.exe

- XNA Game Studio
  - Framework de programação gráfica.
  - http://msdn2.microsoft.com/en-us/xna/aa937795.aspx

- SketchUp 6 – Trial
  - Aplicativo simples de modelagem 3D.
  - http://www.sketchup.com/?sid=509

- Deep Exploration
  - Conversor de modelos e imagens.
  - http://superdownloads.uol.com.br/download/64/deep-exploration/

- Código fonte, modelos, texturas e efeitos.
  - Pacote contendo todos os arquivos usados no guia
  - http://fergonez.net/files/xna_arquivos.rar

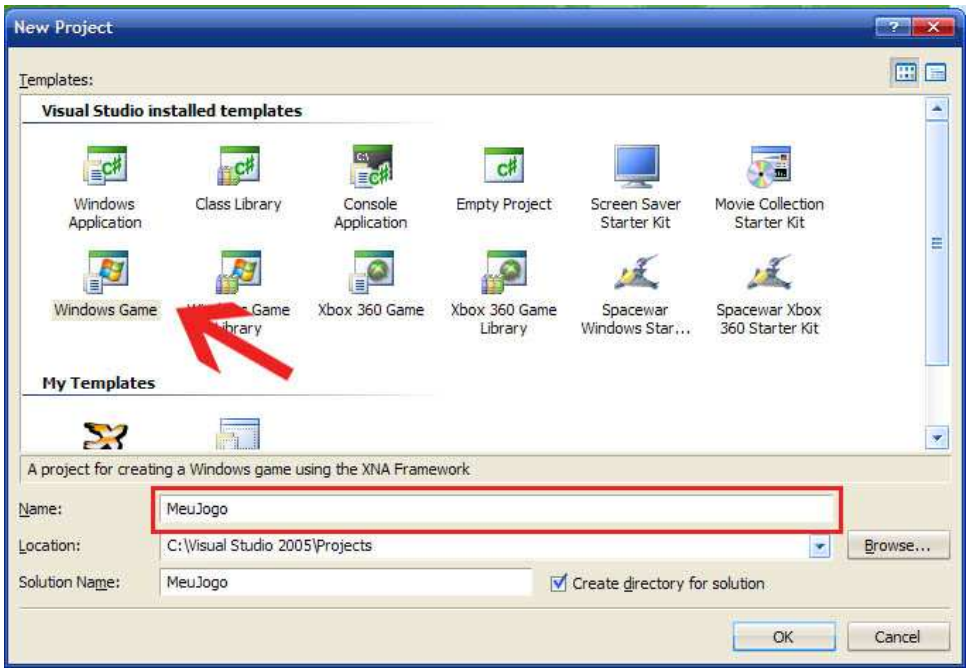
Os dois últimos aplicativos (SketchUp e Deep Exploration) não são gratuitos, mas possuem versões de testes que são suficientes para o nosso tutorial. O SketchUp é um aplicativo de modelagem 3D genial e muito interessante de se trabalhar, pois se diferencia um pouco dos aplicativos tradicionais. Vamos utilizá-lo para modelar nossos objetos. O Deep Exploration é um ótimo aplicativo para visualização e conversão de modelos 3D, com um suporte fantástico a quase todos os formatos existentes. Vamos converter os modelos gerados pelo SketchUp para o formato do XNA através dele.

Com todas as ferramentas em mãos, podemos dar inicio a programação da nossa engine básica.

3. CRIANDO UM PROJETO

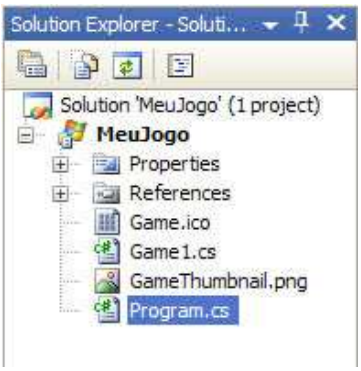
Após ter instalando todos os componentes do capítulo anterior, navegue pelo menu Iniciar até encontrar o aplicativo do XNA Game Studio (normalmente ele se encontra em “Iniciar -> Programas -> Microsoft XNA Game Studio Express -> XNA Game Studio Express”).

Com a IDE carregada, vá até o menu “File -> New Project”. Na janela que aparecer, selecione o item “Windows Game”, dê um nome para o seu projeto (nesse caso, utilizei “MeuJogo”) e confirme a criação do projeto.



Após confirmar o projeto, o Visual C# cria uma estrutura de arquivos padrão para os jogos que utilizam o XNA. Neste tutorial é recomendável que você já possua certo conhecimento de programação e conheça a interface do Visual Studio, pois não trataremos deste assunto nesse guia.

No lado direito, em “Solution Explorer”, temos todos os arquivos gerados pelo C#, sendo que vou frisar alguns deles.



Game.ico – Ícone do seu jogo. Aparecerá no topo da janela e no arquivo executável
Game1.cs – Este arquivo contém o código da classe do jogo. Trabalharemos sobre ele
Program.cs – Arquivo básico do jogo. É o ponto de partida inicial do programa, que apenas chame a classe contida no Game1.cs

Abra o arquivo “Game1.cs” (duplo clique sobre ele) no editor. Estamos agora na área de código “principal” do projeto. Nela estão as funções básicas do XNA, sobre as quais a rotina do jogo acontece.

A classe é dividida basicamente em 6 rotinas: Game1(), Initialize(), LoadGraphicsContent(), UnloadGraphicsContent(), Update() e Draw()

- **Game1** – É a rotina básica da classe do jogo, onde fica boa parte das declarações de variáveis, tipos e classes.
- **Initialize** – Dentro desta rotina ficam todas as operações de inicialização do jogo que não necessitam ter a engine gráfica rodando. Um exemplo seria a leitura de registro e arquivos de configuração do jogo.
- **LoadGraphicsContent** – Aqui é onde todos os itens gráficos do jogo são carregados, como modelos, objetos e texturas.
- **UnloadGraphicsContent** – Nesse método é onde você descarrega os objetos. Essa rotina é pouco utilizada, pois os objetos são descarregados automaticamente quando desnecessários.
- **Update** – Essa rotina fica em uma repetição durante a execução do jogo, onde é possível controlar a lógica, verificar o estado do teclado, mover objetos, etc.
- **Draw** – Semelhante à rotina acima, mas voltada para fazer a apresentação dos gráficos na tela. Tudo o que precisar ser desenhado vai dentro desse método. Fazendo uma analogia, essa função mostraria na tela o “resultado” da rotina “Update”.

4. PREPARANDO MÉTODOS

Neste capítulo vamos configurar e preparar nossas funções para iniciar a engine gráfica. O código padrão gerado pelo XNA já pode ser compilado, mesmo sem termos programado nada. Ele automaticamente cria a janela, define seu tamanho e uma cor de fundo (aperte F5 para rodar o projeto e veja).

