Desenvolvimento de Jogos Utilizando XNA: um Exemplo com o Jogo SpaceX

Joni Pereira de Pinho Rodrigues da Silva
Universidade Severino Sombra, CECETEN,
Curso de Sistemas de Informação
jonipinho@gmail.com

Janaína Veiga
Universidade Severino Sombra, CECETEN,
Curso de Sistemas de Informação
janainavcarvalho@gmail.com

Carlos Vitor de Alencar Carvalho
Universidade Severino Sombra, Centro Universitário de Volta Redonda,Centro Universitário Geraldo di Biase e FAETEC-IST,
cvitorc@gmail.com

Resumo: *A área de jogos eletrônicos tem crescido cada vez mais. Pode-se, nos dias atuais, encontrar jogos em diversos dispositivos: computadores pessoais, consoles domésticos e dispositivos móveis. O desenvolvimento de um jogo não é uma tarefa fácil, pois envolve diversas áreas de conhecimento e também diversas subáreas da computação. A área de desenvolvimento de jogos desperta muita curiosidade e interesse em muitos programadores, entretanto a maioria das instituições de ensino superior com cursos na área de computação não apresentam o foco ou disciplinas específicas para o desenvolvimento de jogos, principalmente em Curso de Sistemas de Informação. Esta foi uma das motivações para o desenvolvimento deste trabalho. Este artigo apresenta o framework XNA e as suas principais funcionalidades para desenvolvimento de jogos eletrônicos interativos. Como resultado e exemplificação o artigo também apresenta o desenvolvimento de um jogo 2D, chamado SpaceX.*

Palavras-chave: *Framework XNA. Jogos Eletrônicos.*

Game Development Using XNA

Abstract: *The gaming area is growing, being found today in various devices: personal computers, consoles and mobile devices. The development of a game is not an easy task because it involves several areas of knowledge and also several subareas of computing. The game development area awakens a lot of curiosity and interest in many programmers, but most institutions of higher education courses in computing do not focus on specific courses about game development, especially in Information Systems Course. This was a motivation for the development of this work. This paper presents the XNA framework and its main features for the development of interactive electronic games. As result and exemplifying, the article also presents the development of a 2D game, called SpaceX.*

Keywords: *XNA Framework. Electronic Games.*

Introdução

O avanço tecnológico e a expansão da informação disponível têm cooperado com a evolução no desenvolvimento de jogos. A área de jogos eletrônicos está crescendo cada vez mais. Pode-se encontrar nos dias atuais em diversos dispositivos: computadores pessoais, consoles domésticos e dispositivos móveis.

Os jogos eletrônicos têm diversos gêneros: estratégia, simulação, aventura, esporte, RPG, passatempo, educação, entre outros. Devido à sua grande capacidade de alcançar públicos diversificados, os jogos tornam-se em grande instrumento de entretenimento, o que proporciona horas de diversão e desafio.

O desenvolvimento de jogos não é uma tarefa trivial: requer atenção especial e diferenciada, pois é um *software* completo que abrange as mais diversas áreas da computação como programação, design, redes, computação gráfica, inteligência artificial, sons entre outras. Contudo, novas ferramentas e métodos de desenvolvimento de jogos têm facilitado seu desenvolvimento, por exemplo, as ferramentas *Ogre3D*, *Unity* e *XNA*.

Segundo Perucia (2005), o desenvolvimento de um projeto de jogo segue as regras e parâmetros de desenvolvimento de um jogo qualquer, entretanto, a grande diferença está na fase inicial de criatividade. O ciclo de desenvolvimento de um jogo é dividido em algumas etapas como: *Brainstorming*, *Game Design*, *Design Document* (DD), *Level Design* e Criação e Desenvolvimento. É na fase de *brainstorming* que a ideia inicial do jogo é proposta. Nesta fase nenhuma ideia pode ser descartada.

A área de desenvolvimento de jogos traz muitas curiosidades e desperta interesse em muitos programadores. Este artigo tem por objetivo apresentar um resumo do *framework* XNA, com mostras de suas principais funcionalidades para desenvolvimento de jogos eletrônicos interativos, para despertar nos amantes de jogos a curiosidade de aprofundar seus conhecimentos na construção de jogos, por *hobby* ou profissionalmente.

