CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA CELSO SUCKOW DA FONSECA



RELATÓRIO DO PROJETO FINAL

CURSO TÉCNICO DE INFORMÁTICA

PROF. EDUARDO BEZERRA

ALUNOS: BRUNO FERRAZ DE ALMEIDA COUTO

ESTEVAN GLADSTONE NASCIMENTO

LUAN CHRISTIAN DE FREITAS BORGES

Rio de Janeiro, 6 de Junho de 2014

CEFET/RJ - Técnico de Informática

6AINFO 2014.1

Sumário

[Objetivo 2](#_Toc389689257)

[Introdução 2](#_Toc389689258)

[Primeiros estudos - A linguagem C# e o XNA 3](#_Toc389689259)

[Passo 1 – Primeiros contatos com a Física 3](#_Toc389689260)

[Passo 2 – Entendendo o 3D e movimentação 4](#_Toc389689261)

[Passo 3 – O conceito dos *Heightmaps* 5](#_Toc389689262)

[Passo 4 – Colisões 7](#_Toc389689263)

[Passo 5 – Mapa e Placar 10](#_Toc389689264)

[Conclusão 11](#_Toc389689265)

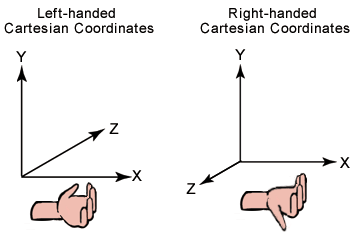
[Referência Bibliográfica 11](#_Toc389689266)

### Objetivo

Nosso projeto busca explorar de forma mais simplificada, mediante os prazos e a nossa inexperiência com o material com o qual trabalhamos, o desenvolvimento de jogos. Com a base do jogo produzida no semestre anterior, traçamos como meta levar o projeto iniciado a um patamar de aplicativo, já com feições de um *game*, com inicio e fim definidos. Além disso, implementar a utilização do acessório *Kinect,* da Microsoft, como alternativa de jogabilidade, tornando-se também um estudo sobre novas tecnologias para desenvolvimento.

### Introdução

No inicio do semestre, quando optamos por este projeto, sabíamos que o mesmo já havia sido iniciado, e o planejado seria dar continuidade a base que nos foi entregue. Tínhamos em mãos um jogo aparentemente simples, mas que por trás já haviam sido programadas funções com uma certa complexidade e dificuldade. Consistia basicamente em uma nave dentro de uma caixa, com movimentação plana e limitada pelas paredes, e com a rotação da própria nave, que mesmo girando sob os três eixos dimensionais (X, Y, Z, representados na imagem abaixo), sua movimentação só era influenciada em X e Z.

[](#_top)

Figura

O jogo foi feito com C#, uma linguagem própria da Microsoft e que está associada à plataforma “.NET” da mesma empresa. No C#, foi desenvolvido um *framework* que reunisse recursos pré-estabelecidos os quais facilitassem o desenvolvimento de jogos. Dentro de uma programação orientada a objetos (POO), oferece classes e métodos já definidos para os diversos usos na programação dos jogos e seus princípios lógicos.

Diário de Produção

### Primeiros estudos - A linguagem C# e o XNA

O primeiro contato com o código foi impactante para nós visto a pouca experiência com programação orientada a objetos. Entretanto, com a pequena experiência com o *Java*, conforme fomos seguindo o estudo das bases da linguagem da Microsoft, através do suporte oferecido aos desenvolvedores pelo site [*www.msdn.com*](http://www.msdn.com), percebemos a similaridade.

