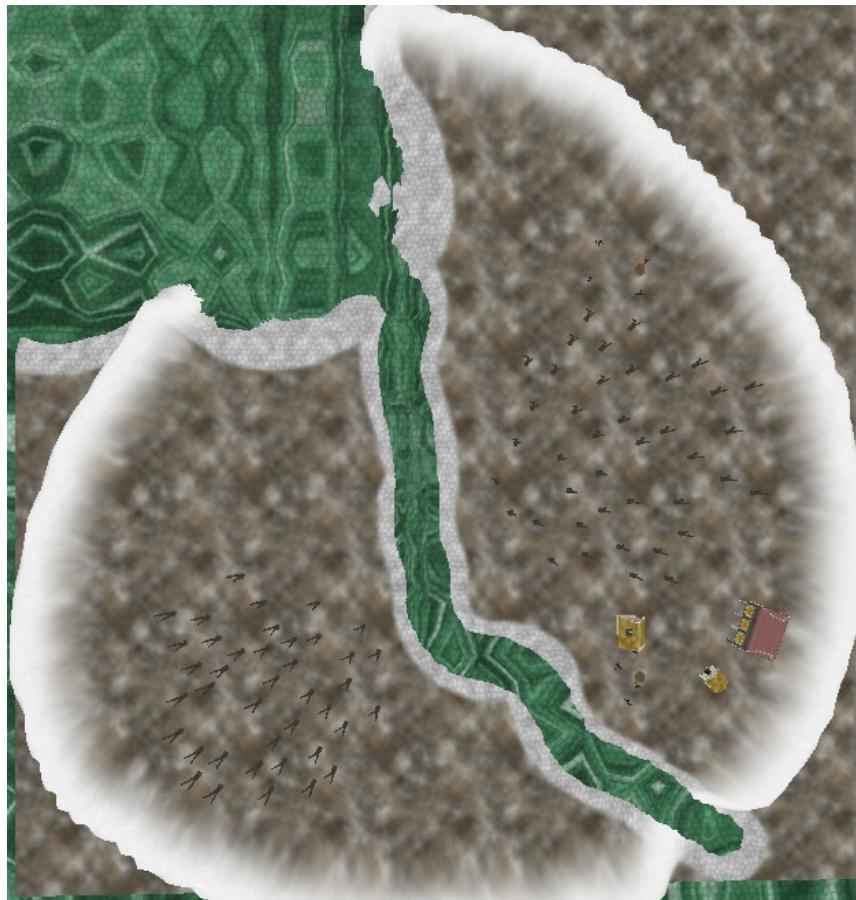


THE FESTIVAL OF PINCER VALLEY

Michael Ferreira da Costa



Prova de Aptidão Profissional
Escola Técnica Empresarial do Oeste
Curso Técnico de Multimédia
Professor Acompanhante – João Caldas Lopes
2012 / 2013

THE FESTIVAL OF PINCER VALLEY

Michael Ferreira da Costa

Prova de Aptidão Profissional apresentada à ETEO como parte dos requisitos para a
obtenção do Curso Multimédia

Prova de Aptidão Profissional

Escola Técnica Empresarial do Oeste

Curso Técnico de Multimédia

Professor Acompanhante – João Caldas Lopes

2012 / 2013

Índice

Introdução.....	5
Finalidade.....	5
Destinatários.....	5
A Calendarização.....	6
Recursos Materiais.....	6
Code::Blocks IDE.....	6
Irrlicht Game Engine.....	7
Bullet Physics Engine.....	7
3D Studio Max	7
Adobe Photoshop CS3	8
fragMotion.....	8
Roadkill 3D	8
Desenvolvimento.....	9
O Conceito.....	9
Modelação de Objetos 3D.....	11
Modeling.....	11
Rigging.....	13
Skinning.....	14
Animation.....	15
Texturing.....	16
Programação do Jogo.....	19
O que é a Programação e o Programador.....	19
Programação em Contexto da PAP.....	20
O Mapa.....	20
O Jogador.....	20
O Monstro.....	21
Avaliação.....	24
Conclusão.....	25
Anexo 01.....	26
Anexo 02.....	33

Introdução

O projeto que escolhi para ser a minha PAP é um video jogo em 3D para o PC. O jogo ocorre durante a época medieval numa aldeia abandonada, bem como no espaço em redor, situada num vale chamada Pincer (garra de caranguejo) devido à sua forma. O jogador é um guerreiro enviado a Pincer Valley para ajudar os aldeões a reunir os recursos com o objetivo de construir fogo-de-artificio para o festival que irá começar num futuro próximo. Após chegar ao vale, o jogador descobre que a aldeia foi abandonada e o vale está ocupado por monstros. O herói assume a responsabilidade de destruir os monstros, desvendar o mistério por detrás dos aldeões desaparecidos e cumprir o seu objetivo inicial: obter recursos para construir o fogo-de-artificio.

Finalidade

A finalidade principal deste projeto sempre foi produzir algo que ainda não tenha sido feito pelos anteriores alunos de multimédia. Optei também por este projeto porque a programação é uma área que me fascina e o desenvolvimento de jogos é uma das minhas metas. Portanto, decidi desenvolver um jogo porque, mesmo querendo entrar nesta área, não tinha qualquer experiência e utilizei esta oportunidade como um catalisador para o meu crescimento como programador e criador de jogos.

Destinatários

O público alvo do meu jogo são jovens adultos, preferencialmente do sexo masculino, com interesse por jogos casuais.

A Calendarização

Tarefas	2012				2013						
	<u>Set</u>	<u>Out</u>	<u>Nov</u>	<u>Dez</u>	<u>Jan</u>	<u>Fev</u>	<u>Mar</u>	<u>Abr</u>	<u>Mai</u>	<u>Jun</u>	<u>Jul</u>
Rigging, skinning, texturizar e animar a personagem (jogador)	X		X	X	X						
Modelar, texturizar, rigging, skinning e animar o monstro			X	X	X	X	X				
Modelar e texturizar os edifícios	X	X	X								
Modelar e texturizar o mapa / terreno		X	X	X							
Modelar e texturizar objectos de cenários		X	X	X	X	X	X	X			
Programação do jogo				X	X	X	X	X	X		
Debug e corrigir erros									X	X	X
Defesa pública da PAP											X
Entrega do pré-projeto			X								
Entrega dos relatórios parcelares					X		X		X		
Entrega dos relatórios finais e da PAP											X
Redação do projeto da PAP							X	X	X		
Apresentação da PAP na escola									X		

Recursos Materiais

Passo a listar todos os recursos de software que foram utilizados no desenvolvimento deste projeto, com uma breve explicação sobre cada um.

Code::Blocks IDE

Este software é um Ambiente Integrado de Desenvolvimento, designado em inglês como Integrated Development Environment (IDE). Tal significa que este programa reúne todas as funcionalidades básicas necessárias para que um programador consiga desenvolver

software, assim como: editor de código, compilador, ligador (linker), depurador (debugger), gerador de modelos (templates), além de outras funcionalidades menores como coloração de palavras e preenchimento automático baseado na linguagem e bibliotecas utilizadas. O objetivo deste tipo de software é concentrar as funcionalidades numa só aplicação e facilitar o trabalho do programador, automatizando alguns dos passos para chegar ao produto final. O Code::Blocks é particularmente utilizado para desenvolver software com a linguagem de programação C/C++. Este IDE também já vem com modelos de projetos, inclusive o modelo do Irrlicht Game Engine, para que o programador possa começar a programar o mais rapidamente possível.

Irrlicht Game Engine

O Irrlicht Game Engine é um grupo de bibliotecas de código focadas no desenvolvimento de jogos. A um outro nível, o objetivo de um game engine é semelhante ao de um IDE: agrupa as funcionalidades que se encontram em todos os jogos, tais como entrada (input) de teclado e de rato, inteligência artificial, gráficos, física, interação entre objetos, colisão e outras. O Irrlicht Game Engine engloba: entrada de teclado e rato, bibliotecas para gráficos e funcionalidades básicas de inteligência artificial e física.

Bullet Physics Engine

O Bullet Physics Engine é outro grupo de bibliotecas focado na simulação de física. Ao contrário do Irrlicht, o Bullet não é utilizado apenas em jogos, sendo no entanto é um dos mais utilizados. O Bullet foi incluído para gerir a colisão entre objetos e simular gravidade.

3D Studio Max

O 3D Studio Max é um programa de manipulação de objetos 3D. Este programa serviu para modelar todos os objetos 3D que se encontram no jogo. Para além da modelação

também foi com este programa que as animações da personagem e do monstro foram criadas. Infelizmente tive que utilizar outros programas, que serão posteriormente mencionados, para conseguir completar e transferir os objetos deste programa para o código do jogo.

Adobe Photoshop CS3

O Adobe Photoshop CS3 é um programa de criação e edição de imagens. Neste projeto, com o auxílio de um manual sobre como desenhar texturas para jogos, utilizei o Photoshop para dar cor e texturas aos meus objetos 3D. Também utilizei este programa para criar o mapa do jogo.

fragMotion

O fragMotion é uma aplicação da mesma categoria do 3DS Max, mas menos desenvolvida. A única utilidade deste programa foi converter os ficheiros dos objetos 3D criados em 3D Studio Max num formato utilizável pelo Irrlicht.

Roadkill 3D

O Roadkill 3D tem como objetivo criar mapas 2D de objetos 3D para incluir texturas nesses mesmos objetos. Através da divisão do objeto em partes, o programa consegue produzir um mapa 2D que é utilizado como um padrão quando for desenhada a textura do objeto.

Desenvolvimento

O Conceito

Antes de começar a programar o jogo, tive de imaginar um conceito. Não parecendo, a criação de um conceito é tão difícil como a fases de desenvolvimento que aparecem mais tarde. Isso porque um conceito não é uma simples história nem uma breve explicação. O conceito é um documento, muito semelhante a este, que descreve todos os aspectos criativos e técnicos de um jogo, desde a sua narrativa até às bibliotecas de código que serão utilizadas para o desenvolver. Devido à falta de experiência prévia nesta área, a progressão do meu conceito sofreu várias alterações de acordo com as minhas capacidades em junção com a informação que ia adquirindo ao longo do tempo. Portanto, no meu caso, o conceito não foi uma fase isolada mas sim uma constante no desenvolvimento do jogo que foi progressivamente alterado de acordo com uma estimativa entre o tempo que tinha e o meu conhecimento.

Comecei o meu conceito pela narrativa. Inicialmente limitei-me a escrever ideias criativas que me ocorreram e, pouco a pouco, fui juntando estas ideias soltas numa unidade. No espaço de um mês já tinha um protótipo da primeira narrativa juntamente com algumas noções sobre os aspectos técnicos. Esta primeira narrativa era sobre um americano nativo que se tinha tornado comerciante durante os descobrimentos e o objetivo do jogador era atravessar longas distâncias para vender as suas mercadorias. Pouco tempo depois de ter criado este conceito decidi começar de novo porque me apercebi que não seria capaz de desenvolver algo tão complexo com o tempo que tinha.

Voltando ao início, recomecei por, mais uma vez, escrever ideias e juntá-las. Algumas semanas mais tarde tinha um novo conceito mais simples e mais desenvolvido que o último. Este segundo conceito tinha pouca narrativa, concentrado-se mais na jogabilidade. O jogador era um guerreiro que iria participar em lutas mortais, numa arena, contra vários tipos de monstros e enquanto ia ganhando, ia subindo numa hierarquia de vencedores até

chegar ao topo. O próprio jogo seria um pequeno espaço circular onde o herói poderia escolher a sua arma e lutar contra monstros de dificuldade variável. Satisfeito com este conceito, comecei a pesquisar sobre o software que iria precisar para realizar a minha ideia.

Enquanto pesquisava sobre as ferramentas que teria de dominar, comecei a adquirir alguma confiança nas minhas capacidades e decidi expandir o conceito. Primeiro, retirei a pequena arena e substitui-a por um mapa maior. Tendo um espaço mais largo para explorar, decidi introduzir uma pequena aldeia para dar mais vida ao cenário. Foi nesta altura que percebi que, ao incluir uma aldeia, o próximo passo seria incluir aldeões para ocupar a aldeia. No entanto, os meses que iria demorar a modelar os objetos 3D e a inteligência artificial que teria que programar, entre outros fatores, levou-me a optar por deixar a aldeia abandonada. Esta decisão deu origem à história atual do jogo.

Enquanto redefinia os aspetos criativos do conceito, comecei a tomar decisões acerca do software que iria utilizar. O game engine que escolhi chamava-se Irrlicht Game Engine. Este veio à minha atenção devido à sugestão do meu professor acompanhante. As razões pelas quais decidi utilizar este software foram as seguintes:

- ser software gratuito de código aberto;
- utilizar a linguagem de programação C++, uma das minhas preferências;
- ser um software com uma alta curva de aprendizagem;
- existir uma grande quantidade documentação online sobre o software, o que facilitou em muito a aprendizagem.

De seguida, escolhi o programa de modelação em 3D que iria utilizar. Esta foi uma escolha fácil, optando pelo 3D Studio Max 2010, porque já tinha utilizado este programa na concretização de outros projetos. O Adobe Photoshop CS3 foi a minha escolha de software para desenhar texturas, pela mesma razão. Com estes programas escolhidos, terminei o conceito, planifiquei o meu tempo e finalmente comecei a desenvolver o jogo.

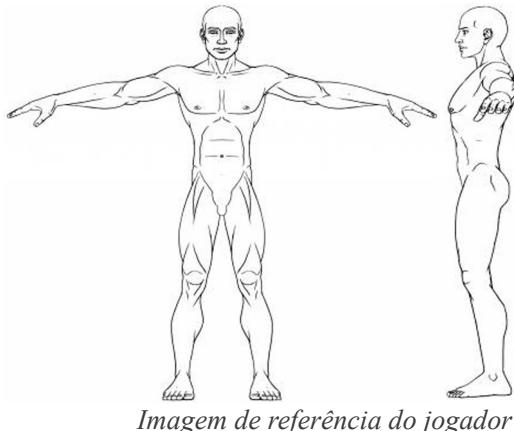
A primeira ação que foi tomada no desenvolvimento do jogo foi a modelação de objetos 3D,

porque sem gráficos é difícil visualizar o progresso do código. Ao longo da sua criação, um objeto 3D tem de passar por cinco fases antes de ser considerado completo. Estas cinco fases são: Modeling, Rigging, Skinning, Animation, Texturing, sendo a 2^a e a 4^a fase específicas para objetos animados.