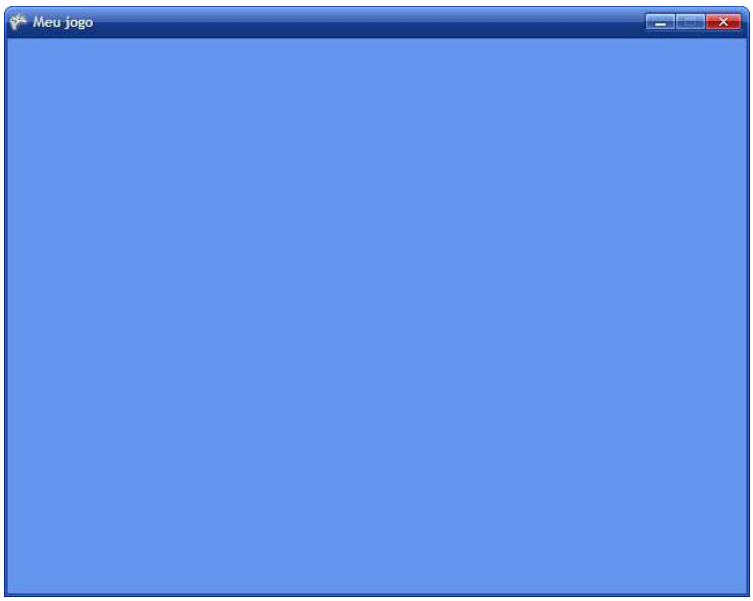
Ao invés de usar a configuração padrão definida pelo XNA, vamos manualmente configurar algumas características da janela, alterando o título e o seu tamanho. Para tal, precisamos de um "device", que nos comunica com o vídeo. O próprio XNA já declara esse device, como pode ser observado no início da classe Game1:

```
GraphicsDeviceManager graphics;
```

Vamos então determinar um tamanho para a janela, adicionar um título e desabilitar o modo "Tela Cheia". Para isso utilizaremos o nosso device, cujo nome é "graphics". Queremos que isso seja feito logo que o jogo seja iniciado, então os comandos devem ficar dentro do método "Initialize()":

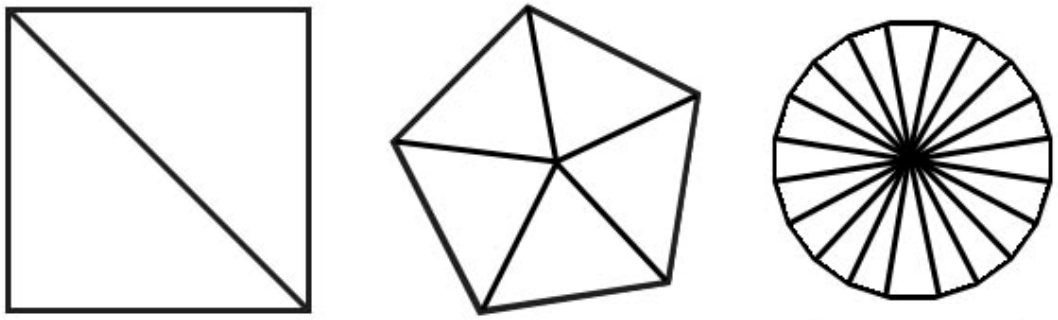
```
protected override void Initialize()
{
      graphics.PreferredBackBufferHeight = 480; // 480 de altura
      graphics.PreferredBackBufferWidth = 640; // 640 de largura
      graphics.IsFullScreen = false; // desabilita o modo tela cheia
      graphics.ApplyChanges(); // aplica as mudanças
      Window.Title = "Meu jogo"; // define um título à janela

      base.Initialize();
}
```

Salve e rode seu projeto (F5). Agora a janela criada possui uma dimensão de 640x480 pixels, como na maioria dos jogos. Você pode explorar os outros comandos do device caso queira, mas eu vou parar por aqui para não fugir muito do propósito do guia.



5. O MUNDO 3D

Todas as formas apresentadas na tela são compostas por triângulos, ou um conjunto deles. Seja qual for a forma geométrica, ela é sempre segmentada em diversos triângulos, tentando gerar o menor número possível dos mesmos. Veja a imagem abaixo que ilustra um pouco essa segmentação:

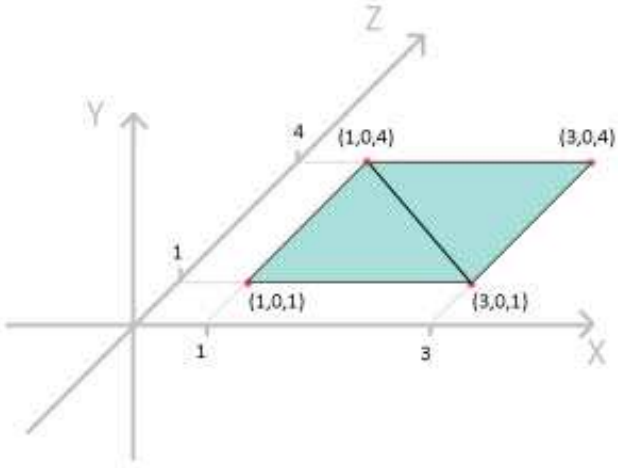


Quadrado
2 Triângulos

Pentágono
5 Triângulos

Círculo aproximado
20 Triângulos

Esses triângulos são definidos por vértices, pontos cartesianos em um espaço bi ou tridimensional e suas componentes X, Y e Z. As regras de geometria analítica que são vistas na escola também podem ser aplicadas para realizar transformações (deslocamentos, rotações, deformações) através de matrizes. Abaixo temos uma representação de um retângulo no espaço tridimensional, com suas coordenadas:

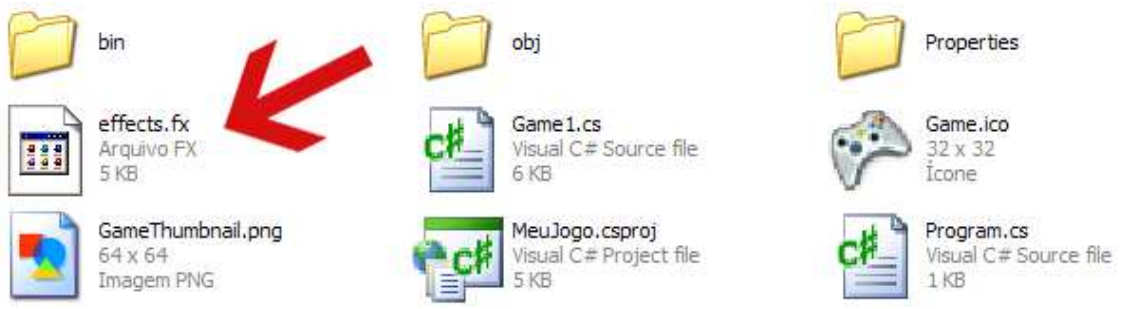


Agora que já temos uma noção de como funciona um mundo tridimensional no XNA, podemos prosseguir para desenhar nosso primeiro sólido na tela.