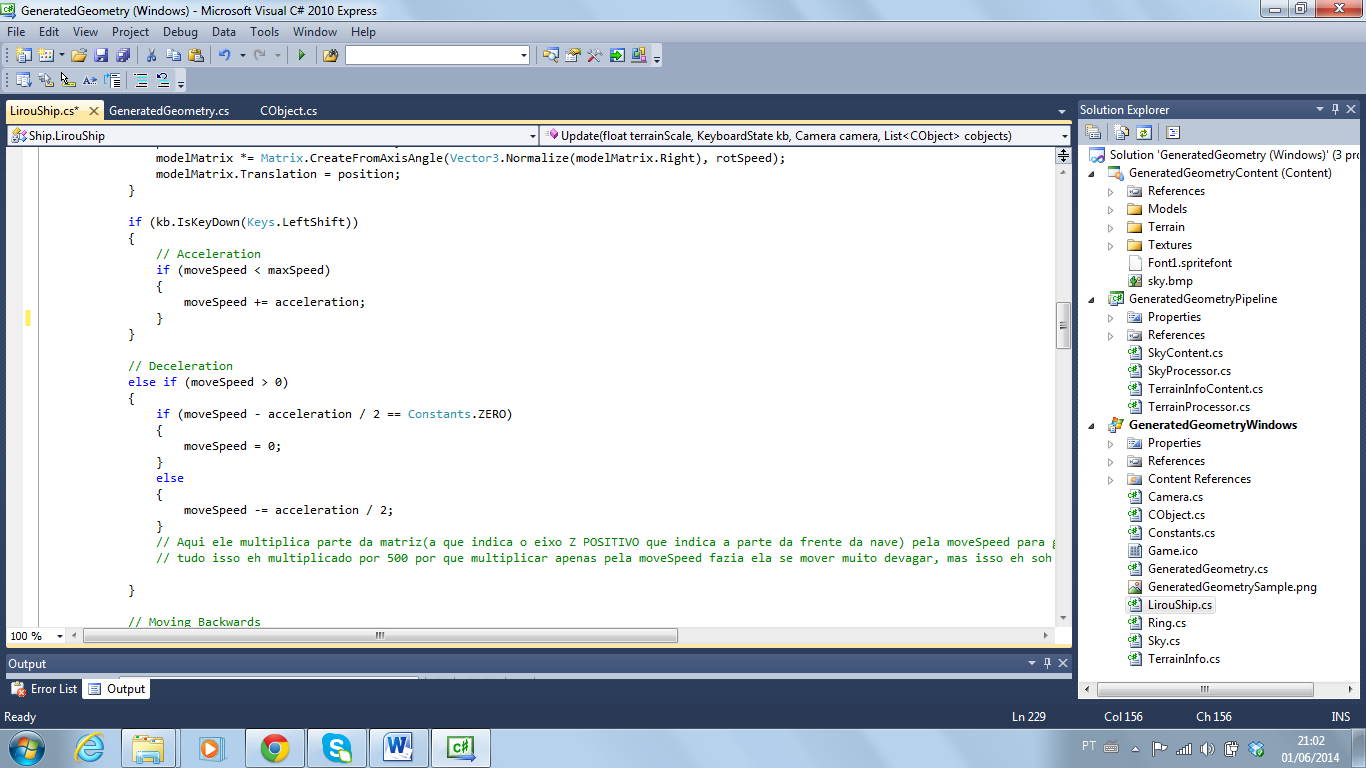
Não há tanto material que difundido que aborde o C# para iniciantes. Muitos fóruns, e outros meios de discussão na Internet acabam retratando tópicos isolados, e mais complexos, dificultando esse início. Mas nesse suporte, o leitor dispõe de uma boa introdução à linguagem, construída sob um escopo dividido entre os princípios da .NET framework, do editor do C#, o Visual Studio (o XNA só funciona na versão 2010) e alguns conceitos do próprio C# como uma linguagem de programação, abordando recursos e características próprios (como estruturas de *delegates* e *namespaces* ou outros já difundidos na programação (a exemplo da manipulação de matrizes ou explicação de eventos, comum em POOs).

E o *XNA* segue sob a mesma tutela, com artigos variados sobre sua biblioteca e recursos. O *framework*, pela popularidade, acaba sendo bem introduzido e explorado por vários materiais, inclusive tutoriais e vídeo-aulas do próprio *Microsoft Developer Network* (MSDN). Mas em um geral, em meios como Youtube, e sites de desenvolvedores experientes , como “ELEMAR DEV”*,* um diário de produção e exemplo de iniciativa brasileira para difusão da programação, introduz-se bem esse *framework*, e conseguimos ser bem guiados através das peculiaridades da ferramenta.

Dessa forma, essa biblioteca da Microsoft, mesmo para quem não tenha muito contato com a programação, é uma introdução suficiente para dar-se os primeiros passos na área, e justamente por essa facilidade, nos ajudou a entender melhor com o que iríamos trabalhar. E já com uma pequena noção, com um pouco de esforço, encontra-se bons matérias que ajudem no progresso para o desenvolvimento. Além disso, nosso uso de *“C”* nos semestres anteriores também ajudou ao entendimento, tido que também tem influências sobre o *C#*.