Na segunda seção são apresentados o *framework* XNA e as principais informações de sua estrutura para desenvolvimento de jogos. Na terceira seção é feita uma introdução para desenvolver o jogo bidimensional SpaceX com utilização de *Microsoft* Visual C# *Express* e *XNA*. Na seção quatro são apresentadas as considerações finais sobre o trabalho e o *framework XNA*.

Conhecendo o *Framework XNA*

Microsoft XNA (sigla em inglês que significa *XNA's Not Acronymed*) é um *framework* desenvolvido pela *Microsoft* em 2006 para auxiliar os desenvolvedores no processo de criação de jogos para *Windows* (PC), *Xbox* 360 (Video *game*) e *Zune* (Media *Player* Portátil).

Segundo Cawood (2007) o *XNA* se tornou um importante avanço na programação de jogos. Antes do *XNA* era muito complicado e caro para um estudante, ou desenvolvedor independente de jogos, ter acesso a um kit de desenvolvimento de console. O *XNA* utiliza a linguagem de programação C# bastante similar com as linguagens de programação C/C++ e Java, o que diminui a curva de aprendizado para desenvolvedores dessas linguagens.

Segundo Lobão (2010) o *XNA* é uma extensão do *Microsoft* Visual C# *Express* que, integrados, criam um ambiente para o desenvolvimento de jogos. Observe na Figura 1, como basicamente o XNA se organiza:

A Figura 1 apresenta as tecnologias presentes na plataforma. Na primeira camada, de cima para baixo, se posiciona o ambiente de desenvolvimento, o *Microsoft Visual C# Express*, uma IDE gratuita de desenvolvimento, que utiliza a linguagem de programação C# disponível para *download*, gratuitamente.

Figura 1. Organização do *XNA* (retirado de Lobão (2010), p. 4)

Na segunda camada está o *framework XNA*, responsável por simplificar o desenvolvimento de jogos tanto para *Windows* quanto *XBOX* 360 e *Zune*. A utilização de um *framework* faz com que o desenvolvedor foque na lógica do jogo e tire a responsabilidade de escrever linhas de código que serão executadas em nível de *hardware*. Por exemplo, quando desenvolve jogos sem o apoio de um *framework*, com linguagens de programação como C/C++, muitas vezes o desenvolvedor tem de se preocupar em implementar técnicas de otimização de desempenho, de gerenciamento de memória, renderização de gráficos entre outras, que não fazem parte do projeto em si. Segundo Lobão (2010), o *XNA* é composto por quatro camadas: jogos, extensões, núcleo e plataforma (Figura 2).

Na Figura 2 é importante o entendimento dos componentes: o Modelo de Aplicação e a *Content Pipeline*.

O Modelo de Aplicação é o responsável por criar e gerenciar as janelas do jogo e por executar a inicialização do *DirectX*. Além disso, é o responsável por gerenciar o *loop* da execução do jogo (Figura 3).

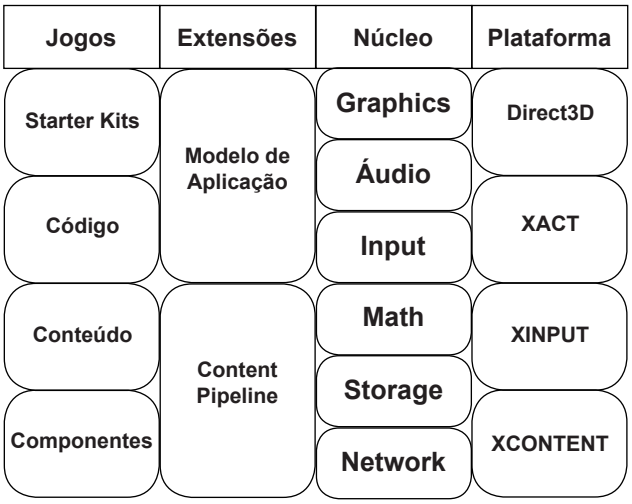

Figura 2. Camadas do *Framework XNA* (adaptado de Lobão (2010), p. 5).