6. EFEITOS

O XNA trabalha de uma forma diferente que o DirectX ou OpenGL na hora de renderizar os polígonos. Para tudo o que for desenhar, é necessário um “Effect”, que determina características e a técnica do sólido desenhado. Esses efeitos são um pouco complicados de se trabalhar e fugiria do propósito do guia explicar como compor um efeito.

No arquivo RAR contendo as fontes utilizadas neste tutorial há um arquivo “effects.fx” genérico que pode ser usado em boa parte dos projetos. Coloque este arquivo na pasta raiz do projeto (na mesma pasta onde ficam os códigos fonte e os ícones).



Com o arquivo copiado, volte para o C# e declare uma variável para o efeito no início da classe do jogo, junto com a declaração do device:

```
Effect effect;
```

Esse arquivo de efeito não está compilado. Precisamos compilá-lo antes da engine poder utilizar. Essa compilação é feita durante a inicialização do jogo, dentro do método “Initialize()”:

```
CompiledEffect compiledEffect =
Effect.CompileEffectFromFile("@/../../../../effects.fx", null, null,
CompilerOptions.None, TargetPlatform.Windows);

effect = new Effect(graphics.GraphicsDevice,
compiledEffect.GetEffectCode(), CompilerOptions.None, null);
```

Tendo o efeito compilado, precisamos definir a técnica que será utilizada para renderizar. Isso é feito dentro do método “Draw()”. A técnica pode ser definida logo após o comando de limpar a tela e preenche-la com uma cor.

O nome da técnica utilizada vai depender do efeito. Para o arquivo “effects.fx”, usaremos a “Pretransformed”. Adicione o código abaixo dentro do módulo “Draw()“ após limpar a tela:

```
effect.CurrentTechnique = effect.Techniques["Pretransformed"];
effect.Begin(); // inicia a técnica

effect.End(); // finaliza a técnica
```

Como pode notar, a técnica possui um começo e um fim, sendo que você pode ter várias técnicas dentro de um só “Draw()”, basta definir blocos diferentes para cada uma.

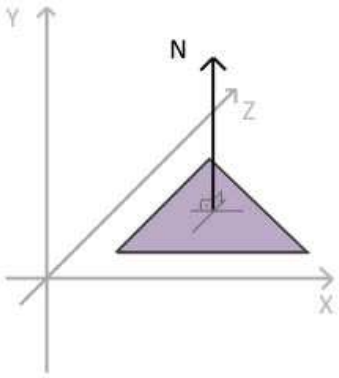
Entre os blocos definidos, é necessário preparar o efeito para ser renderizado. Isso é feito através de um processo interativo que realiza uma varredura por todos os passos contidos dentro do efeito. Essa varredura é feita através de um “foreach”. Abaixo está o código completo do método “Draw()”:

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

	effect.CurrentTechnique = effect.Techniques["Pretransformed"];
	effect.Begin(); // inicia a técnica

	foreach (EffectPass pass in effect.CurrentTechnique.Passes)
	{
		pass.Begin();

		pass.End();
	}
	effect.End(); // finaliza a técnica

	base.Draw(gameTime);
}
```

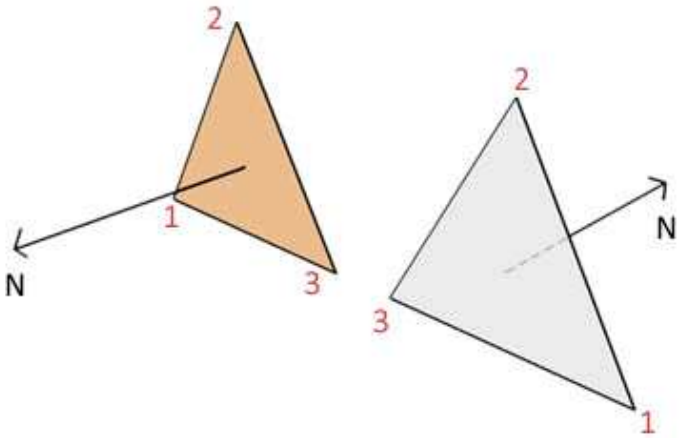
7. DESENHANDO O PRIMEIRO TRIÂNGULO

Com o conhecimento visto no capítulo anterior, podemos desenhar nosso primeiro triângulo na tela. Apesar de ser facilmente entendido, devemos ter alguns cuidados. O maior deles é quanto a ordem da disposição dos vértices. Não podemos simplesmente jogar os vértices de uma forma aleatória, pois o DirectX e o XNA usam essa ordem para determinar a normal da face.

A normal de uma face refere-se ao vetor perpendicular ao plano da mesma. Imagine um quadrado completamente na horizontal. A normal desse quadrado seria um vetor apontando diretamente para cima, formando um ângulo de 90 graus com o plano desse quadrado (que nesse caso seria horizontal).



O XNA determina se a face está voltada para o jogador ou não baseado nessa normal, processo que é chamado de “Culling”. Normalmente tem-se o “Culling” como sendo horário, ou seja, caso os vértices estejam dispostos no sentido horário, a normal será voltada para a câmera e a face será visível ao jogador. Caso os vértices fiquem dispostos no anti-horário, o sentido da normal inverte e a face “inverte” o seu lado, ficando invisível para a câmera. É possível definir as propriedades do “Culling”, fazendo-o exibir os dois lados da face, independente da normal. Isso facilita na hora de criação dos polígonos, mas perde-se muito em desempenho. Por isso vamos manter o “Culling” no sentido horário e apenas tomar cuidado na hora de dispor os vértices.



O primeiro procedimento a ser feito para posicionar os vértices é criar um vetor que possa guardar as informações das coordenadas. Como mencionado anteriormente, existem diversos tipos de vértices, mas vamos escolher um que além de guardar a posição, também armazena a sua cor. Esse tipo é chamado “VertexPositionColor”.

Declare o nosso vetor logo no início da classe do jogo, junto com a declaração do device (o texto em cinza é apenas para situar o local onde foi adicionado o código):

```
GraphicsDeviceManager graphics; //nosso device criado no início
ContentManager content; // não utilizamos ainda
Effect effect; // efeito criado no capitulo anterior
VertexPositionColor[] vertices; // nosso vetor de vértices
```

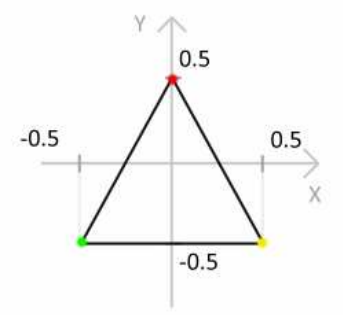
Para não sobrecarregar o método “Initialize”, vamos colocar as informações dos vértices em uma rotina separada e a chamaremos dentro do “Initialize”. O nome da rotina fica a sua escolha (eu utilizei “CarregarVertices”).