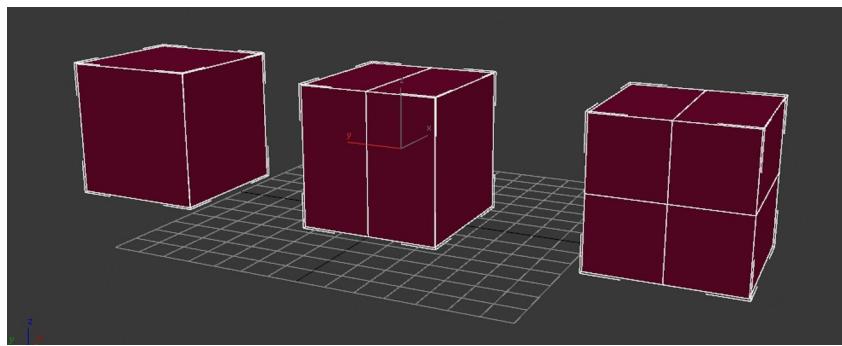
Modelação de Objetos 3D

Modeling

A primeira fase, modeling, é a fase onde um objeto primitivo, normalmente um cubo ou uma esfera, é moldado na forma do objeto que se está a tentar produzir, através da subdivisão. A modelação é normalmente feita com o auxílio de imagens de referência. Uma imagem de referência, ou “blueprint”, é uma imagem que retrata uma perspetiva do objeto que se está a desenvolver. Estas imagens são orientadas de forma a que o modelador consiga moldar o seu objeto da forma como ele, ou a pessoa que fez as imagens de referência, imaginou. Como exemplo, apresento algumas imagens.



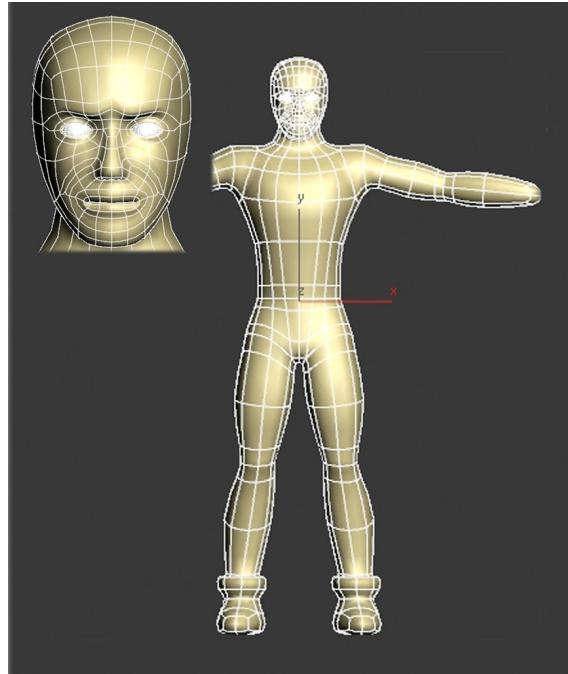
Devido à sua pouca exigência, os primeiros objetos a serem modelados foram os da aldeia abandonada. Apenas um dos edifícios foi criado a partir de uma imagem de referência. No entanto, esta imagem foi a base de inspiração para todos os outros edifícios. Começando com um cubo, o objeto primitivo foi progressivamente subdividido e deformado até se chegar ao produto final. Como os edifícios são objetos de cenário inanimados, estes ultrapassaram as próximas três fases para a fase de texturing, mas irei deixar a explicação desta para o fim.



Exemplo de um objeto a ser subdividido

Depois de ter acabado a maioria dos edifícios passei para a modelação da personagem do jogador. Enquanto a modelagem dos edifícios não requereu muita atenção na posição dos vértices, a modelação da personagem é bastante diferente devido, principalmente, ao facto de ser um objeto orgânico. Como a personagem é um homem, foi dada uma maior atenção ao posicionamento dos vértices de forma a conseguir recriar os músculos e os contornos principais do corpo humano. Se houvesse algum erro na modelação esse erro seria visível, mais tarde, na animação da personagem. Após vários meses, oito tentativas na modelação e a ajuda de um ex-aluno da ETEO, Daniel Inverno, finalmente foi concluída uma personagem que pudesse ser utilizada no jogo. Um dos primeiros erros que teve de ser corrigido foi o uso excessivo da subdivisão. O problema com a subdivisão no desenvolvimento de objetos 3D para jogos é que há um limite de vértices que um objeto não deve exceder porque pode prejudicar a execução do jogo, levando a resultados indesejáveis assim como uma diminuição na velocidade de FPS (Frames Por Segundo). Concluindo, uma das maiores dificuldades em modelar a personagem foi encontrar um equilíbrio entre a quantidade suficiente de vértices para criar um objeto bem definido mas não uma

quantidade que excedesse o limite. Felizmente, o monstro demorou muito menos tempo devido à experiência adquirida e porque a sua criação foi quase idêntica à da personagem.

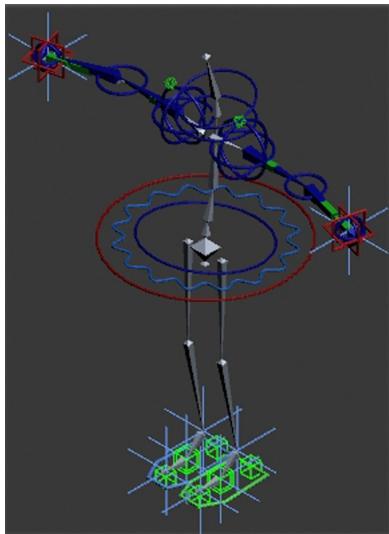


Objeto gráfico final da personagem

Rigging

A próxima fase, rigging, é a fase onde os ossos são criados para possibilitar o movimento de um objeto. Os ossos são objetos 3D normais, mas com a funcionalidade de controlar o movimento de outros objetos 3D. Após a criação do esqueleto e a inclusão de Inverse Kinematics (IK) e Forward Kinematics (FK), que são processos utilizados para definir a rotação e transformação dos membros do esqueleto, foram criados controladores para, mais tarde, facilitar a animação do objeto. Os controladores são objetos primitivos ligados a vários ossos do esqueleto com o objetivo de automatizar animações simples que possam ser utilizadas várias vezes. Alguns exemplos são as várias rotações do pés, o movimento dos elementos que compõem a coluna caso a personagem seja dobrada para a frente ou até o movimento dos pés em juncão com a pernas quando a personagem estiver a andar.

Para criar o esqueleto da personagem foi utilizado um grupo de tutoriais¹ que explicavam



O esqueleto do jogador rodeado pelos controladores

meticulosamente cada passo, não só da criação do esqueleto mas também a fase do skinning que irá ser descrita de seguida. Durante o desenvolvimento do esqueleto as dimensões dos ossos tiveram que ser reguladas porque estes iriam mais tarde afetar a animação da personagem. Tal e qual como os ossos de uma pessoa, se as dimensões estiverem desproporcionadas a movimentação do corpo irá ser afetado. Por outro lado a modelação da personagem e do monstro também tiveram que ter algum rigor técnico pois qualquer erro teria de ser corrigido e o esqueleto reconstruído de raiz.

Skinning

Esta próxima fase está ligada à anterior porque após a criação do esqueleto os ossos têm que ser unidos com o objeto 3D para que este possa ser animado. Não sendo uma fase muito difícil de completar, requer bastante paciência. Utilizando um modificador no 3D Studio Max chamado "Skin", o modelador começa por definir com o modificador quais os ossos que pertencem ao objeto 3D. Depois dos ossos serem unidos ao objeto, os vértices desse objeto têm que ser ligados aos ossos. Existem dois métodos para realizar esta ação. Um deles é a utilização de envelopes, que são caixas de seleção que ligam ao osso selecionado todos os vértices que se encontram dentro dele. O outro método, mais utilizado, é a ligação ao nível do vértice. O modelador tem que selecionar os vértices que pretende ligar ao osso selecionado. A ligação de um vértice a um osso tem uma propriedade “força”. Tal significa que o modelador pode não só ligar um vértice a um osso mas também definir a influência que esse osso tem no vértice. Isto possibilita a ligação de um vértice a vários ossos, sendo vantajoso por permitir a criação de animações mais autênticas. Exemplos encontram-se com maior frequência em modelos com articulações, como por exemplo um joelho, onde mais do que um osso contribui para o movimento de flexão. Enquanto se vai desenvolvendo esta

¹ http://www.youtube.com/watch?v=_gm_g8do32k&list=PLnKw1txyYzRlxh1-BT4CifPXC5TBg2vUd

fase, é recomendado que o modelador vá testando a animação do modelo e corrija a ligação vértice/osso se algum erro for encontrado.

Ambos estes métodos têm as suas vantagens e desvantagens. A vantagem dos envelopes é a rapidez com que o modelador pode completar o skinning do objeto. No entanto, o processo é quase completamente automatizado, dando pouca liberdade ao modelador. Este método só deve de ser aplicado em objetos pouco complicados. Outra forma de utilizar este método é como uma primeira camada para criar uma ligação vértice/osso geral e continuar com um trabalho ao pormenor através do segundo método mencionado. A vantagem da ligação ao nível do vértice é a quantidade de controlo que o modelador tem no resultado final da união vértice/osso. A única desvantagem é a quantidade de tempo despendida devido ao trabalho minucioso. No entanto, com alguma experiência, esta desvantagem pode facilmente ser superada, tornando-a no método mais adequado. É por esta razão que o segundo método é utilizado com maior frequência em ambientes profissionais.

Tal como já foi dito anteriormente, foi utilizado um grupo de tutoriais para completar esta fase. No entanto, esta fase foi um processo de tentativa e erro. Com as informações obtidas através do tutorial, comecei a trabalhar no skinning. Devido a algumas falhas cometidas nas fases anteriores, tive que refazer o modelo da personagem e todas as fases a partir daí. Após várias tentativas acabei o skinning e avancei no desenvolvimento do objeto 3D.

Animation

Na fase de animation o modelador, ou neste caso o animador, dinamiza os objetos 3D através do movimento. Se as fases anteriores foram completadas corretamente esta fase não será complicada. A função do animador é pegar em todas as animações definidas no objeto e juntá-las, de forma organizada, no “timeline”. É no “timeline” que toda a animação do objeto é guardada para que mais tarde possa ser acedida, neste caso, pelo Irrlicht Game Engine, quando for chamada pelo código do jogo. Normalmente a primeira animação que é posta no timeline é uma animação nula onde o objeto está imóvel com os braços esticados

para cada lado e com as pernas direitas e juntas. Esta posição é conhecida como T-Pose.

Comecei a animação de ambos os modelos (personagem e monstro) com o T-Pose. A partir do T-Pose fiz a animação do andamento cíclico. Esta consiste na personagem/monstro andar normalmente, mas quando a animação chega ao fim e volta de novo ao início ocorre uma transição suave de forma a parecer que há um movimento contínuo, razão do nome da animação. Demorou-me bastante tempo a fazer esta animação porque tive algumas dificuldades em perceber as posições das pernas e dos pés, mas com a ajuda de imagens que demonstravam o posicionamento de cada passo consegui acabá-la. As outras animações que foram implementadas no modelo da personagem foram: correr, atacar e lançar o fogo-de-artificio. As animações do monstro foram: correr e atacar.

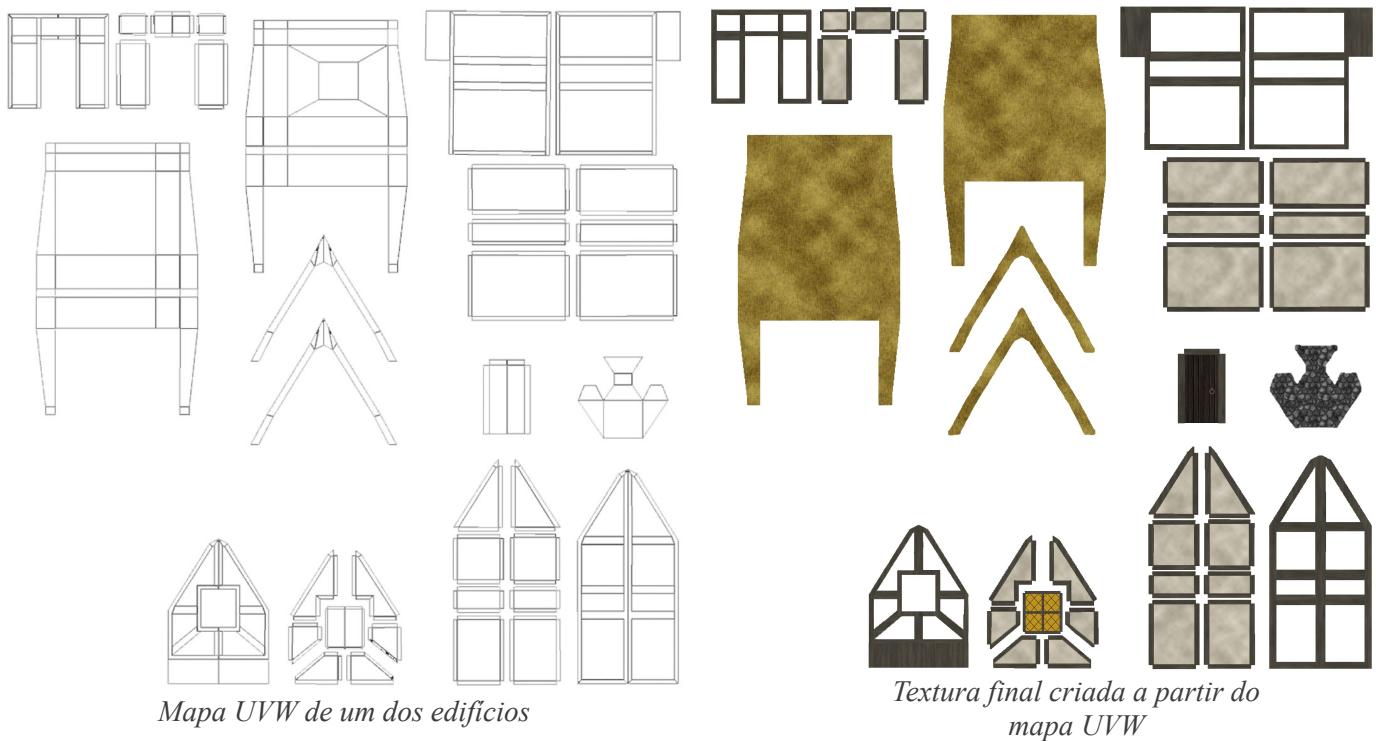
Texturing

Para conseguir adaptar uma textura a um objeto 3D é necessário criar uma representação, um mapa, do objeto em 2D porque uma textura não é nada mais do que uma imagem. Este processo chame-se UVW Mapping. UVW são os eixos da imagem em contraste com os eixos XYZ, utilizados pelo objeto. Na sua forma mais simples, o UVW Mapping é feita em três passos: “desembrulhar” o objeto, criar a textura e aplicar a textura ao objeto. Através do programa de modelação 3D, o modelador começa por organizar as faces do objeto numa superfície 2D. A maneira como o mapa é organizado depende do objeto em si. Se o objeto for um cubo ele é desdobrado em forma de cruz. Caso o objeto seja a cabeça de uma pessoa, há a tendência de dividir a parte detrás da cabeça e esticar cada metade para os lados até que todas as faces sejam vistas no mapa sem estarem sobrepostas. Com o mapa criado, o modelador escolhe um programa de edição de imagem para criar a textura, utilizando o mapa UVW como modelo. Finalmente, a imagem é levada de volta ao programa de modelação 3D onde é “embrulhada” no objeto. Estes passos podem ter ligeiras variações de acordo com o software escolhido.