O método *Initialize* é chamado apenas uma vez, quando o método *Run*() (que inicia o *loop* do jogo) é executado. O método *LoadContent*() é responsável por carregar os componentes gráficos e audiovisuais, como modelos, texturas, músicas etc. O método *Update* é responsável por atualizar o estado dos objetos presentes no jogo. O método *Draw* tem como objetivo gerenciar todo o conteúdo gráfico que será desenhado.

O método *UnloadContent* chama o *garbage collector* para forçar que os objetos não mais em uso sejam despejados.

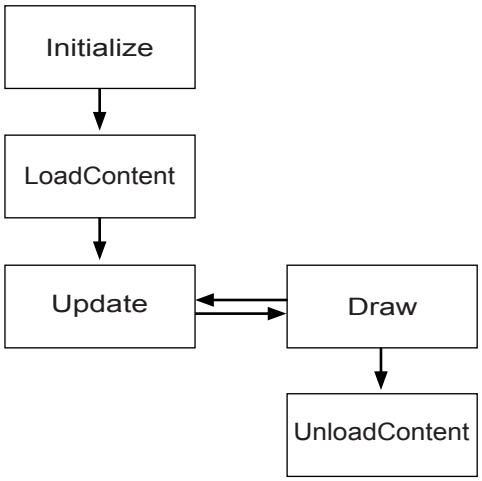

Figura 3. Ciclo de execução de um jogo

O *Content Pipeline* é o componente que fornece as ferramentas para processar todo o conteúdo que fará parte do jogo. Segundo Lobão (2010), a *Content Pipeline* organiza o processamento do conteúdo de forma a simplificar seu tratamento pelo jogo. Compreende um número de passos que incluem importadores ("*importers*") que leem o conteúdo da forma original e geram um formato intermediário, bem determinado: processadores ("*processors*") que geram conteúdo no formato pré-processado, pronto para uso. E, finalmente, o gerenciador de conteúdo ("*content manager*"), em tempo de execução atende às requisições do programa (Figura 4).

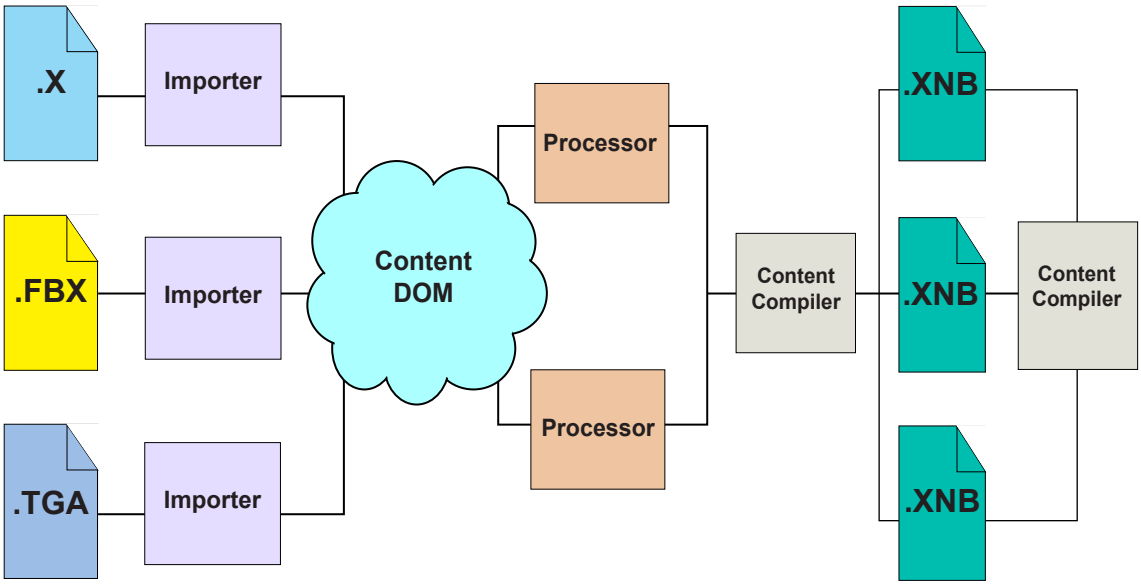

Figura 4. Content Pipeline do XNA (retirado de Lobão(2010), p. 21).