```
private void CarregarVertices()
{
      vertices = new VertexPositionColor[3]; // 1

      vertices[0].Position = new Vector3(-0.5f, -0.5f, 0f); // 2
      vertices[0].Color = Color.Green; // 3
      vertices[1].Position = new Vector3(0, 0.5f, 0f);
      vertices[1].Color = Color.Red;
      vertices[2].Position = new Vector3(0.5f, -0.5f, 0f);
      vertices[2].Color = Color.Yellow;
}
```

Segue abaixo a descrição de cada uma das linhas (enumeradas no comentário)

1. Essa linha apenas dimensiona o nosso vetor de vértices para três, pois precisamos de três vértices para desenhar um triângulo.
2. Determinamos a posição do vértice 0 através da propriedade “Position” com um vetor de 3 dimensões (“Vector3”) e os valores “-0.5f, -0.5f, 0” para as coordenadas X, Y e Z, respectivamente.
3. Definimos a cor do vértice 0 através da propriedade “Color”.

Experimente imaginar a disposição desses vértices no espaço ou desenhe-os em uma folha de papel. Fazendo isso você pode visualizar mais facilmente que os vértices estão ordenados no sentido horário.



Com o nosso método pronto, precisamos chamá-lo no “Initialize()”, logo após as propriedades do device que definimos no início do tutorial:

```
graphics.ApplyChanges(); // aplica as mudanças
Window.Title = "Meu jogo"; // define um título à janela
CarregarVertices();
base.Initialize();
```

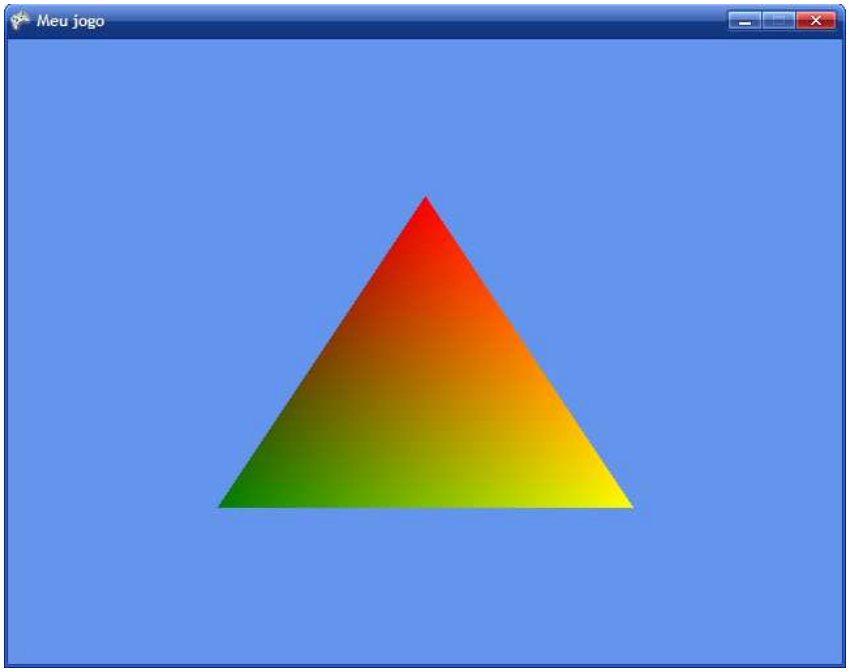
Até o momento, já temos nossos três vértices definidos e carregados, mas ainda não mandamos a engine apresentar esses vértices na tela (renderizar). Para isso, é necessário alterar um pouco o método “Draw()”.

As linhas a seguir se encarregam de renderizar nossos vértices e devem ser colocadas entre o “pass.Begin()” e o “pass.End()”:

```
graphics.GraphicsDevice.VertexDeclaration = new
VertexDeclaration(graphics.GraphicsDevice,
VertexPositionColor.VertexElements); // 1