### Passo 1 – Primeiros contatos com a Física

Nosso primeiro desenvolvimento sobre o jogo que tínhamos em mãos, fora aperfeiçoar a parte “física”. Em nível de manipulação do C#, é algo relativamente simples, visto que demanda mais pela lógica e conceitos básicos de programação que já tínhamos desde o “*C*”. As linhas de código, por exemplo, poderiam ser facilmente adaptadas a um programa deste, visto a similaridade estrutural e por não usar princípios da linguagem orientada a objetos



Figura

*O mesmo se repete para caso “K” esteja pressionado. Mas no caso, se isso ocorrer, a nave diminuirá sua velocidade, passando por zero e tornando-se negativa, simulando a “ré”.*

Primeiramente, concebemos uma aceleração à nave. A instância da classe “*LirouShip”* possui uma variável para a velocidade, e outra para aceleração. Esta permanece constante no decorrer do programa, já “*moveSpeed”* é alterada pela aceleração e os botões “*LeftShift”* e “*K”,* que guiam para frente e para trás, respectivamente, a nave segundo sua orientação no mapa. E se não forem pressionados, a aceleração atua como uma desaceleração sobre a velocidade, diminuindo-a ou aumentando-a, dependendo da perspectiva do movimento. Além disso, vale ressaltar que a o movimento só é realizado depois de testadas todas as condições sobre os botões, evitando a repetição desnecessária de código. Assim, simplificadamente, cada condição altera variáveis, e essas comporão o cálculo que gerará a movimentação no espaço.

### Passo 2 – Entendendo o 3D e movimentação

No início, entender o funcionamento do espaço tridimensional foi um pouco complicado. Apesar de ser bem intuitivo, conceber os cálculos para a programação mostrou-se realmente diferente. A movimentação nos três eixos é bastante similar a bidimensional, porém, quando é necessário o uso das rotações mostra-se bem mais complexa. Primeiro, foi necessário entender que as rotações que ocorrem nos eixos não ocorrem sobre eles, mas sim ao redor, o grande problema era realizar as várias rotações e ainda conseguir movimentar o objeto.

Ao continuar o desenvolvimento do projeto, um dos objetivos era habilitar o movimento nos três eixos, X ,Y e Z, e para isso também mostrou-se necessária a rotação nesses três eixos, porém, a forma como o projeto anterior foi desenvolvido, usando ângulos de Euler e armazenamento em um vetor de 3 posições, o que causava em determinado momento um problema chamado *Gimbal Lock.* O Gimbal Lock consiste na perda da liberdade em um dos eixos de rotação (os chamados *Gimbal axis*) quando dois deles se sobrepõe e não se separam após isso, limitando a rotação para apenas dois eixos.

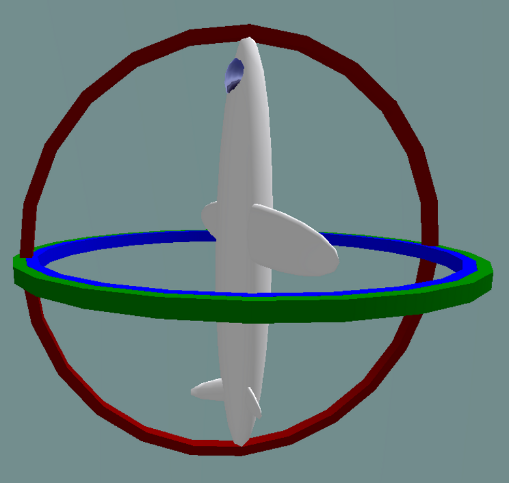
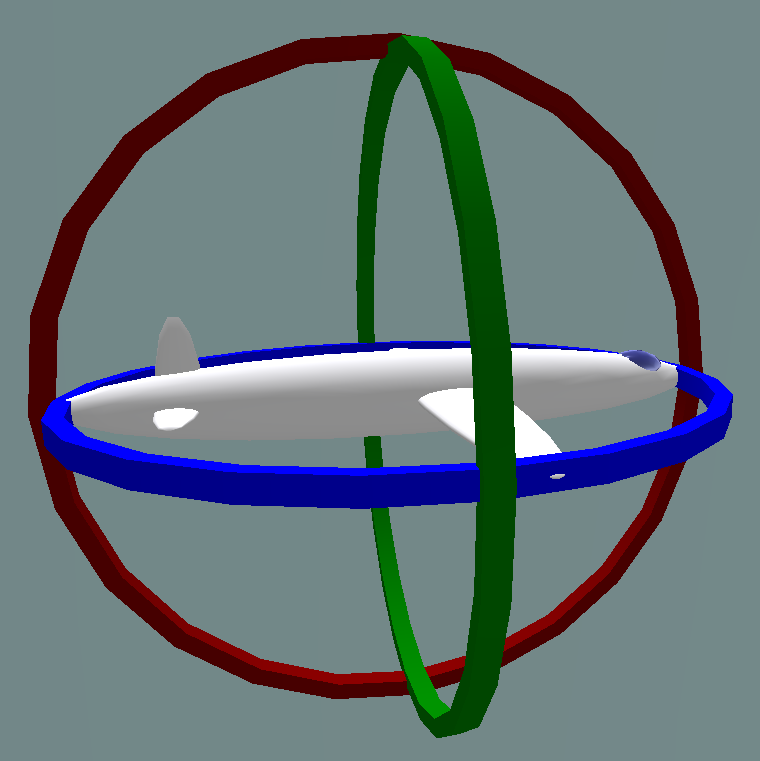
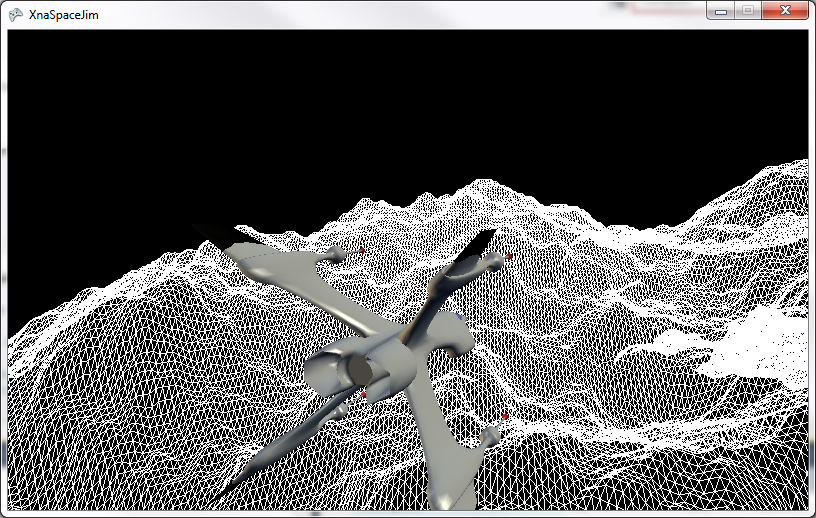