Os primeiros mapas que foram criados foram os dos edifícios. Com o 3D Studio Max, o UVW Mapping dos edifícios não foi complicado porque este consistia em separar o telhado

e as paredes, isolando quaisquer detalhes que se encontrassem dentro destes. Quando o mapa de um edifício era terminado, abria o Adobe Photoshop CS3 e, com o auxílio de um manual sobre como desenvolver texturas para jogos, começava a desenhar as texturas, utilizando o UVW Map como um desenho técnico ("Blueprint"). Finalmente, com a textura concluída, atribuiá esta como material ("material" é o nome que se dá a uma textura no 3D Studio Max) no objeto 3D.

Os mapas mais difíceis foram os do monstro e da personagem porque como estes eram modelos orgânicos o UVW Mapping também teve que ser feito de forma orgânica. Inicialmente experimentei as ferramentas UVW Mapping fornecidas pelo 3DStudio Max, mas cheguei à conclusão que eram pouco automatizadas, não sendo eficientes para completar o trabalho. Depois de alguma pesquisa na Internet descobri um programa chamado Roadkill 3D. O Roadkill 3D é uma ferramenta UVW Mapping que possibilita criar mapas, indiretamente, no 3DStudio Max através de um plugin. Com esta ferramenta apenas precisei de dividir os objetos 3D em várias partes e o programa tratava da organização do mapa.



O mapa gráfico do jogo é produzido através de um “heightmap” criado no Photoshop. O “heightmap” é uma imagem utilizada para guardar valores que podem ser usados em diversas funções. Neste projeto o “heightmap” foi utilizado para guardar a altura de todos os pontos do mapa gráfico. Os valores de altura são representados através das cores preto, branco e todos os tons de cinzento. Por norma o branco representa o ponto mais alto do mapa e o preto o ponto mais baixo enquanto os pontos cinzentos representam diferentes alturas de acordo com a sua tonalidade, ou seja, quanto mais claro for o cinzento maior a altura do ponto.



“Heightmap” utilizado para criar o mapa gráfico do jogo

Programação do Jogo

Esta parte do projeto foi a mais exigente e exaustiva, mas também a mais interessante e recompensadora. Quando decidi que a PAP iria ser um vídeo jogo 3D ainda não tinha a correta noção dos conhecimentos necessários para conseguir realizar esta ideia, mas o objetivo sempre foi esse: começar com nada e crescer até acabar o jogo. Isto tudo porque é a minha crença que a PAP deve de ser uma oportunidade para explorar uma área desconhecida e que exprime o interesse do aluno.

O que é a Programação e o Programador

Antes de começar a escrever sobre o meu progresso nesta fase, falarei um pouco sobre a programação, a função do programador e a sua utilização neste projeto. Para um programador, um projeto é, essencialmente, um problema. A função do programador é dividir o problema nas suas componentes e, através das ferramentas à sua disposição, resolver cada componente do modo mais simples e eficaz possível. Normalmente isto é feito antes de começar a programar para que o programador não esteja longos períodos de tempo parado em frente do computador a tentar resolver o problema inteiro na sua cabeça. Só quando o programador tiver, pelo menos, uma ideia geral de como resolver o problema é que começa a programar.

A programação é o processo de planear, escrever, testar, depurar e manter o código fonte de um programa. O código fonte pode ser escrito em qualquer uma das inúmeras linguagens de programação, tais como C, C++, Python, BASIC, PASCAL, Lua, Java, e a lista continua. Dentro da programação existem vários paradigmas. Um paradigma é um estilo de programação que deve ser escolhido de acordo com o tipo de problema que se está a tentar resolver. Um exemplo de um paradigma é a “programação orientada por objetos” (POO), que separa as componentes dum problema em elementos de código isolados uns dos outros, com a capacidade de se multiplicarem e interagirem. Em código, isto é feito através da criação de classes. Uma classe pode ser vista como um desenho técnico de onde se pode

produzir inúmeras instâncias (cópias). A programação orientada por objetos é o paradigma adotado, no geral, por todos os programadores de jogos. O propósito final é criar um conjunto de instruções que são lidas pelo computador para exibirem o resultado desejado.

Programação em Contexto da PAP

Após várias semanas a testar código e aprender como se trabalhava com o Irrlicht Game Engine comecei a escrever o código do jogo. Mais tarde comecei a trabalhar na física e na colisão entre objetos. O desenvolvimento do código pode-se subdividir em quatro partes principais: O mapa, o jogador, os monstros e o fogo de artifício.

O Mapa

O mapa foi o primeiro elemento a implementar no jogo devido à sua simplicidade. Este consiste apenas em um objeto gráfico, imóvel numa posição predefinida, e um objeto de colisão que se encontra no mesmo plano que o mapa gráfico para que todos os outros objetos não caiam através dele. Como só é preciso existir uma instância do mapa e só tem a função de estar imóvel no espaço foi programado em código corrido, ou seja, não se utilizou o paradigma de programação orientada por objetos pois este é utilizado quando se pretende criar cópias do mesmo objeto ou esse objeto apresenta um comportamento complexo. Cada vez que se inicia o jogo o mapa é o primeiro objeto a ser lido e carregado.

O Jogador

O jogador é composto por cinco elementos: o objeto gráfico e físico do jogador, o objeto gráfico e físico da arma e a câmara. O jogador foi o primeiro a utilizar o paradigma POO pois apesar de haver apenas uma instância esta iniciativa possibilitará a possível criação de um modo multijogador no futuro. O código do jogador também controla a arma, uma espada, utilizada para matar os monstros. Apesar de fazer sentido criar uma classe para o jogador e outra para a arma, como o jogador utiliza uma única arma cuja principal função é

detetar colisões com os monstros, optei por não complicar a estrutura do código. Outras funcionalidades implementadas no jogador foram o movimento e a rotação da câmara. O jogador consegue-se movimentar nas quatros direções através das teclas W(frente), A(esquerda), S(para trás) e D(direita). A "frente" do jogador é definida por um vetor que é controlado pela rotação da câmara. A rotação é produzida pelo movimento do rato e cada vez que a "frente" é redefinida a rotação do jogador é atualizada. O resultado final é um jogador que se consegue movimentar pelo mapa utilizando as direção relativas para onde está a olhar. Algumas propriedades atribuídas na classe jogador são: vitalidade, velocidade e o dano produzido pela espada. Todas estas propriedades são variáveis com valores numéricos. A vitalidade representa a vida do jogador e quando esta é igual ou inferior a zero, o jogador morre e o jogo acaba. A velocidade é a distância que o jogador se move em cada segundo quando está em movimento. O dano é o valor que é retirado da vitalidade de um monstro quando este colide com a espada. Ao decidir implementar uma vista em primeira pessoa (First Person View), optei por criar e utilizar apenas os braços do jogador na versão final do jogo.

O Monstro

Tal como os restantes objetos incorporados no mapa, o monstro consiste em dois elementos: o objeto gráfico e o objeto físico, representados na imagem seguinte.



Representação simultânea dos objetos gráficos e físicos

O monstro tem uma classe bastante mais complicada em relação à classe do jogador porque enquanto as funcionalidades do jogador são controladas pelos eventos do teclado e do rato, todas as funcionalidades do monstro têm que ser criadas com o auxílio da inteligência artificial. Para o monstro é utilizado um “State machine”. Um “State machine” é uma estrutura de informação, sendo uma das formas mais básicas de atribuir inteligência artificial a um sujeito. Essencialmente, o “State machine” age como um gerador dos vários estados (“states”) do sujeito, controlando as transições entre eles e qual é que está ativo em cada momento. No caso do monstro, os seus estados são:

- inativo
- em movimento
- em perseguição do jogador
- a atacar.

Quando são cumpridas condições predefinidas, o “State machine” faz a transição entre o estado ativo e um novo estado. Por exemplo, se o jogador estiver “próximo” do Monstro, este passa ao estado de perseguição.

Outro ramo da inteligência artificial que foi implementada no monstro foi o “Pathfinding”. O “Pathfinding” é utilizado quando queremos que um sujeito, ao saber o seu destino, se desloque até este, pelo caminho mais rápido e eficiente. Tudo começa com a criação de um “Node grid”. Um “Node grid” é um mapa composto por nódulos que contêm uma posição no espaço. Este mapa pode ser composto por objetos gráficos ou, simplesmente, informação abstrata guardada em estruturas de informação complexas. O objetivo deste mapa é criar pontos de referência por onde o monstro se pode guiar até ao seu destino. Antes de caminhar para o seu destino, o monstro começa por analisar cada nódulo do mapa repetidamente até criar um caminho. Inicialmente todos os nódulos são postos numa lista de nódulos não verificados e à medida que são analisados são postos em uma de duas listas: a lista de nódulos rejeitados ou a lista de nódulos que fazem parte do caminho do monstro.

A última funcionalidade que implementei no monstro foi o ataque. Quando o monstro está no estado de perseguição e chega a uma certa distância do jogador o “State machine” faz a transição para o estado de ataque. Enquanto está em ataque, o monstro executa a animação correspondente e se houver colisão entre o monstro e o jogador, este perde vida.



Monstro em modo de ataque, a ser atingido pelo jogador

Avaliação

O objetivo final foi atingido, mas com algumas mudanças. Estas foram feitas à medida que ia descobrindo a dificuldade de várias técnicas que queria implementar em relação ao tempo que tinha. Raramente houve remoção de elementos importantes, mas tive que simplificar alguns aspetos assim como o "pathfinding" do monstro e limitar algumas das características do fogo de artifício. Os aspetos com os quais fiquei mais satisfeito foram os modelos 3D do jogador, do monstro e dos edifícios. Também fiquei bastante satisfeito com o deslocamento e rotação do jogador e da câmara. Fiquei menos satisfeito com os elementos que tive de eliminar ou reduzir na sua complexidade, como a possibilidade do jogador poder criar fogo de artifício personalizado e o nível de inteligência artificial dos monstros. Certas partes do código do jogo poderão ser reescritas de forma mais eficiente e elegante.

Conclusão

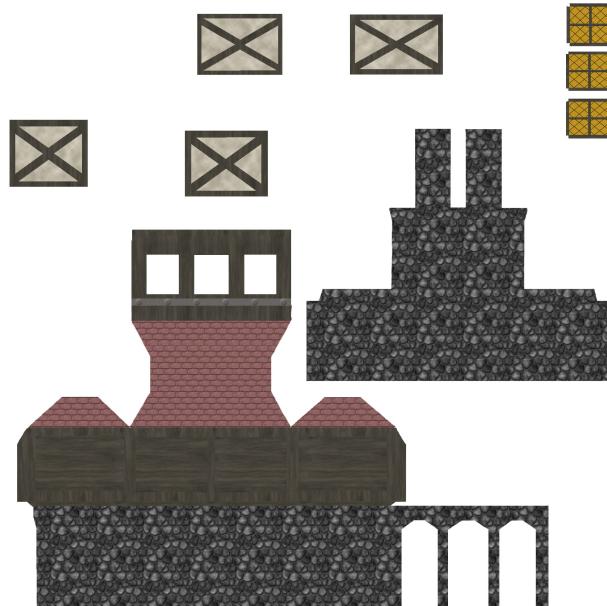
Como é usual na criação de um jogo, houve várias alterações ao longo de todo o desenvolvimento, mas no seu todo os objetivos foram concluídos e fiquei satisfeito com o resultado final. É um tipo de jogo apropriado aos destinatários inicialmente propostos. Ao terminar, a minha intenção é pegar em tudo o que aprendi e começar um novo projeto, continuando a evoluir nesta área.

Anexo 01

Modelos 3D e Texturas



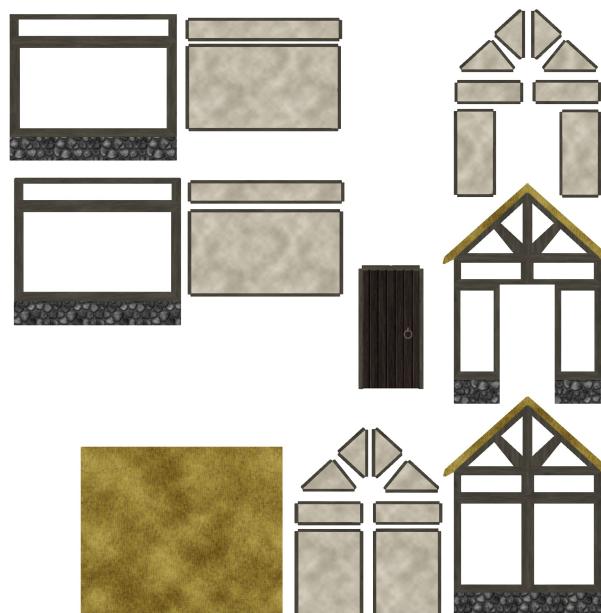
Edifício - “Arched Building”



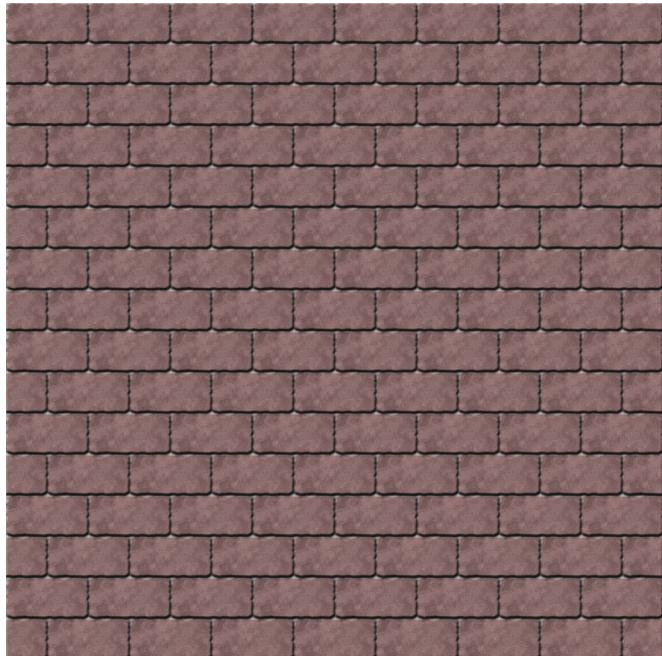
Mapa de Textura do “Arched Building”