graphics.GraphicsDevice.DrawUserPrimitives(PrimitiveType.TriangleList,
vertices, 0, 1); // 2
```

1. A linha marcada com 1 corresponde a declaração dos elementos dos vértices que estão prestes a serem renderizados.
2. Essa é a linha que realmente apresenta os vértices na tela. É composta por 4 argumentos:
    a. **primitiveType** – Indica que a disposição dos vértices no vetor. No nosso caso, indica que os vértices formam uma lista de triângulos. Como temos apenas 3 vértices, temos também apenas 1 triângulo
    b. **vertexData** – O vetor contendo os vértices
    c. **vertexOffset** – Indica o primeiro vértice da lista de triângulos. Como nosso vetor tem 3 vértices, variando de 0 até 2, o primeiro vértice a ser apresentado é o vértice 0.
    d. **primitiveCount** – Número de triângulos que ele deve desenhar com base no vetor de vértices. Como temos apenas 3 vértices, temos no máximo 1 triângulo. Caso o nosso vetor possuísse 6 vértices, poderíamos mandar a engine renderizar 2 triângulos.

Com todo esse código criado, podemos finalmente compilar nosso projeto e visualizar o nosso triângulo colorido na tela.



Caso essa imagem não tenha aparecido, verifique se não faltou alguma linha de código, principalmente no método “Draw()”, que é o responsável por mostrar os itens na tela.

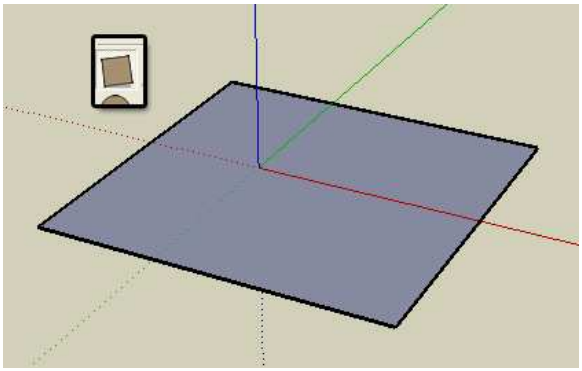
8. CRIANDO E RENDERIZANDO UM SÓLIDO TRIDIMENSIONAL

Vamos partir agora para o mundo 3D de verdade, carregando modelos e texturas criados a parte.

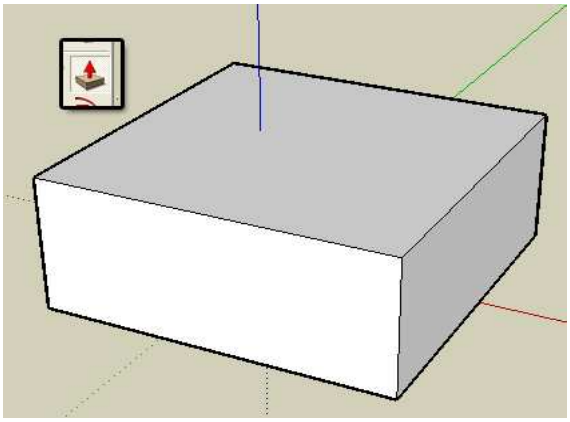
Eu optei por parar de trabalhar no código atual e fazer os próximos capítulos em um novo projeto, para não misturar aquilo que a fizemos até agora com o que está por vir, que é um pouco diferente. Então nesse momento eu recomendo que você salve o projeto atual e crie um completamente novo, repetindo mesmos passos até o capítulo "O mundo 3D". Aquilo que vimos em "Efeitos" e "Desenhando seu primeiro triângulo" nós não vamos mais utilizar.

O primeiro procedimento a ser tomado é criar um simples modelo que possa ser exibido na nossa engine. O formato de objeto que vamos utilizar é o ".X", o padrão utilizado pelo XNA. O aplicativo onde você vai modelar e texturizar o objeto fica a seu critério, desde que seja possível exportar ou converter para um arquivo ".X" posteriormente. Eu particularmente gosto muito do SketchUp, então vou ensinar como criar uma simples caixa com uma textura nesse editor.

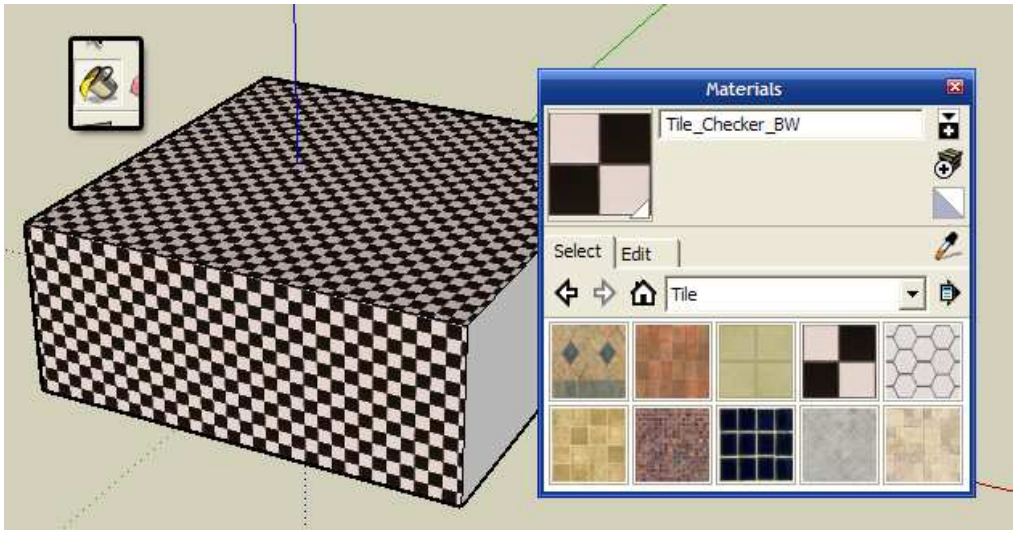
Abra o SketchUp e com a ferramenta "Rectangle" desenhe um retângulo horizontal com as dimensões de 5m x 5m (veja no canto inferior direito).



Agora selecione a ferramenta "Push/Pull", clique sobre o retângulo recém criado, mova o mouse para cima dando volume ao retângulo (transformando-o em um paralelepípedo). Clique novamente para confirmar.

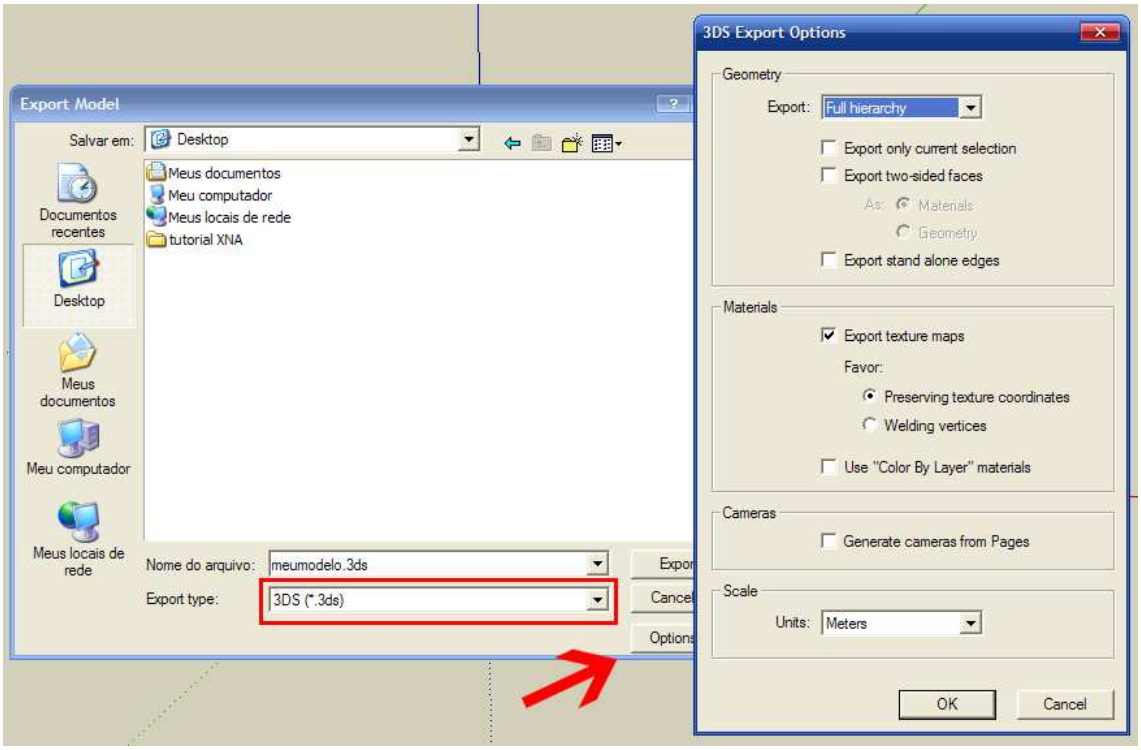


Nosso sólido está pronto, precisamos apenas aplicar uma textura para posteriormente exportar o modelo. Selecione a ferramenta “Paint Bucket” para trazer a janela de materiais. Nessa janela há diversos tipos de texturas, escolha aquela que mais agradar (veja no menu drop-down os diferentes tipos de texturas). Para aplicar, basta selecionar a textura e clicar diretamente nas faces do objeto.



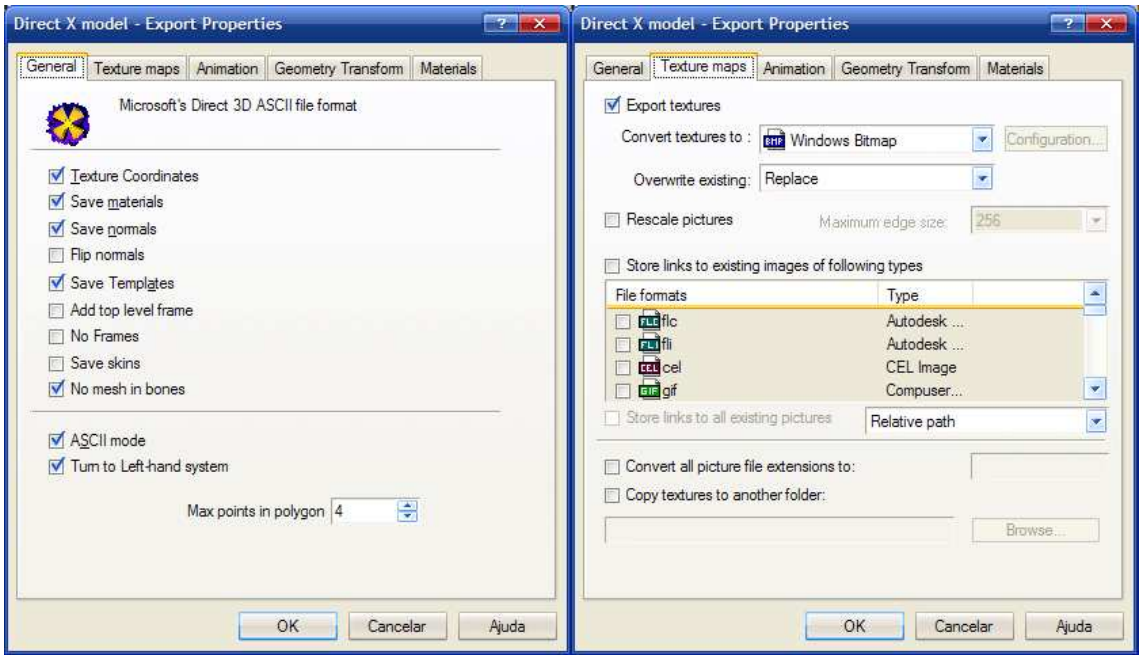
Com o modelo completamente texturizado, vamos exportá-lo para o formato “3DS”, que será posteriormente convertido para “.X” usando o Deep Exploration.

Vá até o menu “File -> Export -> 3D Model”. Na janela de salvar, escolha para exportar como “3DS” e configure a janela “Options” da maneira mostrada pela figura:



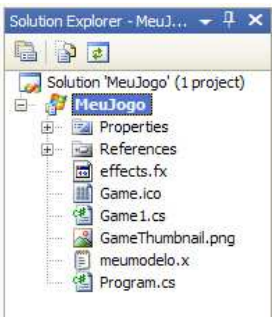
Exporte para um local qualquer, pois o local ainda não tem tanta importância. Após exportar o modelo, foram criados dois arquivos. Um deles é o objeto em si (“3DS”) e o outro é a textura.

Pode fechar o SketchUp, não vamos mais usá-lo. Abra o Deep Exploration e carregue o arquivo 3DS recém exportado através do menu “File -> Open”. Após o objeto ter sido carregado, vá novamente até o menu “File -> Save As...”. Na janela que se abrir, escolha para salvar como tipo “DirectX Model (*.X)” e clique em “Salvar”. Uma outra janela contendo as propriedades do objeto a ser exportado vai aparecer. Configure-a da forma como mostra a figura:



Na aba “Animation”, desmarque a caixa “Export Animation”. Clique em OK.
Se tudo ocorreu bem, um arquivo “.X” foi gerado, junto com um arquivo “BMP”, que corresponde a nossa textura convertida. Copie ambos os arquivos para a pasta raiz do projeto (aquela onde ficam os arquivos de código fonte, efeitos, ícones).

Vamos então voltar ao C#. Na lateral direita, em “Solution Explorer”, clique com o botão direito sobre “MeuJogo”, em seguida “Add -> Existing Item”. Na janela de abrir, selecione “Content Pipeline Files” no campo “Files of Type”. Navegue até o local onde está o arquivo “.X” e clique em “Add”. Seu arquivo “.X” aparece no “Solution Explorer” junto aos outros itens.



Agora que temos o modelo adicionado ao projeto, podemos carregar ele facilmente pelo XNA. No projeto, declare uma variável para o modelo no início da classe (junto a declaração do device).