Figura 3

Após estudar o problema e entendê-lo, encontramos duas maneiras de resolução: o uso de Quatérnions e o uso de Matrizes rotacionais. Quatérnions apesar de mais efetivos que as matrizes rotacionais eram mais complexos. O uso de matrizes rotacionais consiste no armazenamento dos ângulos em uma matriz, onde os ângulos não são armazenados em sua forma simples, mas sim os valores de operações feitas com seu seno e cosseno correspondentes. A opção escolhida foi o uso de matrizes rotacionais por já estarem presentes no código, porém não eram usadas em todo seu potencial. Na própria plataforma do XNA já havia o suporte ao uso de matrizes rotacionais e é um padrão bem difundido, o que facilitou e muito o uso.

### Passo 3 – O conceito dos *Heightmaps*



Figura

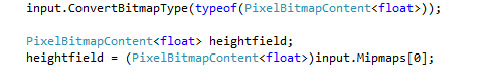
Disponível em: <http://elemarjr.net/2011/08/04/vamos-aprender-xna-parte-12-criando-terrenos-com-heightmaps/>

Pensando em evoluir o *game*, abandonar o cubo que limitava a nave fora nossa próxima meta. Apesar de ao final percebermos a simplicidade, pensar sobre o terreno que comporia o jogo fora um transtorno. Havia formas já padronizadas, normalmente sujeitas a um desenvolvimento pelos próprios *Models*, como a nave e outros objetos poderiam ser feitos, com *meshes* e *bones,* elementos da interface 3D. Mas em maioria, demandava-se uma complexidade que em nossa experiência de um mês ainda não nos cabia. Por fim, ao longo das pesquisas, encontramos uma maneira um pouco mais simples de gerar um terreno: os *heightmaps*.