Basicamente, o que ele faz é importar os conteúdos, processá-los, com o compilador de conteúdo, e gerar um arquivo de conteúdo que será utilizado pelo jogo. A Camada Núcleo é a principal do *XNA Framework*. Os seus componentes fornecem recursos para as mais diversas ações do jogo. Em *Graphics* ficam as classes de auxílio para lidar com a forma de como o Jogo carrega os *sprites*, tamanho de tela, se deve ser executado em tela cheia ou numa janela etc. O componente *Graphics* ainda oferece o recurso *BasicEffects*, que facilita a apresentação de objetos 3D na tela, e o *SpriteBatch*, para a manipulação de gráficos 2D e partículas.

Em Áudio constam as classes que lidam com as músicas e efeitos sonoros do jogo digital. Possui uma série de funcionalidades para gerenciar a forma como o som atua dentro do jogo. O componente de *Input* é responsável por toda a obtenção de entrada de dados, fornecida pelo usuário no jogo com os periféricos de entrada de dados, como mouse, teclado etc. O componente *Math* oferece uma vasta gama de cálculos e funções matemáticas pré-definidas para se trabalhar com colisão, movimentação, física e definição de vetores, matrizes, planos, esferas e tudo o mais relacionado à matemática. Já o componente *Storage* é utilizado para armazenar dados.

O componente *Network* tem como objetivo tornar mais fácil o desenvolvimento do jogo para o ambiente *on-line*, e oferece maneiras simples de conectar o seu jogo entre *XBOX* 360 e *PC*, e conexões locais em uma *LAN*. Na camada plataforma estão os componentes *Direct3D*, responsável pela parte gráfica; *XACT*, responsável pelo áudio; *XINPUT*, responsável pela entrada e saída de periféricos (teclado, mouse e outros) e *XContent*, responsável pela utilização de objetos importados de outras ferramentas.

Desenvolvendo o *SpaceX*

Esta seção tem por objetivo demonstrar o desenvolvimento de um jogo em 2D. Entretanto, antes de iniciar o desenvolvimento de qualquer jogo é necessário fazer seu planejamento. Segundo Lobão (2010), muitos projetos falham por causa do pouco esforço nessa fase, o que conduz a projetos sem um fim definido, que nunca terminam ou que são finalizados, mas não alcançam seus objetivos.

Abordaremos o processo de criação do jogo *SpaceX* de forma simplificada, mediante apresentação das classes e métodos criadas para o jogo.

Planejando o Jogo

Com o crescimento da indústria de jogos eletrônicos, apareceram vários tipos de jogos que podem ser dividido em gêneros diferentes. Segundo Rollings & Morris (2000), podemos destacar os seguintes gêneros: aventura, ação, estratégia, quebra-cabeça, simuladores, brinquedo e educacional.

O jogo apresentado neste artigo se enquadra na modalidade simulador (Joni, 2012). A ideia do jogo é criar uma nave controlada pelo jogador que pode se mover livremente ao redor da tela. O jogador deverá enfrentar uma chuva de meteoros que aparecem no topo da tela de forma randômica, e que ao passar alguns segundos, adiciona novos meteoros na tela. O jogador é capaz de disparar tiros para destruir os meteoros, porém apenas um tiro por vez, ou seja, enquanto o tiro não estiver fora dos limites da tela ou acertar um meteoro o jogador não pode atirar novamente. O jogador ganhará dez pontos por meteoro destruído. A colisão do meteoro com a nave finaliza o jogo, e aparece uma tela que indica esse fim.

Geração das Classes e Criação de Métodos

Para iniciar o processo de desenvolvimento, foi criado um novo projeto no *Microsoft Visual C# Espress Edition* 2008 com o nome *SpaceX*. Na pasta *Content do Solution Explorer* foram adicionados algumas imagens e sons que serão utilizadas no desenvolvimento do jogo: *Background.png* (tela de fundo do jogo), *Meteoro.png* (textura do meteoro), *Nave.png* (textura da nave), *Tiros.png* (textura do tiro), musica.mp3 (música que vai tocar durante a execução do jogo), *explosion.waw* (música que será tocada quando um meteoro colidir com um tiro ou a nave com o meteoro) e uma fonte.