```
GraphicsDeviceManager graphics; //nosso device criado no início
ContentManager content; // não utilizamos ainda
Model meuModelo; // armazenaremos o modelo aqui
```

Os modelos são carregados sempre na rotina “LoadGraphicsContent()”. Carregá-lo é bem fácil, basta apenas uma linha:

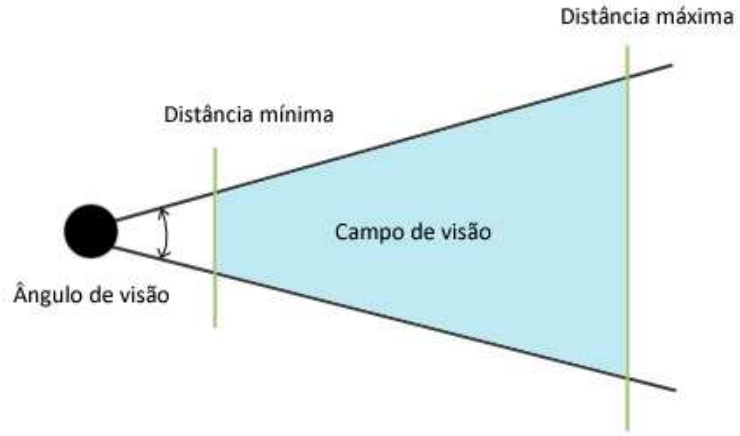
```
protected override void LoadGraphicsContent(bool loadAllContent)
{
      if (loadAllContent)
      {
            meuModelo = content.Load<Model>("meumodelo");
      }
}
```

Como podem notar, bem simples, basta apenas indicar qual o nome do modelo que deseja abrir (sem a extensão “.X”). Com o modelo carregado, precisamos apresentá-lo na tela através do método “Draw()”

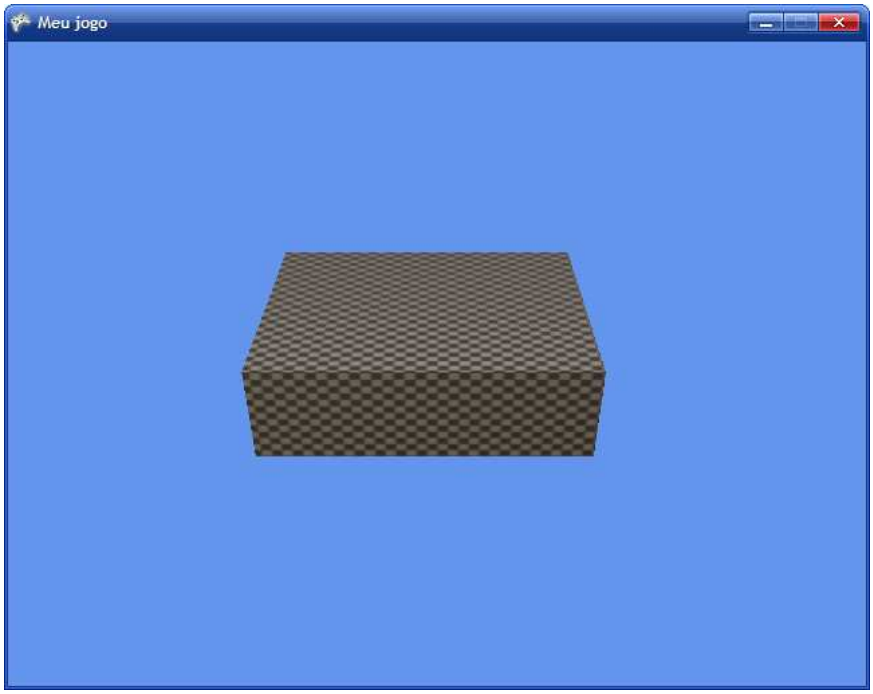
```
protected override void Draw(GameTime gameTime)
{
      graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