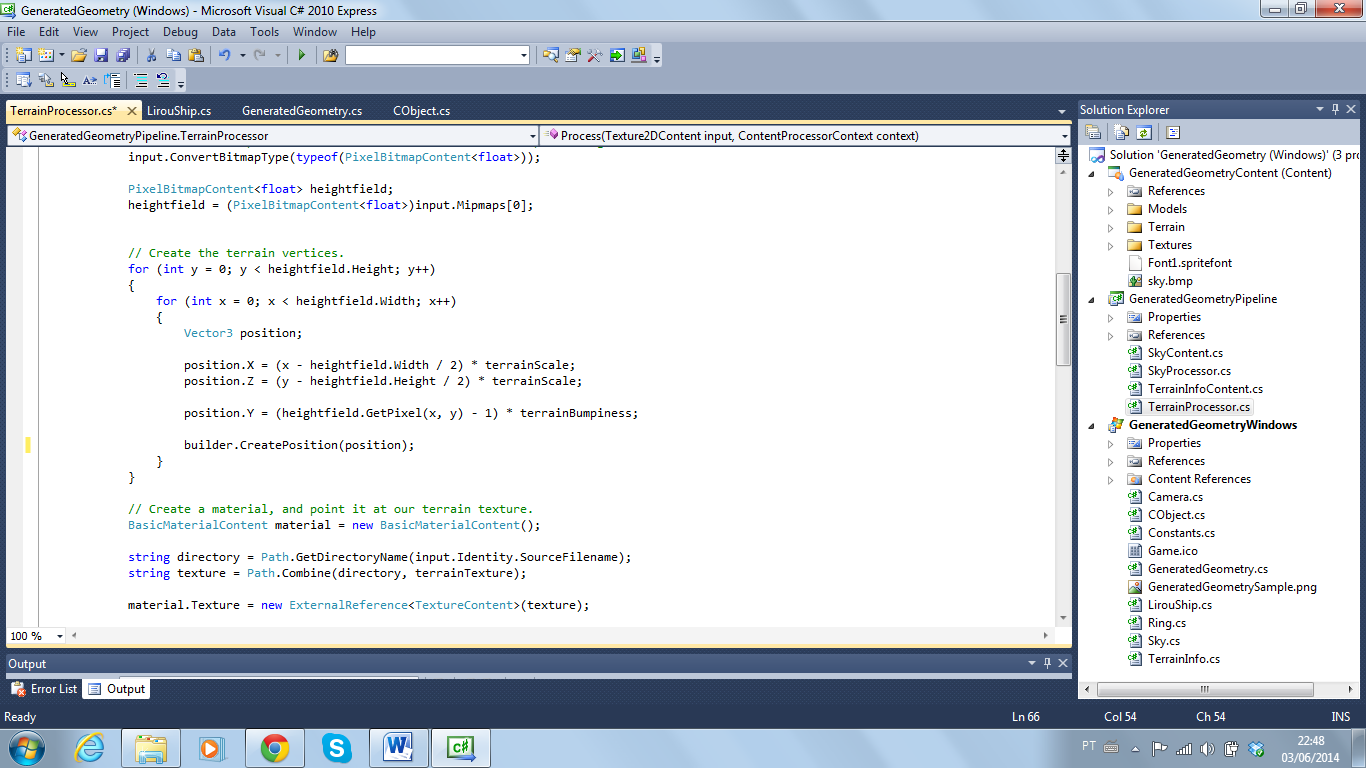
Basicamente, tem-se uma textura que varia em escalas de Cinza, indo do Preto ao Branco, e essa variação quantificamos e transportamos para as alturas do terreno. Em nosso caso, quanto mais próximo do branco, maior a altitude.

Dentro da classe *TerrainProcessor*, como o próprio nome sugere, processa-se o terreno e suas informações para que possa ser desenhado corretamente depois. O primeiro passo, é gerar-se os vértices que orientaram o desenho da textura (como em um poliedro, onde as faces preenchem os espaços entre vértices). Na mesma classe, existem variáveis que participam da transição do mapa monocromático para o 3D. São elas: *terrainScale*, que define a escala do terreno em geral, sobre os planos X,Z (siga a orientação da figura 1 ); *terrainBumpiness*, que define a escala das alturas, sobre o plano Y e *textCoordScale*, que definem o valor para cada unidade da textura em pixels para o modelo do terreno. Logo depois, *heightfield* é gerada e assume numericamente, os valores da imagem em níveis de cinza que modulariza o terreno.

A variável *heightfield* recebendo os valores da imagem.

**

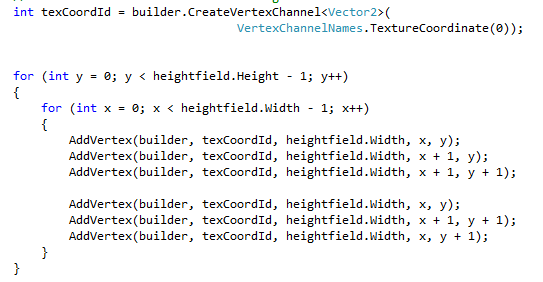
Figura



Dentro dos *rolls*, geram-se os vértices que desenham o escopo do modelo.

Figura

No 3D, um modelo assim feito, é composto por um conjunto de polígonos que desenham-no. Logo, depois dos vértices, é preciso montar-se os triângulos, correspondidos por todos esses pontos gerados através da imagem e os cálculos com as variáveis a fim de associar-se assim a textura “florest.jpg”, que dá a ideia de uma grande floresta encapando nossos morros.



Figura

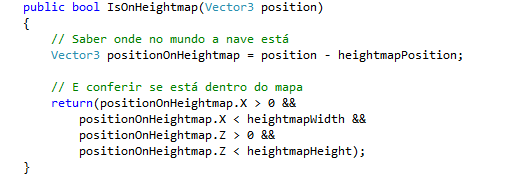
Por fim, a fim de permitirmos uma interação do mapa com o jogo, duas classes mais se correspondem com *TerrainProcessor*. E uma delas, *TerrainInfo*, permitirá a colisão da nave com o “chão”, como está descrito no próximo passo, e a determinação do fim do cenário, motivando o reinício da posição para o centro do terreno.