No desenvolvimento do jogo foram criadas na classe *Game1.cs* algumas variáveis para armazenar: a textura do fundo do jogo, o tamanho da tela, a música de fundo do jogo, o

som da colisão, o controle do fim do jogo, os tiros disparados pelo jogador, uma lista para armazenar os tiros, a textura do tiro, uma variável booleana para controlar apenas um tiro por vez pelo jogador entre outras.

No método *LoadContent* serão carregadas algumas variáveis como: a variável *background*, com a imagem que foi utilizada para o fundo do jogo; a variável *TamanhoTela*, com as coordenadas do tamanho da tela; a variável com a música que está na pasta *Content* e os parâmetros para que música toque até o jogo terminar. Este método foi utilizado para carregar valores referentes às classe Nave que foram: a textura que a nave vai possuir, a posição inicial da nave na tela e o tamanho da imagem. No método *LoadContent* foi carregado ainda o som da explosão quando acontecer a colisão entre o jogador e o meteoro. Para a classe Tiro, foi carregada no método *LoadContent* a textura do tiro. No método *LoadContent* foi carregada também a variável fonte, adicionada no projeto para o placar do jogo. O método *Draw* foi utilizado para desenhar todas as imagens do jogo.

A nave é um dos principais personagens do jogo e será comandada pelo jogador. Para inserir a nave no jogo foi criada uma nova classe com o nome Nave que terá algumas propriedades para armazenar textura, posição, tamanho e velocidade. Na classe *Game1* foi necessário fazer alguns ajustes para que a nave apareça no jogo. Portanto, foi criado um novo objeto do tipo Nave com o nome jogador no inicio da classe *Game1*. Como definido no planejamento do jogo, a nave será controlada pelo jogador. Para capturar a entrada de dados pelo teclado foi necessário escrever um código que perceba que a posição do jogador no jogo vai receber alguns valores (X,Y) dependendo da tecla apertada pelo usuário. Existe uma restrição que controla o movimento para que a nave não saia da tela do jogo . Por exemplo, caso seja apertada a tecla *Up* do teclado a posição y do objeto vai diminuir e, consequentemente, o objeto vai subir e o mesmo acontecerá com as outras teclas, de acordo com o valor que está proposto entre parênteses, no método.

Para inserir os asteroides no jogo foi criada uma nova classe com o nome Asteroides com algumas propriedades para armazenar valores como a textura que o asteroide vai possuir, posição inicial e a velocidade em que vai se movimentar na tela. Na classe *Game1* foi criada uma lista para armazenar os meteoros e uma variável para armazenar a textura do meteoro. Para que os meteoros sejam adicionados durante um determinado tempo foi criada uma variável com o nome tempo para armazenar o tempo do jogo, e um gerador de números randômicos para que o meteoro receba esses valores como parâmetro de velocidade na classe *Game1*.

No método *Update* foi utilizada a variável tempo que foi criada para armazenar o tempo do jogo e foram criados os meteoros a partir deste tempo determinado. O primeiro ponto a destacar é o parâmetro do tipo *gameTime* recebido pela variável tempo. Este parâmetro é crucial para a lógica do jogo, uma vez que o jogador deve saber quanto tempo se passou desde o inicio da sua execução. A variável tempo armazena o valor do tempo do jogo com o objetivo de criar um novo meteoro a cada 1 segundo. Após criar um meteoro a variável tempo recebe o valor 0 e só cria outro meteoro após 1 segundo. Quando um meteoro é adicionado recebe valores como textura, posição, velocidades X e Y e é adicionado em uma lista que armazena os meteoros criados. Para adicionar movimento ao meteoro basta apenas incrementar o valor da posição do meteoro no eixo X e Y com a velocidade. No método *Draw,* os meteoros foram desenhados utilizando um contador para percorrer a lista que contem os meteoros.