      Matrix[] transforms = new Matrix[meuModelo.Bones.Count]; // 1
      meuModelo.CopyAbsoluteBoneTransformsTo(transforms); // 1

      foreach (ModelMesh mesh in myModel.Meshes) // 2
      {
            foreach (BasicEffect effect in mesh.Effects) // 3
            {
                  effect.EnableDefaultLighting();

                  effect.View = Matrix.CreateLookAt(
                  new Vector3(15,10,0), new Vector3(0,0,0),
                  Vector3.Up); // 4

                  effect.Projection =
                  Matrix.CreatePerspectiveFieldOfView(
                  MathHelper.ToRadians(45.0f),
                  graphics.GraphicsDevice.Viewport.Width /
                  graphics.GraphicsDevice.Viewport.Height,
                  1.0f,
                   10000.0f); // 5
            }
            mesh.Draw();
      }
}
```

1. Cria uma matriz de transformação para o modelo, sobre a qual é possível fazer diversas operações algébricas que resultam em uma mudança no estado do objeto (girar, mover, alterar escala).

2. Um loop para varrer todas as “mesh” contidas dentro do modelo. Na maioria dos casos os modelos contam com apenas 1 mesh, mas é sempre bom preparar o código para todas as situações
3. Semelhante ao segundo, mas referente aos efeitos presentes em cada mesh. Novamente, na maioria dos casos cada mesh possui apenas um efeito.
4. Cria a matriz de visualização. O primeiro argumento é o vetor posição da câmera, o segundo é o vetor alvo da câmera e o último argumento é o vetor que determina o sentido vertical do “mundo”.
5. Cria a matriz de projeção em si. Nela você determina o campo de visão, a proporção (normalmente a mesma da janela), a distância mínima que uma face deve estar de você para que ela seja exibida e a distância máxima que face é exibida.



Salve o seu projeto e aperte F5 para iniciar a execução. Após todas essas linhas de código, você deve obter algo semelhante a isso:



9. CONFIGURANDO CÂMERAS

Até o momento, tudo o que fizemos foi mostrado estaticamente na tela, sem ter a possibilidade de movimento de câmeras. Neste capítulo vamos aprender a configurar uma câmera e também como trabalhar com “Inputs” (teclado, mais especificamente).

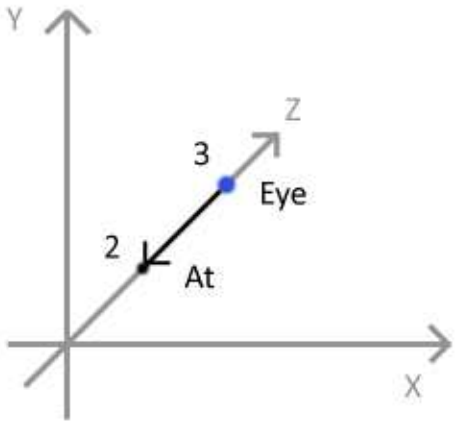
Há muita matemática envolvida, principalmente trigonometria. Como a documentação sobre esse assunto é facilmente encontrada em livros de matemática, não vou explicar os procedimentos exatos que utilizei para fazer a lógica do movimento da câmera.

Uma câmera é formada por dois pontos. Um deles é a posição do observador (“Eye”) e o outro é para onde ele está olhando (“At”). Ambos os pontos são definidos através de um vetor tridimensional (“Vector3”). Além desses dois vetores, precisamos armazenar a angulação vertical e horizontal da câmera através de um “Vector2”.

Declare os três vetores junto com todas as outras declarações, no início da classe:

```
Model meuModelo; // armazenaremos o modelo aqui
Vector3 Eye = new Vector3(0, 0, 3); // posicao da camera
Vector3 At = new Vector3(0, 0, 2); // alvo da camera
Vector2 Angle = new Vector2(0, 0); // angulo da camera
```

Como podem notar, eu posicionei o vetor “Eye” uma unidade atrás do “At” na direção Z, sendo que ambos estão sobre os eixos X e Y (pois ambos são 0).



Tendo os vetores definidos, precisamos alterá-los conforme as teclas pressionadas pelo jogador. Essa verificação é feita dentro do módulo “Update()”. Os cálculos envolvidos aqui podem parecer complexos, mas são apenas relações trigonométricas, facilmente encontradas na internet e em livros. Por esse motivo, não vou explicar passo a passo o que cada linha faz.

```
KeyboardState teclado = Keyboard.GetState();
if (teclado.IsKeyDown(Keys.Up)) {
      if ((Angle.Y + 1) <= 88)
            Angle.Y += 1;
}
if (teclado.IsKeyDown(Keys.Down)) {
      if ((Angle.Y - 1) >= -88)
            Angle.Y -= 1;
}
if (teclado.IsKeyDown(Keys.Right)) {
      Angle.X += 1;
      if ((Angle.X + 1) == 360)
            Angle.X = 0;
}
if (teclado.IsKeyDown(Keys.Left)) {
      Angle.X -= 1;
      if ((Angle.X - 1) == -1)
            Angle.X = 359;
}
if (teclado.IsKeyDown(Keys.W)) {
      Eye.X = At.X + (At.X - Eye.X) * (Single)(1.3);
      Eye.Z = At.Z + (At.Z - Eye.Z) * (Single)(1.3);
      Eye.Y = At.Y + (At.Y - Eye.Y) * (Single)(1.3);
}
if (teclado.IsKeyDown(Keys.S)) {
      Eye.X = At.X - (At.X - Eye.X) * (Single)(2);
      Eye.Z = At.Z - (At.Z - Eye.Z) * (Single)(2);
      Eye.Y = At.Y - (At.Y - Eye.Y) * (Single)(2);
}
if (teclado.IsKeyDown(Keys.D)) {
      Eye.X = Eye.X + (Single)(Math.Cos(((Double)(Angle.X) + 90) *
                  Math.PI / 180)) * (Single)1.3;
      Eye.Z = Eye.Z + (Single)(Math.Sin(((Double)(Angle.X) + 90) *
                  Math.PI / 180)) * (Single)1.3;
}
if (teclado.IsKeyDown(Keys.A)) {
      Eye.X = Eye.X + (Single)(Math.Cos(((Double)(Angle.X) - 90) *
      Math.PI / 180)) *  (Single)1.3;
      Eye.Z = Eye.Z + (Single)(Math.Sin(((Double)(Angle.X) - 90) *
      Math.PI / 180)) *  (Single)1.3;
}