### Passo 4 – Colisões

As colisões, sob a perspectiva do grupo, foram o grande desafio do projeto, e talvez, o ápice. A complexidade subiu um pouco, e a dificuldade, tratando-se de duas colisões diferentes (uma quanto o mapa, e outra quanto aos próprios objetos), aumentou um pouco.

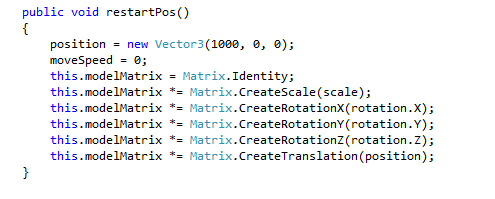
Passo 4.1 – Colisão do Terreno

Apesar de ser gerado como um modelo, com *heightmaps* não é inviável manipular-se as *boundingSpheres*, que é o conceito usado para melhor detectar-se as colisões entre objetos 3D. Para contornar esse problema, foi preciso realizar uma verificação manual, e um tanto quanto robusta, para simular a colisão. E por isso, criar-se *TerrainInfo* é trivial para essa verificação. Nesta, o método *IsOnHeightMap()*, informa se determinada instância que esteja no “Mundo” encontra-se dentro dos limites do terreno, e associado a outra verificação da classe *LirouShip*, impede que se navegue fora dessas fronteiras.



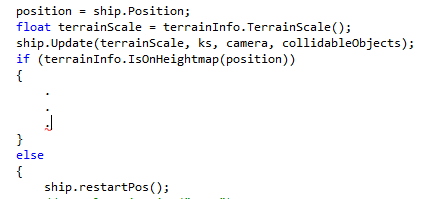
*TerrainInfo*

Figura



*LirouShip*

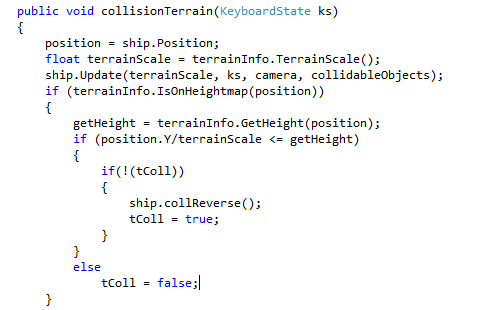
Figura



*GeneratedGeometry*

Figura

Já o método *GetHeight()* é o trecho responsável pela detecção da colisão. Tendo como parâmetro uma posição sob três dimensões, ele retorna a altura relativa do terreno naquela mesma coordenada. Dessa forma, como *TerrainInfo* está instanciada no programa principal, podemos sempre requisitar no decorrer do jogo este método e fazer a comparação com a nave, “gerando” assim uma colisão.



Note que há uma variável *tColl* que é declarada como do tipo *bool* e valor *false*, antes de *collisionTerrain.* Sua função, nada mais é que regular a colisão e impedir que a nave “grude” no terreno. Com ela, só realiza-se a inversão do movimento, que simula o impacto, quando a mesma está como *false*. Assim, evita-se que, quando ainda estiver havendo o movimento reverso, acabe novamente acontecendo uma “colisão”, gerando um *bug.*

Figura

Passo 4.1 – Colisão entre objetos

Depois de concluirmos a interação da nave com o terreno, passamos para uma etapa que nos rendeu bastante tempo, de forma que não está perfeitamente funcionando até o presente momento. A dupla que iniciou este projeto já havia dado passos frente a colisão entre objetos. E com o material que eles nos apresentaram, onde a colisão acontecia entre a nave e as paredes do cubo onde ela estava inserida, preferimos “começar do zero” a tentar reutilizá-lo para nosso propósito.

No MSDN, encontramos uma seção que introduzia bem o sistema de colisão mais padronizado no XNA e o conceito por trás dele: as *BoundingSpheres*. Essas esferas são usadas como delimitadoras de um campo sobre o objeto referido, e através de suas propriedades, são manipuladas para acompanhar e melhor restringir esse volume, para que a colisão seja a mais perfeita possível.

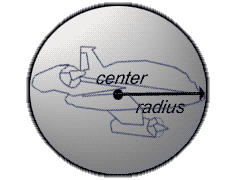


Figura BoundingSphere única de modelo

Visando tornar a colisão mais precisa, especificamos que a as *boundingSpheres* estariam sobre cada *mesh*, unidades que compõem o modelo 3D. Dessa forma, cobrimos de forma mais minuciosa a superfície do objeto, e polindo assim, a colisão entre objetos.