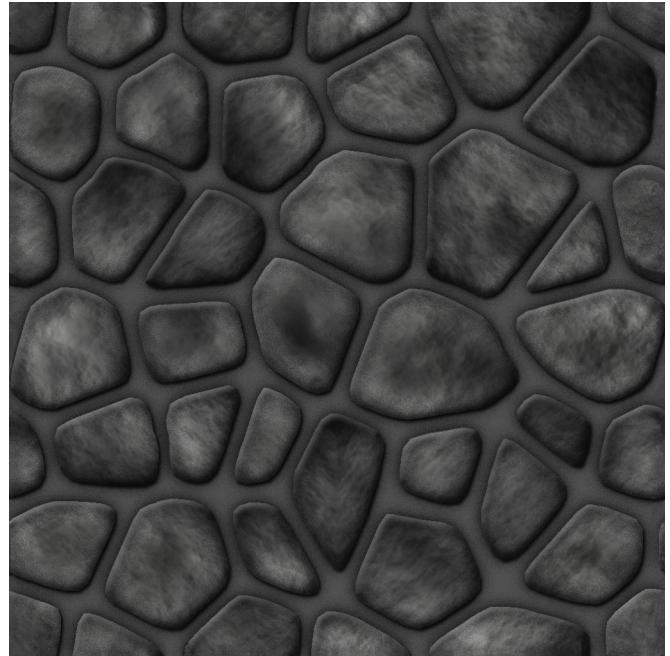
Edifício - "House 02"



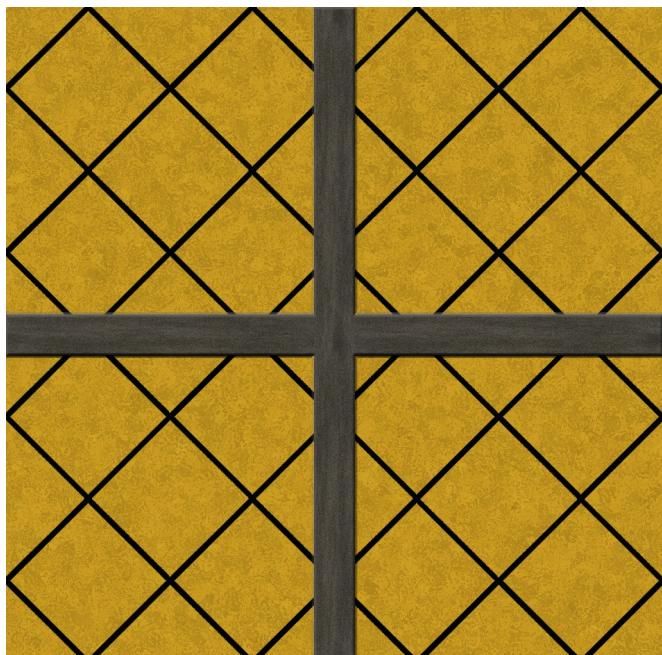
Mapa de Textura do "House 02"



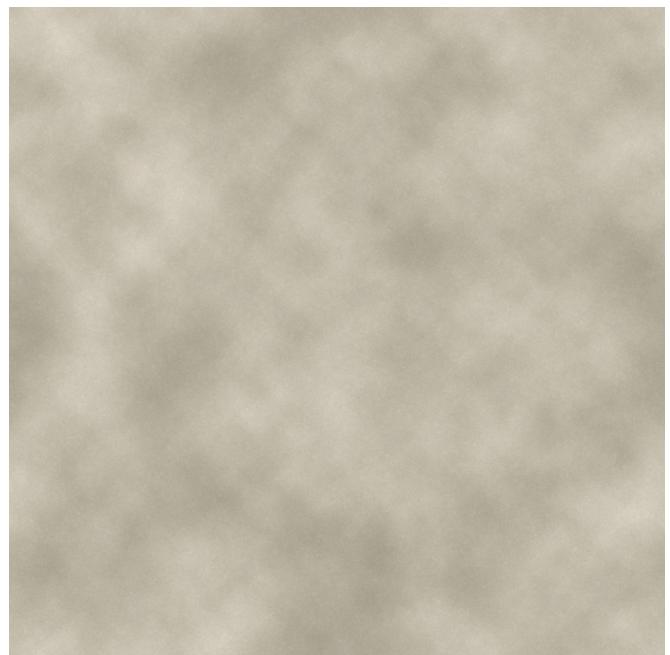
Textura - Tijolo



Textura – Muro de Pedra



Textura - Janela



Textura - Argamassa



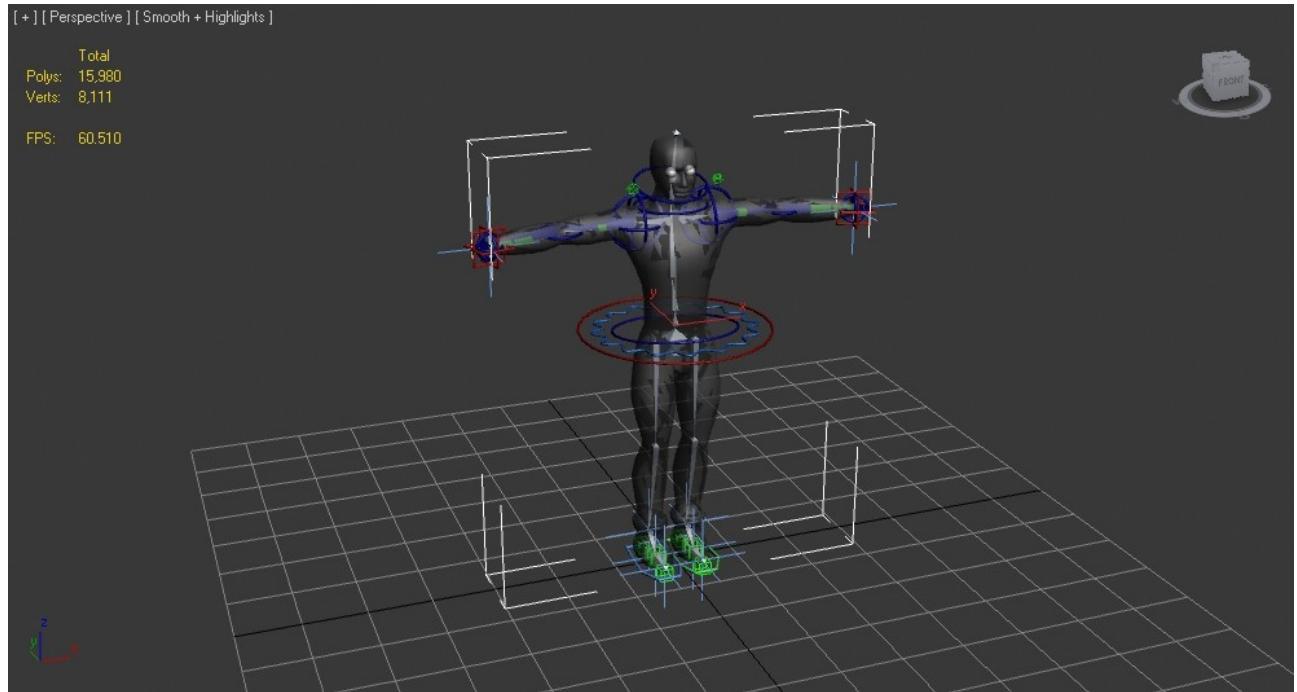
Modelo 3D da Árvore



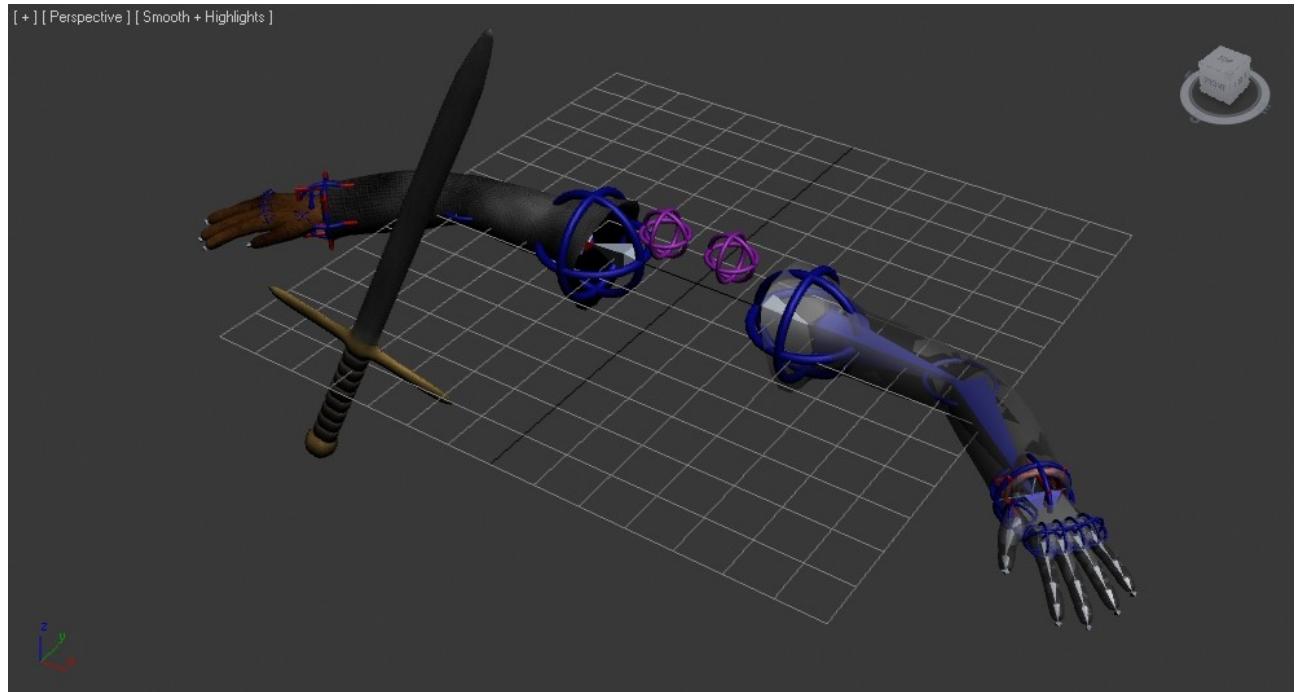
Textura - Madeira



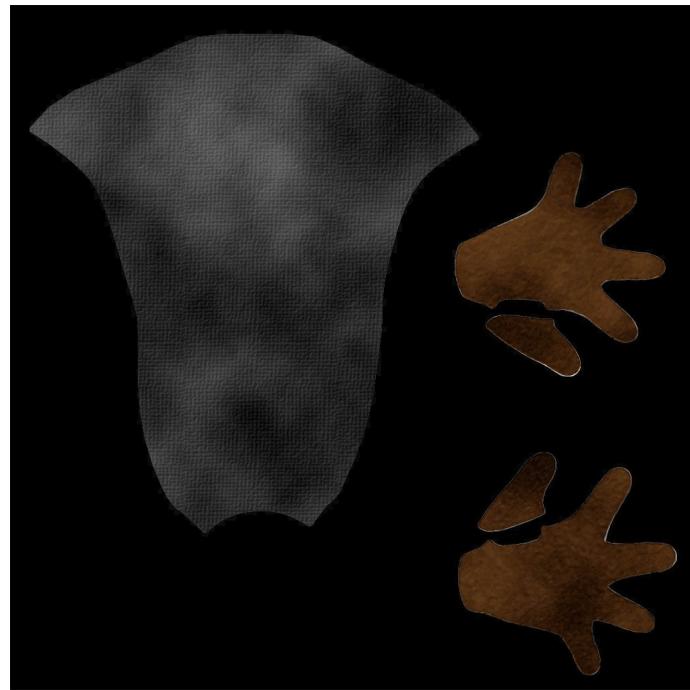
Textura – Tronco de Árvore



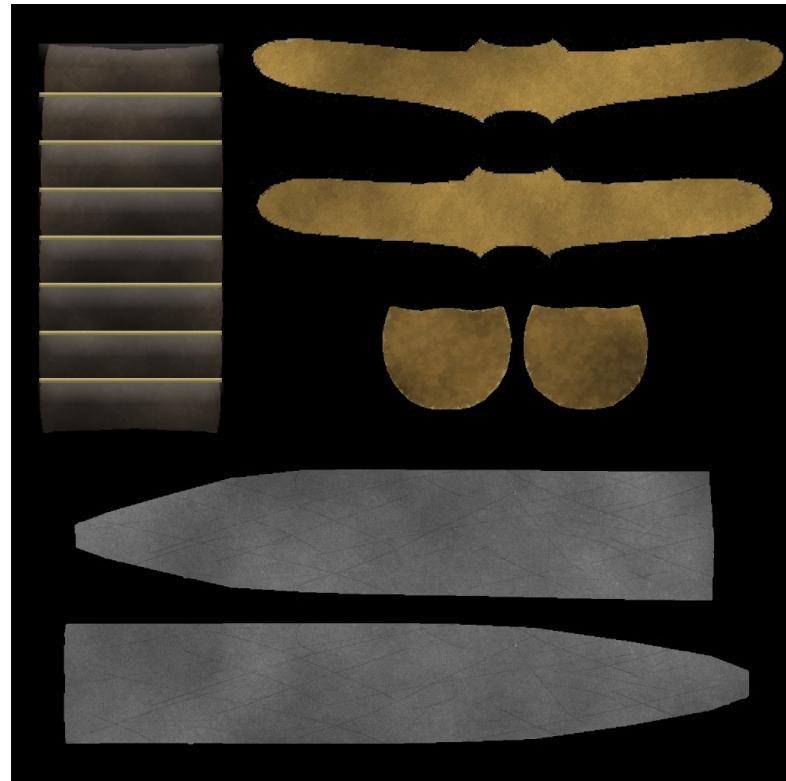
Modelo do Jogador (transparente) e os ossos de animação