Para adicionar movimento ao meteoro basta apenas incrementar o valor da posição do meteoro no eixo X e Y com a velocidade. Para que o código ficasse mais legível foi criado um método na classe *Game1* depois do método *Draw* com o nome MovimentandoAsteroides. Este método recebe a posição inicial dos eixo X e Y do meteoro e soma aos valores da variável velocidade no eixo X e Y que são gerados randomicamente quando o meteoro é criado. Para finalizar, foi chamado este método dentro do método *Update*.

Ao adicionar movimento aos meteoros foi possível perceber que seus valores são atualizados constantemente, e com o resultado saem da tela, porém não do jogo. Uma forma de resolver este problema foi criar um método que verifique se o meteoro está dentro dos limites da tela. Caso não esteja, recebe novos valores que o repõem novamente no jogo. Foi criado um método com o nome *ColocarNaPosicaoInicial* depois do método *Draw*. Este método verifica se a posição do meteoro está ultrapassando os limites da tela. Caso esteja, passa uma nova posição para o meteoro na tela do jogo. Para finalizar, foi chamado este método dentro do método *Update*.

Na classe Nave, abaixo do método *Draw*, foi criado um método com o nome Colide (Figura 5) que verifica se as posições das coordenadas X e Y do objeto estão dentro do segundo objeto, ou seja, deve-se verificar se os valores de X e Y do objeto que queremos testar é menor ou igual aos valores de X e Y do outro objeto .

```
        public bool Colide(Asteroides outraSprite)
          {
              if (this.posicao.X + textura.Width > outraSprite.posicao.X &&
this.posicao.X < outraSprite.posicao.X + outraSprite.textura.Width &&
this.posicao.Y + textura.Height > outraSprite.posicao.Y && this.posicao.Y <
outraSprite.posicao.Y + outraSprite.textura.Height)
            return true;
               else
            return false;
            }
```

Figura 5. Método Colide

Na classe *Game1*, abaixo do método *Draw*, foi criado um método com o nome VerificaColisao (Figura 6) com a função de verificar se há colisão entre a nave do jogador e algum meteoro. Caso isso aconteça, vai tocar o som da explosão, a variável fim do jogo vai receber valor verdadeiro, a música de fundo do jogo vai parar e os meteoros serão removidos do jogo. Dentro do método *Update* da classe *Game1* será chamado o método que foi criado para verificar se há colisão entre o jogador e algum meteoro.

```
//verificando a colisao
        public void VerificaColisao()
        {
            for (int i = 0; i < meteorolist.Count; i++)
            {
                if (jogador.Colide(meteorolist[i]))
                {

                    explosao.Play();
                    FimdoJogo = true;
                    MediaPlayer.Stop();
                    for (i = 0; i < meteorolist.Count; i++)
                    {
                        meteorolist.RemoveAt(i);
                    }

                }

            }
        }
```

Figura 6. Método VerificaColisao.

Para que a nave dispare tiros, foi criada uma nova classe no projeto com o nome Tiros e com algumas propriedades para armazenar: textura, posição inicial, velocidade e um método para verificar a colisão entre o tiro e os meteoros. No método *Update* foi criada uma estrutura para que quando o jogador aperte a tecla-espaço do teclado o tiro seja disparado. Após a criação deste método, o tiro já está sendo disparado pelo jogador, porém não está se movimentando no jogo, para adicionar movimento ao tiro é preciso incrementar o valor da posição tiro no eixo X e Y com o valor da velocidade. Para isso foi criado um método na classe *Game1* depois do método *Draw* com o nome MovimentandoTiros (Figura 7). No método *Update* foi chamado o método MovimentandoTiros.

```
        //criando o movimento dos tiros
        protected void MovimentandoTiros()
        {
            for (int i = 0; i < tiroslist.Count; i++)
            {

                tiroslist[i].posicao.X += tiroslist[i].velocidadeX;
                tiroslist[i].posicao.Y -= tiroslist[i].velocidadeY;

            }

        }
```

Figura 7. Método para movimentar o tiro

Para remover os tiros foi criado um método na classe *Game1*, abaixo do método *Draw*, com o nome RemoveTiros para que fique verificando se o tiro saiu da tela do jogo. Caso isso aconteça o tiro será removido.