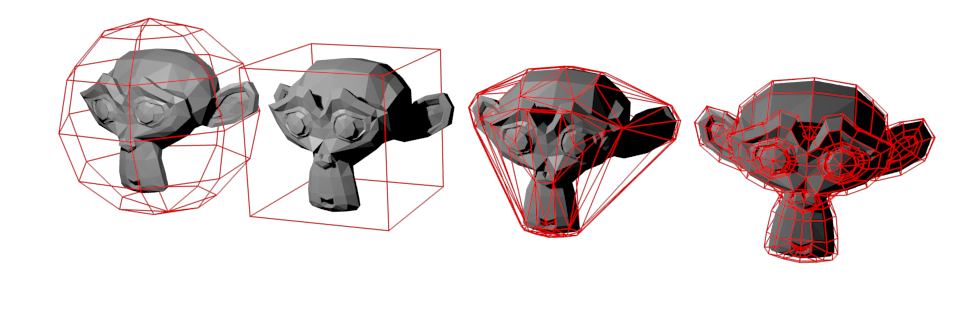


Figura – Diferença de volume abrangido

Nas *boundingSpheres* usamos o atributo *.Radius* para especificar melhor o raio de cada esfera para cada modelo. Depois, é só usar o atributo *.Intersect*, que verifica se duas esferas estão se intercedendo, o que nada mais é do que a própria colisão.

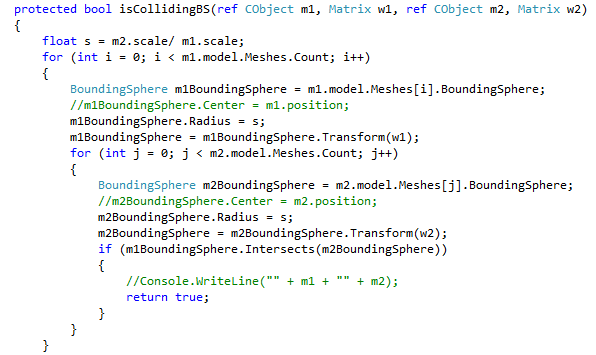


Figura Método de detecção de colisão

Apesar de funcionar, infelizmente esse sistema inteiro não funciona corretamente quando usamos modelos mais simples, como anéis e esferas, mas funciona com perfeição quando objetos iguais. Pesquisamos bastante, o que consumiu muitos dias, mas os padrões de colisão seguiam o que nós elaboramos, ou eram ao menos correspondentes. E como o prazo começou a se aproximar, preferimos deixar os modelos que funcionavam a exemplo da colisão, e caso tivéssemos tempo, apurássemos depois o caso.

### Passo 5 – Mapa e Placar

Nosso último feito no projeto foi um tanto simples, e que exigiu somente uma manipulação do XNA sob algumas técnicas matemáticas. O *framework* disponibiliza uma classe de ferramentas conhecida como *spriteBatch*. Basicamente, elas funcionam para o desenho 2D na tela, seja de animações com *Sprites* (sequência de imagens que geram uma animação) a placares, contadores, e afins. Nesse caso, desenhar o placar foi bem simples, já que antes de implantá-lo, já usávamos uma variável de teste, que escrevia no *console* a pontuação referida às colisões entre objetos. Dessa forma, o placar apenas escreve esse total da variável *points*.

Já o mapa demandou um pouco mais de nosso empenho. Tendo conhecimento da ferramenta de desenho de retângulos pelo *spriteBatch*, utilizamos de uma técnica para fazer um de fundo “transparente”, com somente as bordas destacadas. E utilizando um pouco de uma experiência que tivemos durante o Curso de Informática, em PhP, quando desenhamos gráficos, e para isso precisávamos transpor os números para uma unidade gráfica, pudemos construir um mapa. Sob essa mesma lógica, com os números de posição no mapa e as dimensões do mapa, usamos um cálculo simples para representar com pontos, os objetos do mapa, e com as bordas do quadrado, os limites do mapa. Com frequente atualização do método *Draw,* simula-se também um mapa em tempo real.