Modelo dos braços do jogador com os ossos de animação e a espada



Textura do braço direito do jogador



Textura da espada

Anexo 02

Código Fonte

```
1 #include <iostream>
2 #include <math.h>
3 #include <time.h>
4 #include <irrlicht.h>
5 #include <btBulletCollisionCommon.h>
6 #include <btBulletDynamicsCommon.h>
7 #include <BulletCollision/CollisionShapes/btHeightfieldTerrainShape.h>
8 #include "BulletCollision/CollisionDispatch/btGhostObject.h"
9 #include "collision_filter.h"
10 #include "event_receiver.h"
11 #include "scene_object.h"
12 #include "fireworks.h"
13 #include "manager.h"
14 #include "dummy.h"
15 #include "spawn.h"
16 #include "debugDraw.cpp"
17
18 #define LOCK_CURSOR true
19
20 using namespace std;
21 using namespace irr;
22 using namespace scene;
23 using namespace video;
24 using namespace gui;
25 using namespace core;
26
27 void UpdateRender(btRigidBody *TObject);
28 void getHeightfieldData(int terrainSize, ITerrainSceneNode *node, f32 *heightDataPtr, f32 *minHeight, f32 *maxHeight);
29
30 int main()
31 {
32 //    IrrlichtDevice *nullVice=createDevice(EDT_NULL);
33 //    dimension2d<u32> resolution(1366, 768); //nullVice->getVideoModeList()
34 //    ->getDesktopResolution();
35 //    nullVice->drop();
36     IrrlichtDevice *device=createDevice(EDT_OPENGL, dimension2d<u32>(1366,
37 768), 16, false, false, false, 0);
38     device->setWindowCaption(L"Game Pre-Alpha");
39
40     /// Initializing Irrlicht.
41     ISceneManager *smgr=device->getSceneManager();
42     IVideoDriver *driver=device->getVideoDriver();
43     IGUIEnvironment *guienv=device->getGUIEnvironment();
44
45     /// Bullet default setup.
46     btBroadphaseInterface *m_broadphase=new btDbvtBroadphase();
47
48     btDefaultCollisionConfiguration *m_collisionConfiguration=new btDefault
49     CollisionConfiguration();
50     btCollisionDispatcher *m_dispatcher=new btCollisionDispatcher(m_collisi
51     onConfiguration);
52
53     btSequentialImpulseConstraintSolver *solver=new btSequentialImpulseCons
54     traintSolver;
55
56     btDynamicsWorld *m_dynamicsWorld=new btDiscreteDynamicsWorld(m_dispatch
57     er, m_broadphase, solver, m_collisionConfiguration);
58     btVector3 worldGravity(0.0f, -50.0f, 0.0f);
59     m_dynamicsWorld->setGravity(worldGravity);
60
61     /// Set rand() seed.
```

```
56     srand(time(NULL));
57
58     // Declaring delta time variables.
59     f32 deltaTime;
60     u32 then=device->getTimer()->getTime();
61     position2d< s32> cursorPosition;
62     float sensitivity=0.08f;
63     float throwStrength=0.0f;
64     bool attacking=false;
65
66     vector2d<u32> desktopResolution=device->getVideoModeList()->getDesktopResolution();
67     vector2d< s32> screenCenter;
68     screenCenter.X=desktopResolution.X/2;
69     screenCenter.Y=desktopResolution.Y/2;
70
71     // Adding a terrain mesh to the game via heightmap.
72     ITexture *heightMapTexture=driver->getTexture("./assets/map_image_resized.jpg");
73     ITerrainSceneNode *map=smgr->addTerrainSceneNode(heightMapTexture->getName(), 0, -1, vector3df(0.0f, 0.0f, 0.0f), vector3df(0.0f, 0.0f, 0.0f), vector3df(1.0f, 1.0f, 1.0f));
74     map->setMaterialTexture(0, driver->getTexture("./assets/terrain_texture_map.jpg"));
75     map->setMaterialFlag(EMF_LIGHTING, false);
76
77     // Retrieving heightfield height data from mesh declared above.
78     int worldCollidesWith=PLAYER_COLLISION|FIREWORK_COLLISION|MONSTER_COLLISION;
79     int arraySize=heightMapTexture->getSize().Width;
80     f32 minHeight=0.0f, maxHeight=0.0f;
81     f32 heightData[arraySize*arraySize];
82     f32 *heightDataPtr=&heightData[0];
83
84     getHeightfieldData(arraySize, map, heightDataPtr, &minHeight, &maxHeight);
85
86     btCollisionShape *groundShape=new btHeightfieldTerrainShape(arraySize, arraySize, heightDataPtr, 1.f, minHeight, maxHeight, 1, PHY_FLOAT, false);
87     map->setScale(vector3df(10.0f, 2.6f, 10.0f));
88     groundShape->setLocalScaling(btVector3(map->getScale().X-0.1f, map->getScale().Y-0.1f, map->getScale().Z-0.1f));//(btVector3(9.86f, 3.0f, 9.86f));
89
90     // Ground motion state.
91     btTransform transform;
92     vector3df irrTerrainTrans=map->getTerrainCenter();
93     btVector3 initPos(irrTerrainTrans.X, irrTerrainTrans.Y, irrTerrainTrans.Z);
94     transform.setIdentity();
95     transform.setOrigin(initPos);
96     btDefaultMotionState *groundMotionState=new btDefaultMotionState(transform);
97
98     // Ground rigid body.
99     btRigidBody::btRigidBodyConstructionInfo groundRigidBodyCI(0, groundMotionState, groundShape, btVector3(0, 0, 0));
100    btRigidBody *groundRigidBody=new btRigidBody(groundRigidBodyCI);
101    groundRigidBody->setUserPointer((void *)map);
102
103    // Adding ground rigid body to the world.
104    m_dynamicsWorld->addRigidBody(groundRigidBody, WORLD_COLLISION, worldCollidesWith);
```

```
105     /// Adding skydome.
106     IMeshSceneNode *skydome=smgr->addMeshSceneNode(smgr->getMesh("./assets/
107 skydome.DAE"));
108     skydome->setMaterialFlag(EMF_LIGHTING, false);
109     skydome->setMaterialTexture(0, driver->getTexture("./assets/t_skydome.j
110 pg"));
111     skydome->setPosition(vector3df(2360.0f, 0.0f, 2360.0f));
112     skydome->setRotation(vector3df(-90.0f, 0.0f, 0.0f));
113     skydome->setScale(vector3df(50.0f, 50.0f, 50.0f));
114
115     /// Adding water to the world.
116     IAnimatedMesh *waterMesh=smgr->addHillPlaneMesh("waterMesh", dimension2
d<f32>(150.0f, 150.0f), dimension2d<u32>(35, 35), 0, 0,
117                                         dimension2d<f32>(0.0f,
118                                         0.0f), dimension2d<f32>(10.0f, 10.0f));
119
120     ISceneNode *waterNode=smgr->addWaterSurfaceSceneNode(waterMesh->getMesh
(0), 5, 300, 14);
121     waterNode->setPosition(vector3df(2560.0f, 240.0f, 2560.0f));
122
123     waterNode->setMaterialTexture(0, driver->getTexture("./assets/river_sto
nes_texture.psd"));
124     waterNode->setMaterialTexture(1, driver->getTexture("./assets/water.jpg
"));
125     waterNode->setMaterialType(EMT_REFLECTION_2_LAYER);
126     waterNode->setMaterialFlag(EMF_LIGHTING, false);
127
128     /// Setting up player.
129     Player player(device, m_dynamicsWorld, smgr->getMesh("./assets/untitled
.obj"), driver->getTexture("./assets/player_texture.jpg"),
130                     vector3df(1.0f, 0.0f, 0.0f), vector3df(0.0f, 1.0f, 0.0f))
;
131
132     /// Player setup done!
133
134     /// Event receiver setup.
135     SApplicationContext context;
136     context.device=device;
137     EventReceiver receiver(context);
138     device->setEventReceiver(&receiver);
139
140     device->getCursorControl()->setPosition(position2df(0.5f, 0.5f));
141
142     /// Boolean used to switch between player's first person perspective an
d the flying camera.
143     bool FP_POV=true;
144     /// FPS Camera for looking around the map.
145     ICameraSceneNode *FPSCamera=smgr->addCameraSceneNodeFPS(0, 150.f, 1.f);
146     FPSCamera->setFarValue(10000);
147     device->getCursorControl()->setVisible(false);
148     smgr->setActiveCamera(player.getCamera());
149
150     /// Setting up debugDraw.
151     DebugDraw debugDraw(device);
152     debugDraw.setDebugMode(btIDebugDraw::DBG_DrawWireframe |
153                             //btIDebugDraw::DBG_DrawAabb |
154                             btIDebugDraw::DBG_DrawContactPoints |
155                             //btIDebugDraw::DBG_DrawText |
156                             //btIDebugDraw::DBG_DrawConstraintLimits |
157                             btIDebugDraw::DBG_DrawConstraints);
158
159     m_dynamicsWorld->setDebugDrawer(&debugDraw);
160
```

```
157     irr::video::SMaterial debugMat;
158     debugMat.Lighting=false;
159
160     bool debug_draw_bullet=false;
161
162     /// Defining the different firework types.
163     FireworkInfo firework01(3200, 3400, SColor(255, 0, 0, 0), SColor(255, 2
164     55, 255, 255), 500, 500,
165             dimension2df(10.0f, 10.0f), dimension2df(20.0f,
166             20.0f), 360, driver->getTexture("./assets/firework_particles/red_particle.
167             png"));
168     FireworkInfo firework02(3200, 3400, SColor(255, 0, 0, 0), SColor(255, 2
169     55, 255, 255), 500, 500,
170             dimension2df(10.0f, 10.0f), dimension2df(20.0f,
171             20.0f), 360, driver->getTexture("./assets/firework_particles/green_partic
172             le.png"));
173     FireworkInfo firework03(3200, 3400, SColor(255, 0, 0, 0), SColor(255, 2
174     55, 255, 255), 500, 500,
175             dimension2df(10.0f, 10.0f), dimension2df(20.0f,
176             20.0f), 360, driver->getTexture("./assets/firework_particles/blue_particle
177             .png"));
178     FireworkInfo firework04(3200, 3400, SColor(255, 0, 0, 0), SColor(255, 2
179     55, 255, 255), 500, 500,
180             dimension2df(10.0f, 10.0f), dimension2df(20.0f,
181             20.0f), 360, driver->getTexture("./assets/firework_particles/purple_partic
182             le.png"));
183     /// Creating firework manager.
184     Manager mngr;
185
186     /// Creating randomly positioned trees.
187
188     int size=40, ray=800, spacing=150;
189     bool addTree=false;
190     vector3df center(1500.0f, 290.0f, 2500.0f);
191     ScenicObject *forest01[size];
192     srand((unsigned)time(0));
193     btCollisionShape *treeShape=new btBoxShape(btVector3(10, 100, 10));
194
195     /// Set all array elements to NULL.
196     for(int i=0; i<size;i++){forest01[i]=NULL;}
197
198     /// Add object to the first element in the array.
199     forest01[0]=new ScenicObject(device, m_dynamicsWorld, smgr->getMesh("./
200         assets/forest_tree.DAE"), 0, getPointInCircle(center, ray),
201             vector3df(-90.0f, 0.0f, 0.0f), vector3df(1.0f
202             , 1.0f, 1.0f), driver->getTexture("./assets/tree_texture.jpg"), treeShape);
203     // Adding tree branches as child node.
204     forest01[0]->addChildNode(device, smgr->getMesh("./assets/forest_tree_b
205         ranches.DAE"), vector3df(0.0f, 0.0f, 0.0f),
206             vector3df(0.0f, 0.0f, 0.0f), vector3df(1.0f, 1.0
207             f, 1.0f), driver->getTexture("./assets/tree_texture.jpg"));
208
209     /// in the first parameter the array element count starts with the seco
210     nd element.
211     for(int i=1; i<size; addTree=true)
212     {
213         /// Get position for new object.
214
215         vector3df point=getPointInCircle(center, ray);
216         cout << point.X << endl;
217         cout << point.Z << endl;
218     }
```

```
202     for(int n=0; n<size; n++)
203     {
204         if(forest01[n]!=NULL)
205         {
206             int dist=sqrt(pow(point.X-forest01[n]->getGraphicsObject()->getPosition().X, 2) + pow(point.Z-forest01[n]->getGraphicsObject()->getPosition().Z, 2));
207             if(dist<spacing){addTree=false;}
208         }
209     }
210
211     if(addTree==true)
212     {
213         forest01[i]=new ScenicObject(device, m_dynamicsWorld, smgr->getMesh("./assets/forest_tree.DAE"), 0, point,
214                                     vector3df(-90.0f, 0.0f, 0.0f), vector3df(1.0f, 1.0f, 1.0f), driver->getTexture("./assets/tree_texture.jpg"), treeShape);
215         forest01[i]->addChildNode(device, smgr->getMesh("./assets/forest_tree_branches.DAE"), vector3df(0.0f, 0.0f, 0.0f),
216                                     vector3df(0.0f, 0.0f, 0.0f), vector3df(1.0f, 1.0f, 1.0f), driver->getTexture("./assets/tree_texture.jpg"));
217         cout << i;
218         i++;
219     }
220 }
221
222 size=40;
223 ray=700;
224 spacing=120;
225 addTree=false;
226 center=vector3df(3550.0f, 290.0f, 3800.0f);
227 ScenicObject *forest02[size];
228
229 /// Set all array elements to NULL.
230 for(int i=0; i<size;i++){forest02[i]=NULL;}
231
232 /// Add object to the first element in the array.
233 forest02[0]=new ScenicObject(device, m_dynamicsWorld, smgr->getMesh("./assets/forest_tree.DAE"), 0, getPointInCircle(center, ray),
234                                     vector3df(-90.0f, 0.0f, 0.0f), vector3df(1.0f, 1.0f, 1.0f), driver->getTexture("./assets/tree_texture.jpg"), treeShape);
235 // Adding tree branches as child node.
236 forest02[0]->addChildNode(device, smgr->getMesh("./assets/forest_tree_branches.DAE"), vector3df(0.0f, 0.0f, 0.0f),
237                                     vector3df(0.0f, 0.0f, 0.0f), vector3df(1.0f, 1.0f, 1.0f), driver->getTexture("./assets/tree_texture.jpg"));
238
239 /// in the first parameter the array element count starts with the second element.
240 for(int i=1; i<size; addTree=true)
241 {
242     /// Get position for new object.
243     vector3df point=getPointInCircle(center, ray);
244
245     for(int n=0; n<size; n++)
246     {
247         if(forest02[n]!=NULL)
248         {
249             int dist=sqrt(pow(point.X-forest02[n]->getGraphicsObject()->getPosition().X, 2) + pow(point.Z-forest02[n]->getGraphicsObject()->getPosition().Z, 2));
250             if(dist<spacing){addTree=false;}
```

```
251         }
252     }
253
254     if(addTree==true)
255     {
256         forest02[i]=new ScenicObject(device, m_dynamicsWorld, smgr->get
257 Mesh("./assets/forest_tree.DAE"), 0, point,
258             vector3df(-90.0f, 0.0f, 0.0f), vector3df(1.0f
259 , 1.0f, 1.0f), driver->getTexture("./assets/tree_texture.jpg"), treeShape);
260         forest02[i]->addChildNode(device, smgr->getMesh("./assets/fores
261 t_tree_branches.DAE"), vector3df(0.0f, 0.0f, 0.0f),
262             vector3df(0.0f, 0.0f, 0.0f), vector3df(1.0f, 1.0
263 f, 1.0f), driver->getTexture("./assets/tree_texture.jpg"));
264         cout << i;
265         i++;
266     }
267
268 // Monster Spawners
269 SpawnEssentials ess(device, m_dynamicsWorld);
270 Spawn spawn01(ess, smgr->getMesh("./assets/spawn_mesh.DAE"), driver->ge
271 tTexture("./assets/spawn_texture.jpg"), vector3df(1412.0f, 300.0f, 1512.0f
272 );
273 Spawn spawn02(ess, smgr->getMesh("./assets/spawn_mesh.DAE"), driver->ge
274 tTexture("./assets/spawn_texture.jpg"), vector3df(1412.0f, 300.0f, 3800.0f
275 );
276
277 // Character animated arm.
278 IAnimatedMeshSceneNode *rightArm=smgr->addAnimatedMeshSceneNode(smgr->g
279 etMesh("./assets/arm_ani_test.b3d"), player.getNode(), 42);
280 rightArm->setLoopMode(true);
281 rightArm->setFrameLoop(11, 15);
282 rightArm->setAnimationSpeed(15);
283 rightArm->setMaterialFlag(EMF_LIGHTING, false);
284 rightArm->setMaterialTexture(0, driver->getTexture("./assets/Right_Arm_
285 Texture.psd"));
286 rightArm->setPosition(vector3df(-23.0f, -20.0f, 0.0f));
287 rightArm->setRotation(vector3df(0.0f, 90.0f, 0.0f));
288 rightArm->setScale(vector3df(0.8, 0.8, 0.8));
289
290 AnimationEndCallBack rightArmCallback;
291 rightArmCallback.setAttackPtr(&attacking);
292 rightArm->setAnimationEndCallback(&rightArmCallback);
293
294 IMeshSceneNode *sword=smgr->addMeshSceneNode(smgr->getMesh("./assets/Sw
295 ord.DAE"), rightArm->getJointNode("Right_Palm_Bone"));
296 sword->setMaterialFlag(EMF_LIGHTING, false);
297 sword->setMaterialTexture(0, driver->getTexture("./assets/Sword_Texture
298 _Map.psd"));
299 sword->setPosition(vector3df(-8.0f, -5.0f, 0.0f));
300 sword->setRotation(vector3df(180.0f, 0.0f, 25.0f));
301
302 // Setting up village.
303 IMeshSceneNode *villageCenter=smgr->addSphereSceneNode(10);
304 villageCenter->setPosition(vector3df(1500.0f, 300.0f, 3500.0f));
305 villageCenter->updateAbsolutePosition();
306
307 // First arched building.
308 btCollisionShape *archedBuildingShape=new btBoxShape(btVector3(90, 100,
309 130));
310 ScenicObject archedBuilding(device, m_dynamicsWorld, smgr->getMesh("./a
```

```

300     sets/village/arched_building.DAE"), villageCenter,
301     vector3df(-700.0f, 70.0f, 0.0f), vector3df(
302     -90.0f, 20.0f, 0.0f), vector3df(2.0f, 2.0f, 2.0f),
303     driver->getTexture("./assets/village/textures/arched_building_texture.jpg"),
304     archedBuildingShape);
305
306     /// First house type 01.
307     btCollisionShape *house_01_BuildingShape=new btBoxShape(btVector3(100,
308     50, 50));
309     ScenicObject house_01_Building(device, m_dynamicsWorld, smgr->getMesh(
310     "./assets/village/house_building_01.DAE"), villageCenter,
311     vector3df(0.0f, 0.0f, 0.0f), vector3df(0.0f,
312     0.0f, 0.0f), vector3df(2.0f, 2.0f, 2.0f),
313     driver->getTexture("./assets/village/textures/house_building_01_texture.jpg"),
314     house_01_BuildingShape);
315
316     /// First house type 02.
317     btCollisionShape *house_02_BuildingShape=new btBoxShape(btVector3(45,
318     100, 45));
319     ScenicObject house_02_Building(device, m_dynamicsWorld, smgr->getMesh(
320     "./assets/village/house_building_02.DAE"), villageCenter,
321     vector3df(-500.0f, 0.0f, 300.0f), vector3df(
322     -90.0f, 140.0f, 0.0f), vector3df(1.0f, 1.0f, 1.0f),
323     driver->getTexture("./assets/village/textures/house_building_02_texture.jpg"),
324     house_02_BuildingShape);
325
326     IMeshSceneNode *branch=smgr->addMeshSceneNode(smgr->getMesh("./assets/
327     forest_tree_branch.DAE"));
328     branch->setScale(vector3df(1.0f, 1.0f, 1.0f));
329
330     while(device->run())
331     {
332         /// Delta time update.
333         const f32 now=device->getTimer()->getTime();
334         deltaTime=(f32)(now-then)/1000.f;
335         then=now;
336
337         /// Input events go here.
338         if(receiver.IsKeyDown(KEY_ESCAPE))
339         {
340             device->closeDevice();
341             break;
342         }
343
344         if(receiver.isCursorMoving())
345         {
346             cursorPosition=device->getCursorControl()->getPosition();
347
348             if(LOCK_CURSOR)
349             {
350                 if((cursorPosition.X==683 && cursorPosition.Y==372) ||
351                     (cursorPosition.X==0 && cursorPosition.Y==0)){receiver.
352                 resetCursorMove();}
353                 else
354                 {
355                     cursorPosition.X-=683;
356                     cursorPosition.Y-=372;
357                     device->getCursorControl()->setPosition(position2d<f32>
358                     (0.5f, 0.5f));
359
360                     /// Add the player update function here.
361                 }
362             }
363         }
364     }