At.X = Eye.X + (Single)(Math.Cos((Double)(Angle.X) * Math.PI / 180) *
            Math.Cos((Double)(Angle.Y) * Math.PI / 180));
At.Z = Eye.Z + (Single)(Math.Sin((Double)(Angle.X) * Math.PI / 180) *
            Math.Cos((Double)(Angle.Y) * Math.PI / 180));
At.Y = Eye.Y + (Single)(Math.Sin((Double)(Angle.Y) * Math.PI / 180));
```

Para fazer a câmera funcionar basta alterar os dois primeiros argumentos da matriz de visualização dentro da rotina “Draw()” para os vetores “Eye” e “At”, pois agora os vetores não possuem valores fixos e a matriz precisa ser atualizada cada vez que a câmera muda de posição.

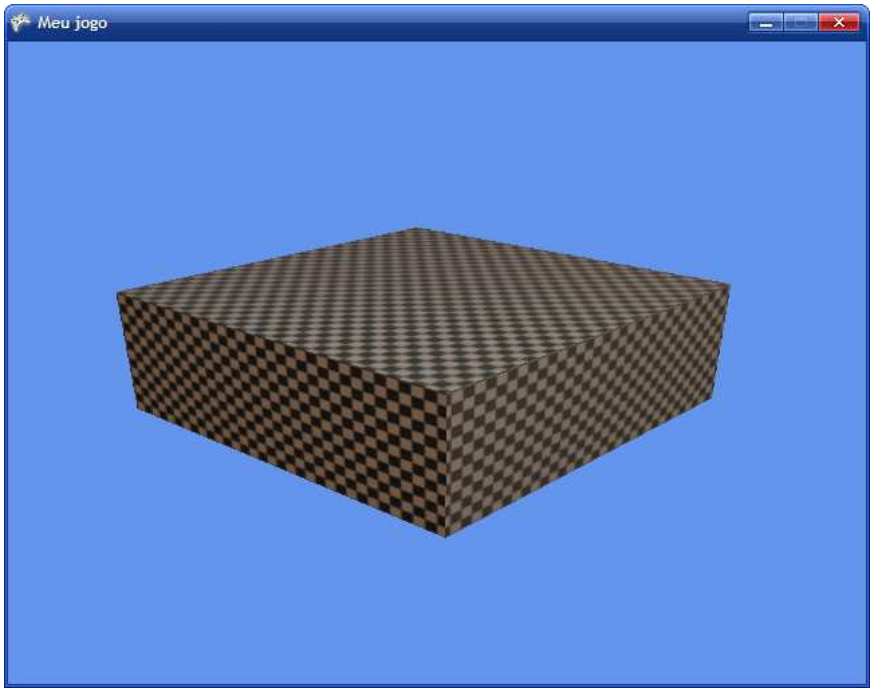
Troque a linha:

```
effect.View = Matrix.CreateLookAt(new Vector3(15,10,0),
            new Vector3(0,0,0), Vector3.Up);
```

Por:

```
effect.View = Matrix.CreateLookAt(Eye, At, Vector3.Up);
```

Pronto, agora temos a nossa câmera funcionando. Rode o jogo e use as teclas WASD em conjunto com as setas para se movimentar no local.



10. MOVENDO OBJETOS

Neste último capítulo vamos ver como se move um modelo pelo espaço. O processo é bem semelhante ao movimento da câmera. Se você não encontrou dificuldades no capítulo anterior, este será bem tranqüilo.

A lógica básica á incrementar as componentes da posição do objeto e aplicar uma matriz de transformação que move o modelo.

Inicialmente, vamos criar um novo “Vector3” (novamente, lá no topo da classe) para guardar a posição do objeto, que se encontra inicialmente na origem do sistema:

```
Vector3 At = new Vector3(0, 0, 2); // alvo da camera
Vector2 Angle = new Vector2(0, 0); // angulo da camera
Vector3 Posicao = new Vector3(0, 0, 0); //posicao do objeto
```

Em seguida, vamos incrementar o valor de suas coordenadas conforme o tempo passa. A melhor forma de fazer isso é incrementando dentro da rotina “Update()” (coloquei logo após o término da verificação do teclado):

```
At.Y = Eye.Y + (Single)(Math.Sin((Double)(Angle.Y) * Math.PI / 180));

Posicao.X += 0.03f;
Posicao.Y += 0.03f;
Posicao.Z += 0.03f;

base.Update(gameTime);
```

Falta somente criar a matriz de transformação, que irá mover o sólido com base nas informações do vetor “Posição”. A matriz fica dentro do método “Draw()”, junto com as outras matrizes de visualização e projeção.

```
effect.World = transforms[mesh.ParentBone.Index] *
Matrix.CreateRotationY((float)Math.PI / 2f) *
Matrix.CreateTranslation(Posicao);
```

Como pode notar, além de transladar o objeto, é possível também fazer ele girar em torno do eixo Y (ou qualquer outro, basta definir). Como nós não estamos incrementando nenhum ângulo, a rotação vai ser nula (pode-se até remover a rotação da transformação).

Feito isso, basta rodar o aplicativo e ver o movimento.

11. CONCLUSÃO

Vamos ficando por aqui, espero que tenham gostado. Tentei deixar o tutorial simples e com o maior número de ilustrações possível, já que é algo que exige muita representação visual. Evitei colocar informações muito complexas e deixei de falar alguns conceitos, pois a minha intenção com este tutorial era criar um guia prático, sem colocar muitas regras, de modo que o leitor possa ler e ao mesmo tempo ir programando, sem ter que quebrar a cabeça para decifrar frases.

Com os assuntos tratados neste guia já é possível fazer jogos, ainda que simples, em três dimensões, já que muitos deles se resumem a um cenário e objetos se mexendo na tela, faltando apenas programar a lógica do jogo.

Em caso de dificuldade para realizar alguma operação ou programar algo que não foi explicado neste tutorial, recomendo consultar o próprio Help do XNA Game Studio ("Help -> Contents"), que possui uma vasta gama de artigos, tutoriais, referências e exemplos de código.

Abraços e até a próxima!

Guia Prático para Iniciantes – Microsoft® XNA
Escrito: Fernando Birck (Fergo) – http://www.fergonez.net
Julho 2007

Referências:

- MSDN
  - http://msdn2.microsoft.com/en-us/xna/default.aspx
- Riemer's
  - http://www.riemers.net/