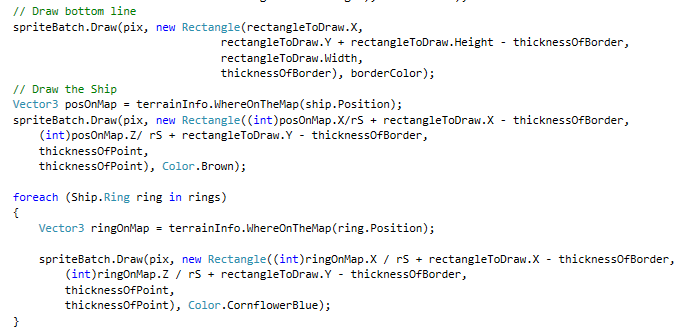


Figura \*rS Define a unidade de medida do retângulo

Apesar da simplicidade, ao longo da implementação, surgiu um problema quanto a ordem em que deveriam ser desenhados os componentes da tela. E assim, por sugestão dos antecessores do projeto, seguimos um padrão de segregar o desenho do céu, ou a *skyBox,* e o resto do mundo. Dessa maneira, respeitamos os “planos” de cada elemento na tela, e contornamos os problemas que apareciam.

### Conclusão

Após dois meses e meio, pesquisando e tendo novas ideias, que às vezes levariam o jogo a outro patamar, e outras, precisaria que o mesmo fosse refeito, apareceu-nos um grande obstáculo. Quando nossos planos já tinham sido alcançados e já estávamos encaminhando para dar passos à frente do *Kinect*, o tempo mostrou-se muito curto. Nosso prazo estava determinado para praticamente duas semanas, e mediante ao curto tempo, aliado às tarefas além do projeto, preferimos polir o projeto e deixar o Sensor de Movimentos a ser trabalhado futuramente, caso um grupo queira dar continuidade. Agora, no fim, percebemos que um jogo custa tempo, dedicação e planejamento, servindo de experiência para futuros projetos.

Tivemos a experiência de aplicar conhecimentos que muitas vezes não nos cativavam justamente por não demonstrarem-se interessantes, como foi o caso das *Matrizes*, fundamentais para a base do nosso jogo. Tendo de olhar para programação sob outra perspectiva, sentimos que nosso conhecimento foi aprimorado.

Fazendo um levantamento sobre o final do projeto, e consequentemente o final do curso, notamos o impacto daquele, já que nossa experiência trazia a programação em uma área voltada para prestação de serviços, onde manipulávamos informações, organizávamos paginas de internet ou gerenciávamos bancos de dados. Mas o desenvolvimento de jogos visa mais uma proposta de entretenimento. E o nosso projeto de jogo, objetivou um entretenimento com bases de simulação. Dessa forma, aplicando sobre a programação conceitos que absorvemos em outras disciplinas, pudemos participar de um desenvolvimento que “abriu horizontes” sobre o que essa ciência da Informática pode abranger.

### Referência Bibliográfica

MICROSOFT DEVELOPER NETWORK. “Guia de Programação *C#”. Disponivel em: <* http://msdn.microsoft.com/pt-br/library/67ef8sbd.aspx >

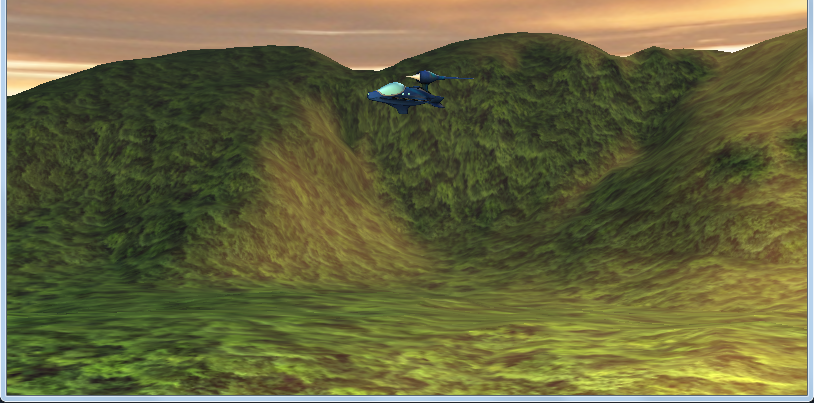
MICROSOFT DEVELOPER NETWORK. “*Bounding Volumes and Collision”.* Disponivel em: < http://msdn.microsoft.com/en-us/library/bb313876.aspx >

MICROSOFT DEVELOPER NETWORK. “*Testing for Collisions”*. Disponivel em: < http://msdn.microsoft.com/en-us/library/bb203906.aspx >

DANTAS, Cleber “*Série aprenda C# - Introdução a plataforma .NET e ao C#*” Disponivel em: < http://www.linhadecodigo.com.br/artigo/984/serie-aprenda-csharp-introducao-a-plataforma-net-e-ao-csharp.aspx >

HAZEN, Steve. *Matrix Basics. “How to step away from storing an orientation as ’3 angles’ “ Disponível em:* <http://stevehazen.wordpress.com/2010/02/15/matrix-basics-how-to-step-away-from-storing-an-orientation-as-3-angles/>

JR, Elemar. “*Vamos Aprender XNA”. Disponivel em : <* http://elemarjr.net >

**