```

```
348             player.pRotation(vector2df(cursorPosition.X*sensitivity
349 , cursorPosition.Y*sensitivity));
350         }
351     }
352
353     if(receiver.IsKeyPressed(KEY_KEY_C)==1)
354     {
355         if(FP_POV)
356         {
357             smgr->setActiveCamera(FPSCamera);
358             FP_POV=false;
359         }
360         else
361         {
362             smgr->setActiveCamera(player.getCamera());
363             FP_POV=true;
364         }
365     }
366
367     if(receiver.IsKeyDown(KEY_KEY_W)){player.pMovement(deltaTime, FORWARD);
368     if(receiver.IsKeyDown(KEY_KEY_S)){player.pMovement(deltaTime, BACKWARD);
369
370     if(receiver.IsKeyDown(KEY_KEY_A)){player.pMovement(deltaTime, LEFT);
371     if(receiver.IsKeyDown(KEY_KEY_D)){player.pMovement(deltaTime, RIGHT);
372
373     if(receiver.IsKeyDown(KEY_KEY_1)){player.setSelectedFW(TYPE01);}
374     if(receiver.IsKeyDown(KEY_KEY_2)){player.setSelectedFW(TYPE02);}
375     if(receiver.IsKeyDown(KEY_KEY_3)){player.setSelectedFW(TYPE03);}
376     if(receiver.IsKeyDown(KEY_KEY_4)){player.setSelectedFW(TYPE04);}
377
378     if(receiver.IsKeyDown(KEY_PLUS)){player.setInventoryItem(player.getSelectedFW(),
379     player.getInventoryItemValue(player.getSelectedFW())+1);}
380     if(receiver.IsKeyDown(KEY_MINUS)){player.setInventoryItem(player.getSelectedFW(),
381     player.getInventoryItemValue(player.getSelectedFW())-1);}
382
383     if(receiver.IsMouseButtonPressed(M_LEFT)==1)
384     {
385         if(!attacking)
386         {
387             attacking=true;
388             rightArm->setLoopMode(false);
389             rightArm->setFrameLoop(15, 25);
390             for(int i=0; i<MONSTER_CAP; i++)
391             {
392                 if(spawn01.getMonsters(i)!=NULL)
393                 {
394                     bool intersect=sword->getTransformedBoundingBox().intersectsWithBox(spawn01.getMonsters(i)->getNode()->getTransformedBoundingBox());
395                     if(intersect)
396                     {
397                         spawn01.getMonsters(i)->setDamageTexture(driver->getTexture("./assets/damage_texture.jpg"));
398                         spawn01.getMonsters(i)->removeVitality(5);
399                     }

```

```
400             if(spawn02.getMonsters(i)!=NULL)
401             {
402                 bool intersect=sword->getTransformedBoundingBox().i
403                 ntersectsWithBox(spawn02.getMonsters(i)->getNode()->getTransformedBoundingBoxB
404                 ox());
405                     if(intersect)
406                     {
407                         spawn02.getMonsters(i)->setDamageTexture(driver
408                         ->getTexture("./assets/damage_texture.jpg"));
409                         spawn02.getMonsters(i)->removeVitality(5);
410                     }
411             }
412         }
413         if(receiver.IsMouseButtonUp(M_RIGHT))
414         {
415             throwStrength+=10.0f;
416         }
417         if(receiver.IsMouseButtonPressed(M_RIGHT)==0)
418         {
419             int selectedFirework=player.getSelectedFW();
420             if(player.getInventoryItemValue(player.getSelectedFW())>0)
421             {
422                 switch(selectedFirework)
423                 {
424                     case 0:
425                         mngr.addFirework(device, firework01, m_dynamicsWorld, d
426                         river->getTexture("./assets/spawn_texture.jpg"),
427                         player.getCamera()->getTarget(), player.getForwardVecto
428                         r(), throwStrength);
429                         break;
430
431                     case 1:
432                         mngr.addFirework(device, firework02, m_dynamicsWorld, d
433                         river->getTexture("./assets/spawn_texture.jpg"),
434                         player.getCamera()->getTarget(), player.getForwardVecto
435                         r(), throwStrength);
436                         break;
437
438                     case 2:
439                         mngr.addFirework(device, firework03, m_dynamicsWorld, d
440                         river->getTexture("./assets/spawn_texture.jpg"),
441                         player.getCamera()->getTarget(), player.getForwardVecto
442                         r(), throwStrength);
443                         break;
444
445                     case 3:
446                         mngr.addFirework(device, firework04, m_dynamicsWorld, d
447                         river->getTexture("./assets/spawn_texture.jpg"),
448                         player.getCamera()->getTarget(), player.getForwardVecto
449                         r(), throwStrength);
449                     break;
450
451             player.setInventoryItem(selectedFirework, player.getInvento
452             ryItemValue(selectedFirework)-1);
453         }
```

```

450         throwStrength=0;
451     }
452
453     receiver.resetKeyPressed();
454     receiver.resetMousePressed();
455
456     m_dynamicsWorld->stepSimulation(1/60.f, 10);
457
458     /// Update player's camera target.
459     player.cUpdateTarget();
460
461     mngr.update(deltaTime);
462
463     /// Synchronize the player's graphics body with the physics body.
464     UpdateRender(player.getPlayerRigidBody());
465
466     /// Synchronize monster's graphics and physics bodies.
467
468     /// Redraw firework.
469     for(int i=0; i<100; i++){if(mngr.redrawFireworks(i)!=false){UpdateR
ender(mngr.redrawFireworks(i));}}
470
471     /// Spawner update function
472     spawn01.updateMonsters(device, m_dynamicsWorld, &player, smgr->getM
esh("./assets/animated_zombie.b3d"),
473                             driver->getTexture("./assets/zombie_texture.jp
g"), deltaTime);
474
475     spawn02.updateMonsters(device, m_dynamicsWorld, &player, smgr->getM
esh("./assets/animated_zombie.b3d"),
476                             driver->getTexture("./assets/zombie_texture.jp
g"), deltaTime);
477
478     /// Updating player's health bar.
479     float health=player.getHealth();
480     float healthBarWidth=(500*health)/100;
481
482     /// Clearing GUI Environment each frame so the 2d images and text a
ren't painted over each other.
483     guienv->clear();
484
485     /// Draw everything in the scene.
486     driver->beginScene(true, true, SColor(0, 255, 255, 255));
487     smgr->drawAll();
488
489     driver->draw3DLine(player.getCamera()->getAbsolutePosition(), playe
r.getCamera()->getAbsolutePosition()+player.getForwardVector(), SColor(255,
255, 0, 0));
490
491     /// User Interface.
492
493     driver->draw2DImage(driver->getTexture("./assets/health_bar.jpg"),
position2d<s32>(683-(healthBarWidth/2), 20),
494                                         rect<s32>(position2d<s32>(0, 0), dimension2d<s3
2>(healthBarWidth, 20)),
495                                         0, SColor(255,255,255,255), true);
496
497     driver->draw2DImage(driver->getTexture("./assets/r1Bag.png"), posit
ion2d<s32>(1306, 10), rect<s32>(position2d<s32>(0, 0), dimension2d<s32>(50,
50)),
498                                         0, SColor(255,255,255,255), true);
499     guienv->addStaticText(stringw(player.getInventoryItemValue(0)).c_st

```

```
500      (), rect<s32>(1325, 60, 1375, 70));
501      driver->draw2DImage(driver->getTexture("./assets/r2Bag.png"), position2d<s32>(1306, 80), rect<s32>(position2d<s32>(0, 0), dimension2d<s32>(50, 50)),
502          0, SColor(255,255,255,255), true);
503      guienv->addStaticText(stringw(player.getInventoryItemValue(1)).c_st
r(), rect<s32>(1325, 130, 1375, 140));
504      driver->draw2DImage(driver->getTexture("./assets/r3Bag.png"), position2d<s32>(1306, 150), rect<s32>(position2d<s32>(0, 0), dimension2d<s32>(50, 50)),
505          0, SColor(255,255,255,255), true);
506      guienv->addStaticText(stringw(player.getInventoryItemValue(2)).c_st
r(), rect<s32>(1325, 200, 1375, 210));
507      driver->draw2DImage(driver->getTexture("./assets/r4Bag.png"), position2d<s32>(1306, 220), rect<s32>(position2d<s32>(0, 0), dimension2d<s32>(50, 50)),
508          0, SColor(255,255,255,255), true);
509      guienv->addStaticText(stringw(player.getInventoryItemValue(3)).c_st
r(), rect<s32>(1325, 270, 1375, 280));
510
511      switch(player.getSelectedFW())
512      {
513          case 0:
514              driver->draw2DImage(driver->getTexture("./assets/output.png"),
position2d<s32>(1306, 10), rect<s32>(position2d<s32>(0, 0), dimension2d<s32
>(50, 50)),
515                  0, SColor(255,255,255,255), true);
516          break;
517
518          case 1:
519              driver->draw2DImage(driver->getTexture("./assets/output.png"),
position2d<s32>(1306, 80), rect<s32>(position2d<s32>(0, 0), dimension2d<s32
>(50, 50)),
520                  0, SColor(255,255,255,255), true);
521          break;
522
523          case 2:
524              driver->draw2DImage(driver->getTexture("./assets/output.png"),
position2d<s32>(1306, 150), rect<s32>(position2d<s32>(0, 0), dimension2d<s32
>(50, 50)),
525                  0, SColor(255,255,255,255), true);
526          break;
527
528          case 3:
529              driver->draw2DImage(driver->getTexture("./assets/output.png"),
position2d<s32>(1306, 220), rect<s32>(position2d<s32>(0, 0), dimension2d<s32
>(50, 50)),
530                  0, SColor(255,255,255,255), true);
531          break;
532
533          default:
534              break;
535      }
536      guienv->drawAll();
537
538      if (debug_draw_bullet)
539      {
540          driver->setMaterial(debugMat);
541          driver->setTransform(irr::video::ETS_WORLD, irr::core::Identity
Matrix);
542          m_dynamicsWorld->debugDrawWorld();
543      }
544  }
```

```

543
544 //         integer-=1*deltaTime;
545 //         if(integer<=0)
546 //         {
547 //             switch(debug_draw_bullet)
548 //             {
549 //                 case true:
550 //                     debug_draw_bullet=false;
551 //                     integer=200;
552 //                     break;
553 //
554 //                 case false:
555 //                     debug_draw_bullet=true;
556 //                     integer=4;
557 //                     break;
558 //             }
559 //         }
560         driver->endScene();
561     }
562 }
563
564
565 /// Passes bullet's orientation to irrlicht
566 void UpdateRender(btRigidBody *T0bject)
567 {
568     ISceneNode *node=static_cast