Na classe *Game1* foi criada uma variável do tipo inteiro para somar os pontos quando o tiro acertar os meteoros e abaixo do método *Draw* foi criado um método que verifique se há a colisão entre o tiro e o meteoro (Figura 8).

```
public void VerificaColisaoTiros()
{
    for (int i = 0; i < meteorolist.Count; i++)
    {
        for (int j=0; j <tiroslist.Count; j++)
        {

            if (tiroslist[j].ColideTiros(meteorolist[i]))
            {

             explosao.Play();
             tiroslist.RemoveAt(j);
             meteorolist.RemoveAt(i);
             apertouTiro = true;
             pontos += 10;


            }

        }

    }
}
```

Figura 8. Método que verifica colisão entre tiros e meteoros

O método VerificaColisaoTiros (Figura 8) está verificando se algum tiro colidiu com algum meteoro. Caso isso aconteça, são removidos o tiro e meteoro. A função apertouTiro receberá um valor verdadeiro permitindo que o jogador dispare um novo tiro e a variável pontos recebe dez pontos. No método *Update* foi chamado este método.

Para adicionar um placar foi necessário adicionar uma fonte no projeto. Depois de adicionada a fonte no projeto, na classe *Game1* foi criada uma variável do tipo *SpriteFont.*

Para finalizar o jogo, foi utilizada a variável booleana para controlar o jogo, tornando esta tarefa bem fácil. A variável que controla o fim do jogo recebe o valor verdadeiro quando há uma colisão entre o jogador e o meteoro. Deste modo, foi criada uma estrutura no inicio do método *Update* de forma que todos os métodos fiquem dentro dessa estrutura e o jogo só ira continuar se o valor dessa variável for falso. Depois de criar essa estrutura, no método *Draw* da classe *Game1* foi desenhada uma tela especificando que o jogo acabou e informando como iniciar um novo jogo.

Para que o jogador possa jogar novamente é necessário que a variável que controla o fim do jogo receba o valor verdadeiro, então foi criada uma estrutura no método *Update* da classe *Game1* de forma que quando o jogador apertar a tecla Enter do teclado essa variável receba o valor verdadeiro.

Quando o jogador apertar a tecla Enter do teclado várias funções estão recebendo novos valores para iniciar um novo jogo. Por fim, compilando o projeto, aparece o resultado final do desenvolvimento do jogo *SpaceX* (Figura 9).

Figura 9. Interface final do Jogo *SpaceX*

Considerações Finais

O objetivo do trabalho foi apresentar, de forma sucinta, alguns dos principais conceitos envolvidos na implementação de jogos por computador utilizando o *framework XNA*. Como proposto no início do artigo foi abordado o desenvolvimento de um jogo, de forma sucinta, do início ao fim utilizando o *framework XNA* que resultou em um jogo que foi chamado de *SpaceX*.

Consideramos que o *XNA* demonstrou ser uma ferramenta robusta para o desenvolvimento de jogos, a qual pode ser utilizada por desenvolvedores iniciantes como forma de se inserirem no mercado de desenvolvimento nacional, porém cabe ressaltar que a área de desenvolvimento de jogos é muito extensa e este trabalho apenas apontou alguns conceitos básicos. Neste contexto, consideramo-nos satisfeitos se o artigo contribuir para a diminuição de dúvidas nesta área, bem como para o crescimento de trabalhos que explorem, de forma direta ou indireta, o tema.

Referências

Cawood, S.; Mcgree, P. (2007) Microsoft XNA Game Studio Creator's Guide. New York: Osborne. 456 p.

Lobão, A. S. et. al. (2010) XNA 3.0 para Desenvolvimento de Jogos no Windows, Zune e Xbox 360, Rio de Janeiro: Brasport, 431 p.

Perucia, A. et. al. (2005) Desenvolvimento de Jogos Eletrônicos. Porto Alegre: Novatec, 320 p.

Rollings, A.; Morris, D.(2000) Game Architecture and Design, Arizona: Coriolis, 742 p.

Silva, J. P. de P. R. da. (2012) Desenvolvimento de Jogos utilizando XNA. Monografia de Graduação do curso de Sistemas de Informação. Universidade Severino Sombra, 63 p.