```

```
601     vector3df maxEdge=box.MaxEdge/scale;
602
603     for(f32 z=minEdge.Z; z<maxEdge.Z; z+=stepWidthZ)
604     {
605         for(f32 x=minEdge.X; x<maxEdge.X; x+=stepWidthX)
606         {
607             const f32 curVal=((ITerrainSceneNode*)node)->getHeight(x, z);
608
609             heightDataPtr++;
610             *heightDataPtr=curVal;
611
612             if(curVal>*maxHeight){*maxHeight=curVal;}
613             if(curVal<*minHeight){*minHeight=curVal;}
614         }
615     }
616 }
617 }
```

```
1 #ifndef SPAWN_CLASS_INCLUDED
2 #define SPAWN_CLASS_INCLUDED
3
4 #define MONSTER_CAP 5
5
6 #include <iostream>
7 #include <irrlicht.h>
8 #include "monster.h"
9 #include "dummy.h"
10
11 using namespace irr;
12
13 struct SpawnEssentials
14 {
15     IrrlichtDevice *device;
16     btDynamicsWorld *dynamicsWorld;
17
18     SpawnEssentials(IrrlichtDevice *dev, btDynamicsWorld *world)
19     {
20         device=dev;
21         dynamicsWorld=world;
22     }
23 };
24
25 class Spawn
26 {
27     public:
28     Spawn(SpawnEssentials ess, scene::IMesh *spawnMesh, video::ITexture *spawnTexture, core::vector3df position);
29     ~Spawn();
30
31     //void addMonster(scene::IAnimatedMesh *monsterMesh, video::ITexture *monsterTexture);
32     void updateMonsters(IrrlichtDevice *dev, btDynamicsWorld *dynamicsWorld,
33     Player *player,
34                     scene::IAnimatedMesh *monsterMesh, video::ITexture
35     *monsterTexture, float dt);
36
37     Monster *getMonsters(int i);
38
39     private:
40     /// Checks if player is within monster's "view area".
41     void isPlayerNear(int element, scene::IMeshSceneNode *p);
42     /// Checks if any monsters needs to be destroyed.
43     void isDead(int element, Player *p);
44     /// Current live monster count in the spawner.
45     unsigned int currMonsters;
46     /// Cooldown period after spawning a new monster.
47     float cooldown;
48     SpawnEssentials *essentials;
49     /// Graphics Object
50     scene::IMeshSceneNode *spawnNode;
51     /// Physics Object.
52     // Pointer array where zombie objects will be referenced.
53     Monster *monster[MONSTER_CAP];
54 };
```

```
1 #include "spawn.h"
2
3 Spawn::Spawn(SpawnEssentials ess, scene::IMesh *spawnMesh, video::ITexture
4 *spawnTexture, core::vector3df position):currMonsters(0), cooldown(0)
5 {
6     essentials=&ess;
7
8     // Setting up graphics object.
9     spawnNode=essentials->device->getSceneManager()->addMeshSceneNode(spawn
10 Mesh);
11     spawnNode->setMaterialTexture(0, essentials->device->getVideoDriver()->
12 getTexture(spawnTexture->getName()));
13     spawnNode->setMaterialFlag(video::EMF_LIGHTING, false);
14     spawnNode->setPosition(position);
15
16     // Setting monster array elements to null.
17     for(int i=0; i<MONSTER_CAP; i++){monster[i]=NULL;}
18 }
19
20 Spawn::~Spawn(){}
21
22 void Spawn::updateMonsters(IrrlichtDevice *dev, btDynamicsWorld *dynamicsWo
23 rld, Player *player,
24                               scene::IAnimatedMesh *monsterMesh, video::ITextu
25 re *monsterTexture, float dt)
26 {
27     // This cycle updates each monster that isn't dead (empty/NULL).
28     for(int i=0; i<MONSTER_CAP; i++)
29     {
30         if(monster[i]!=NULL)
31         {
32             // Change damage texture.
33             if(monster[i]->getDamageTextureOn()){monster[i]->damageTextureC
34 ountDown(monsterTexture, dt);}
35
36             switch(monster[i]->getState())
37             {
38                 // Spawning state.
39                 case 0:
40                     break;
41
42                 // Idle state.
43                 case 1:
44                 {
45                     // Checks if the monster is ready to die.
46                     isDead(i, player);
47                     isPlayerNear(i, player->getNode());
48
49                     if(monster[i]!=NULL)
50                     {
51                         if(monster[i]->getVitality()<=0) // If the monster'
52                             s health is less than or equal to zero...
53                         {
54                             monster[i]->setState(DYING);
55                             monster[i]->getNode()->setLoopMode(false);
56                             monster[i]->getNode()->setFrameLoop(100, 160);
57                         }
58                     }
59
60                     if(monster[i]->getRest()<=0)
61                     {
62                         core::vector3df destination=getPointInCircle(monste
63 
```

```
56     [i]->getNode()->getPosition(), 200);
57             monster[i]->setDestination(destination);
58             monster[i]->setForwardVector(destination);

59             monster[i]->getNode()->setLoopMode(true);
60             monster[i]->getNode()->setFrameLoop(20, 100);
61             monster[i]->setState(MOVING);
62         }
63     else
64     {
65         float r=monster[i]->getRest();
66         monster[i]->setRest(r-=1*dt);
67     }
68 }
69 break;

70 // Moving state.
71 case 2:
72 {
73     /// Checks if the monster is ready to die.
74     isDead(i, player);
75     isPlayerNear(i, player->getNode());

76     float distance = pow(monster[i]->getDestination().X-mon
77     ster[i]->getNode()->getPosition().X, 2) + pow(monster[i]->getDestination().
78     Z-monster[i]->getNode()->getPosition().Z, 2);
79     if(distance < 1)
80     {
81         monster[i]->getRigidBody()->setLinearVelocity(btVec
82         tor3(0, 0, 0));
83         monster[i]->setRest(5);
84         monster[i]->getNode()->setLoopMode(true);
85         monster[i]->getNode()->setFrameLoop(0, 0);
86         monster[i]->setState(IDLE);
87     }
88     else
89     {
90         monster[i]->moveMonster(dt);
91         monster[i]->SynchPosition();
92     }
93 }
94 break;

95 // Chasing state.
96 case 3:
97 {
98     isDead(i, player);
99     float distance=pow(player->getNode()->getPosition().X-m
100    onster[i]->getNode()->getPosition().X, 2) +
101                                         pow(player->getNode()->getPosition().Z-
102    monster[i]->getNode()->getPosition().Z, 2);

103     if(distance < 500000)
104     {
105         monster[i]->setDestination(player->getNode()->getPo
106         sition());
107         monster[i]->setForwardVector(player->getNode()->get
108         Position());
109         monster[i]->moveMonster(dt);
110         monster[i]->SynchPosition();
111     }
112     else{monster[i]->setState(MOVING);}
```

```
110
111             if(player->getNode()->getTransformedBoundingBox().inter
112     sectsWithBox(monster[i]->getNode()->getTransformedBoundingBox()))
113         {monster[i]->setState(ATTACKING);}
114     }
115
116     // Attacking state.
117     case 4:
118         isDead(i, player);
119         if(!player->getNode()->getTransformedBoundingBox().intersec
tsWithBox(monster[i]->getNode()->getTransformedBoundingBox())){monster[i]->
setState(CHASING);}
120         else
121         {
122             float hp=player->getHealth();
123             hp-=(5.0f/1.0f)*dt;
124             player->setHealth(hp);
125         }
126         break;
127
128     // Bury state.
129     case 6:
130         delete monster[i];
131         monster[i]=NULL;
132         break;
133
134     default:
135         break;
136     }
137 }
138
139
140     /// Spawn new monsters in the pool if there is enough space
141     if(cooldown<=0)
142     {
143         cooldown=20;
144         for(int i=0; i<MONSTER_CAP; i++)
145         {
146             if(monster[i]==NULL)
147             {
148                 monster[i]=new Monster(dev, dynamicsWorld, monsterMesh, mon
sterTexture, spawnNode);
149                 monster[i]->setAnimationCallback(monster[i]);
150                 break;
151             }
152         }
153     }
154     else
155     {
156         /**lower countdown value**/
157         cooldown=cooldown-1*dt;
158     }
159 }
160
161 Monster* Spawn::getMonsters(int i)
162 {
163     return monster[i];
164 }
165
166 void Spawn::isDead(int element, Player *p)
167 {
```

```
168     if(monster[element]!=NULL)
169     {
170         if(monster[element]->getVitality()<=0) // If the monster's health i
171             {
172                 monster[element]->getRigidBody()->setLinearVelocity(btVector3(0
173 .0f, 0.0f, 0.0f));
174                 monster[element]->setState(DYING);
175                 monster[element]->getNode()->setLoopMode(false);
176                 monster[element]->getNode()->setFrameLoop(100, 160);
177                 int random=rand()%4;
178                 p->setInventoryItem(random, p->getInventoryItemValue(random)+1)
179             ;
180         }
181     }
182 void Spawn::isPlayerNear(int element, scene::IMeshSceneNode *p)
183 {
184     float distance=pow(p->getPosition().X-monster[element]->getNode()->getP
185 osition().X, 2)+pow(p->getPosition().Z-monster[element]->getNode()->getPosi
186 tion().Z, 2);
187     if(distance < 500000)
188     {
189         monster[element]->getNode()->setLoopMode(true);
190         monster[element]->getNode()->setFrameLoop(20, 100);
191         monster[element]->setState(CHASING);
192     }
}
```

```
1 #ifndef INCLUDE_SCENE_OBJECT_CLASS
2 #define INCLUDE_SCENE_OBJECT_CLASS
3
4 #include <iostream>
5 #include <irrlicht.h>
6 #include <btBulletCollisionCommon.h>
7 #include <btBulletDynamicsCommon.h>
8 #include "collision_filter.h"
9
10 using namespace std;
11 using namespace irr;
12
13 class ScenicObject
14 {
15     public:
16         ScenicObject(IrrlichtDevice *device, btDynamicsWorld *world, scene::IMesh
17 h *mesh, scene::ISceneNode *parent, core::vector3df position,
18                 core::vector3df rotation, core::vector3df scale, video::ITexture
19 *texture, btCollisionShape *shape);
20         ~ScenicObject();
21
22         void addChildNode(IrrlichtDevice *device, scene::IMesh *mesh, core::vector3df
23 position,
24                     core::vector3df rotation, core::vector3df scale, video::ITexture
25 *texture);
26
27         scene::IMeshSceneNode *getGraphicsObject();
28         btRigidBody *getPhysicsObject();
29
30     private:
31         scene::IMeshSceneNode *graphicsObject;
32         scene::IMeshSceneNode *childNode;
33         btRigidBody *collisionObject;
34         btDefaultMotionState *motionState;
35     };
36
37 #endif
38
```

```
1 #include "scene_object.h"
2
3 ScenicObject::ScenicObject(IrrlichtDevice *device, btDynamicsWorld *world, s
4 cene::IMesh *mesh, scene::ISceneNode *parent, core::vector3df position,
5 core::vector3df rotation, core::vector3df scale,
6 video::ITexture *texture, btCollisionShape *shape)
7
8 {
9     /// Build graphics object.
10    graphicsObject=device->getSceneManager()->addMeshSceneNode(mesh, parent,
11 - 1, position, rotation, scale);
12    graphicsObject->updateAbsolutePosition();
13
14    graphicsObject->setMaterialFlag(video::EMF_LIGHTING, false);
15    graphicsObject->setMaterialTexture(0, texture);
16
17    /// Build collision object.
18    int treeCollidesWith=PLAYER_COLLISION|FIREWORK_COLLISION|MONSTER_COLLISI
19 ON;
20    btCollisionShape *treeShape=shape;
21
22    btScalar mass=0;
23    btVector3 treeInertia;
24    treeShape->calculateLocalInertia(mass, treeInertia);
25
26    /// Setting up position and rotation.
27    btVector3 pos(graphicsObject->getAbsolutePosition().X, graphicsObject->g
28 etAbsolutePosition().Y, graphicsObject->getAbsolutePosition().Z);
29    btTransform trans;
30    trans.setIdentity();
31
32    btQuaternion btQuat;
33    btQuat.setEulerZYX((graphicsObject->getRotation().X+90)*core::DEGTORAD,
34 graphicsObject->getRotation().Y*core::DEGTORAD, graphicsObject->getRotation(
35 ).Z*core::DEGTORAD);
36
37    trans.setOrigin(pos);
38    trans.setRotation(btQuat);
39
40    /// Finishing up physics object.
41    motionState=new btDefaultMotionState(trans);
42    btRigidBody::btRigidBodyConstructionInfo treeRigidBodyCI(mass, motionSta
43 te, treeShape, treeInertia);
44    collisionObject=new btRigidBody(treeRigidBodyCI);
45    collisionObject->setUserPointer((void *)graphicsObject);
46    world->addRigidBody(collisionObject, SCENIC_COLLISION, treeCollidesWith)
47 ;
48    collisionObject->setGravity(btVector3(0.0f, 0.0f, 0.0f));
49 }
50
51 ScenicObject::~ScenicObject(){}
52
53 void ScenicObject::addChildNode(IrrlichtDevice *device, scene::IMesh *mesh,
54 core::vector3df position,
55 core::vector3df rotation, core::vector3df sc
56 ale, video::ITexture *texture)
57 {
58     childNode=device->getSceneManager()->addMeshSceneNode(mesh, graphicsObje
59 ct, -1, position, rotation, scale);
60     childNode->setMaterialFlag(video::EMF_LIGHTING, false);
61     childNode->setMaterialTexture(0, texture);
62 }
```

```
51
52 scene::IMeshSceneNode* ScenicObject::getGraphicsObject(){return graphicsObject;}
53
54 btRigidBody* ScenicObject::getPhysicsObject(){return collisionObject;}
55
```

```
1 #ifndef INCLUDE_ZOMBIE_CLASS
2 #define INCLUDE_ZOMBIE_CLASS
3
4 #include <iostream>
5 #include <cmath>
6 #include <irrlicht.h>
7 #include <btBulletCollisionCommon.h>
8 #include <btBulletDynamicsCommon.h>
9 #include "collision_filter.h"
10
11 using namespace std;
12 using namespace irr;
13
14 enum State{SPAWN=0, IDLE=1, MOVING=2, CHASING=3, ATTACKING=4, DYING=5, BURY=6};
15
16 /// Monster class declaration.
17 class Monster;
18
19 class AnimationEndCallBack:public scene::IAnimationEndCallBack
20 {
21     public:
22         AnimationEndCallBack();
23         ~AnimationEndCallBack();
24
25         void setMonsterPointer(Monster *m);
26         void setAttackPtr(bool *a);
27
28         void OnAnimationEnd(scene::IAnimatedMeshSceneNode *node);
29
30     private:
31         Monster *monsterPtr;
32         bool *attackPtr;
33 };
34
35 core::vector3df getPointInCircle(core::vector3df center, int ray);
36
37 /// Monster class header definition.
38 class Monster
39 {
40     public:
41         Monster(IrrlichtDevice *device, btDynamicsWorld *world, scene::IAnimate
42 dMesh *monsterMesh, video::ITexture *monsterTexture, scene::IMeshSceneNode
43 *parent);
44         ~Monster();
45
46         void setAnimationCallback(Monster *m);
47
48         scene::IAnimatedMeshSceneNode *getNode();
49         btRigidBody *getRigidBody();
50
51         void removeVitality(int vt);
52         int getVitality();
53
54         void setState(State st);
55         int getState();
56
57         void setRest(float r);
58         float getRest();
59
59         void setDestination(core::vector3df d);
60         core::vector3df getDestination();
```

```
60     void setForwardVector(core::vector3df vec);
61     core::vector3df getForwardVector();
62
63     void setDamageTexture(video::ITexture *damageTexture);
64     void damageTextureCountDown(video::ITexture *texture, float dt);
65     bool getDamageTextureOn();
66
67     void moveMonster(float dt);
68
69     void SynchPosition();
70
71 private:
72     /// Health Points.
73     int vitality;
74     /// Keeps track of what the monster physics object collides with.
75     int monsterCollidesWith;
76     /// Current state of the monster.
77     State state;
78     /// Speed of translation.
79     float velocity;
80     /// Variable keeps track of how much time is left before the monster moves again.
81     float resting;
82     /// Boolean which tells if the damage texture is on.
83     bool damageTextureOn;
84     /// Amount of time before the monster's texture is reverted back to normal.
85     float damageTextureTimer;
86     /// Destination that the monsters must reach when in the moving state.
87     core::vector3df destination;
88     /// Monster forward vector.
89     core::vector3df forwardVect;
90     /// Used to trigger events at the end of an animation.
91     AnimationEndCallBack callBack;
92     /// Graphics Object
93     scene::IAnimatedMeshSceneNode *monster;
94     /// Monster motion state.
95     btDefaultMotionState *motionState;
96     /// Physics Object.
97     btRigidBody *monsterRigidBody;
98     /// Physics world.
99     btDynamicsWorld *dynamicsWorld;
100 };
101 #endif
102
103
```

```
1 #include "monster.h"
2
3 /// AnimationEndCallBack class definition.
4 AnimationEndCallBack::AnimationEndCallBack(){}
5 AnimationEndCallBack::~AnimationEndCallBack(){}
6
7 void AnimationEndCallBack::setMonsterPointer(Monster *m){monsterPtr=m;}
8
9 void AnimationEndCallBack::setAttackPtr(bool *a){attackPtr=a;}
10
11 void AnimationEndCallBack::OnAnimationEnd(scene::IAnimatedMeshSceneNode *node)
12 {
13     switch(node->getID())
14     {
15         /// right arm node.
16         case 42:
17             node->setLoopMode(true);
18             node->setFrameLoop(11, 15);
19             *attackPtr=false;
20             break;
21
22         /// left arm node.
23         case 16:
24             break;
25
26         /// monster node.
27         case 666:
28             switch(monsterPtr->getState())
29             {
30                 case SPAWN:
31                 {
32                     monsterPtr->getNode()->setLoopMode(true);
33                     monsterPtr->getNode()->setFrameLoop(20, 100);
34                     monsterPtr->setState(MOVING);
35                 }
36                 break;
37
38                 case DYING:
39                 {
40                     monsterPtr->setState(BURY);
41                 }
42                 break;
43
44                 default:
45                 break;
46             }
47             break;
48
49         default:
50             break;
51     }
52 }
53
54 core::vector3df getPointInCircle(core::vector3df center, int ray)
55 {
56     float rotation=rand()/(float)RAND_MAX*2*3.1416;
57     float distance=rand()/(float)RAND_MAX*ray;
58
59     float x = cos(rotation)*distance+center.X;
60     float z = sin(rotation)*distance+center.Z;
61 }
```

```
62     return core::vector3df(x, center.Y, z);
63 }
64
65 // Monster class definition.
66 Monster::Monster(IrrlichtDevice *device, btDynamicsWorld *world, scene::IAnimatedMesh *monsterMesh, video::ITexture *monsterTexture, scene::IMeshSceneNode *parent)
67 :vitality(2), state(SPAWN), velocity(20), resting(0), damageTextureOn(false),
68 , damageTextureTimer(0), destination(core::vector3df(0.0f, 0.0f, 0.0f))
69 {
70     dynamicsWorld=world;
71     // Setting up graphics object
72     monster=device->getSceneManager()->addAnimatedMeshSceneNode(monsterMesh,
73 , 0, 666);
74     monster->setMaterialTexture(0, device->getVideoDriver()->getTexture(monsterTexture->getName()));
75     monster->setMaterialFlag(video::EMF_LIGHTING, false);
76     monster->setPosition(parent->getPosition());
77
78     // Set destination, forward vector and graphic object's rotation.
79     destination=getPointInCircle(monster->getPosition(), 200); // Gets
80     a random point within a predefined circle.
81
82     setForwardVector(destination);
83
84     float rot=atan2(forwardVect.Z, forwardVect.X)*core::RADTODEG;
85
86     // Rotating monster according to the forward vector.
87     monster->setRotation(core::vector3df(0.0f, rot+90, 0.0f));
88
89     // Get monster node bounding box.
90     core::vector3df edges=monster->getTransformedBoundingBox().getExtent();
91
92     // Setting up physics object.
93     monsterCollidesWith=WORLD_COLLISION|SCENIC_COLLISION|PLAYER_COLLISION;
94     btCollisionShape *monsterShape=new btCapsuleShape(8, 50);
95     btScalar mass=2;
96     btVector3 monsterInertia;
97     monsterShape->calculateLocalInertia(mass, monsterInertia);
98
99     // Setting initial position and rotation.
100    btTransform trans;
101    trans.setIdentity();
102    btVector3 initPos(monster->getPosition().X, monster->getPosition().Y+(7
103 , monster->getPosition().Z);
104
105    btQuaternion btQuat;
106    btQuat.setEulerZYX(0.0f, rot*core::DEGTORAD, 0.0f);
107
108    trans.setOrigin(initPos);
109    trans.setRotation(btQuat);
110
111    motionState=new btDefaultMotionState(trans);
112    btRigidBody::btRigidBodyConstructionInfo monsterRigidBodyCI(mass, motionState, monsterShape, monsterInertia);
113    monsterRigidBody=new btRigidBody(monsterRigidBodyCI);
114    monsterRigidBody->setUserPointer((void *)monster);
115    world->addRigidBody(monsterRigidBody, MONSTER_COLLISION, monsterCollide
sWith);
116    monsterRigidBody->setAngularFactor(btVector3(0.0f, 0.0f, 0.0f));
117    monsterRigidBody->setActivationState(DISABLE_DEACTIVATION);
118 //    monsterRigidBody->setGravity(btVector3(0, 0, 0));
```

```
115     /// Node animation setup.  
116     monster->setLoopMode(false);  
117     monster->setFrameLoop(170, 310);  
118 }  
119  
120 Monster::~Monster()  
121 {  
122     damageTextureOn=false;  
123     monster->remove();  
124     monsterRigidBody->setUserPointer((void *)NULL);  
125     delete monsterRigidBody->getMotionState();  
126     dynamicsWorld->removeRigidBody(monsterRigidBody);  
127     delete monsterRigidBody;  
128 }  
129  
130  
131 void Monster::setAnimationCallback(Monster *m)  
132 {  
133     /// Setting up animation end callback.  
134     callBack.setMonsterPointer(m);  
135     monster->setAnimationEndCallback(&callBack);  
136 }  
137  
138 scene::IAnimatedMeshSceneNode* Monster::getNode(){return monster;}  
139  
140 btRigidBody* Monster::getRigidBody(){return monsterRigidBody;}  
141  
142 int Monster::getVitality(){return vitality;}  
143  
144 void Monster::removeVitality(int vt){vitality-=vt;}  
145  
146 int Monster::getState(){return state;}  
147  
148 void Monster::setState(State st){state=st;}  
149  
150 void Monster::setRest(float r){resting=r;}  
151  
152 float Monster::getRest(){return resting;}  
153  
154 void Monster::setDestination(core::vector3df d){destination=d;}  
155  
156 core::vector3df Monster::getDestination(){return destination;}  
157  
158 void Monster::setForwardVector(core::vector3df destination)  
159 {  
160     /// Calculating unit vector and saving it as the forward vector of the  
161     /// monster.  
162     float magnitude=(sqrt(pow(destination.X - monster->getPosition().X, 2)+  
163     pow(destination.Z - monster->getPosition().Z, 2)));  
164  
165     /// Since the monster's forward vector only matters on the XZ plane, th  
166     /// e Y axis shall remain 0.  
167     forwardVect=core::vector3df((destination.X - monster->getPosition().X)/  
168     magnitude, 0, (destination.Z - monster->getPosition().Z)/magnitude);  
169 }  
170  
171 core::vector3df Monster::getForwardVector(){return forwardVect;}  
172  
173 void Monster::setDamageTexture(video::ITexture *damageTexture)  
174 {  
175     monster->setMaterialTexture(0, damageTexture);  
176     damageTextureOn=true;
```

```
173     damageTextureTimer=0.5;
174 }
175
176 void Monster::damageTextureCountDown(video::ITexture *texture, float dt)
177 {
178     if(damageTextureTimer<=0)
179     {
180         monster->setMaterialTexture(0, texture);
181         damageTextureOn=false;
182     }
183     else{damageTextureTimer-=1*dt;}
184 }
185
186 bool Monster::getDamageTextureOn(){return damageTextureOn;}
187
188 void Monster::moveMonster(float dt)
189 {
190     btVector3 frwd(forwardVect.X, forwardVect.Y, forwardVect.Z);
191     monsterRigidBody->setLinearVelocity(frwd*velocity);
192 }
193
194 void Monster::SynchPosition()
195 {
196     core::vector3df edges=monster->getTransformedBoundingBox().getExtent();
197
198     // Set position.
199     btVector3 point=monsterRigidBody->getCenterOfMassPosition();
200
201     monster->setPosition(core::vector3df((f32)point[0], (f32)point[1]-(edges.Y/2), (f32)point[2]));
202
203     // Set rigid body rotation.
204     btVector3 forward=monsterRigidBody->getLinearVelocity();
205
206     float rot=atan2(forward.z(), forward.x())*core::RADTODEG;
207
208     // Set graphics body rotation.
209     monster->setRotation(core::vector3df(0, 90-rot, 0));
210 }
211 }
```

```
1 #ifndef INCLUDE_MANAGER_CLASS
2 #define INCLUDE_MANAGER_CLASS
3
4 #include <iostream>
5 #include <irrlicht.h>
6 #include "fireworks.h"
7
8 using namespace std;
9 using namespace irr;
10
11 class Manager
12 {
13     public:
14     Manager();
15     ~Manager();
16
17     void addFirework(IrrlichtDevice *device, FireworkInfo info, btDynamicsWo
rld *world,
18                     video::ITexture *texture, core::vector3df pos, core::ve
ctor3df dir, float thrwStr);
19     void update(float dt);
20     btRigidBody *redrawFireworks(int i);
21
22     private:
23     Fireworks *fireworks[100];
24 };
25
26 #endif
27
```

```
1 #include "manager.h"
2
3 Manager::Manager(){for(int i=0; i<100; i++){fireworks[i]=NULL;}}
4 Manager::~Manager(){}
5
6 void Manager::addFirework(IrrlichtDevice *device, FireworkInfo info, btDynam
icsWorld *world, video::ITexture *texture, core::vector3df pos, core::vector
3df dir, float thrwStr)
7 {
8     for(int i=0; i<100; i++)
9     {
10         if(fireworks[i]==NULL)
11         {
12             fireworks[i]=new Fireworks(device, info, world, texture, pos, di
r, thrwStr);
13             break;
14         }
15     }
16 }
17
18 void Manager::update(float dt)
19 {
20     for(int i=0; i<100; i++)
21     {
22         if(fireworks[i]!=NULL)
23         {
24             FireworkState fs=fireworks[i]->getState();
25
26             switch(fs)
27             {
28                 case FLYING:
29 //                     fireworks[i]->checkHeight();
30 //                     cout << "moving state!\n";
31 //                     break;
32
33                 case FIREWORK:
34 //                     fireworks[i]->countDown(dt);
35 //                     cout << "firework state!\n";
36 //                     break;
37
38                 case CLEANUP:
39                     delete fireworks[i];
40                     fireworks[i]=NULL;
41 //                     cout << "cleanup state!\n";
42 //                     break;
43
44                 default:
45                     break;
46             }
47         }
48     }
49 }
50
51 btRigidBody* Manager::redrawFireworks(int i)
52 {
53     if(fireworks[i]!=NULL){return fireworks[i]->getRigidBody();}
54     else{return 0;}
55 }
56
```

```
1 #ifndef INCLUDE_FIREWORKS_CLASS
2 #define INCLUDE_FIREWORKS_CLASS
3
4 #include <iostream>
5 #include <irrlicht.h>
6 #include <btBulletCollisionCommon.h>
7 #include <btBulletDynamicsCommon.h>
8 #include "collision_filter.h"
9
10 using namespace std;
11 using namespace irr;
12
13 enum FireworkState{FLYING=0, FIREWORK=1, CLEANUP=2};
14
15 struct FireworkInfo
16 {
17
18     int minPart, maxPart;
19     video::SColor minColor, maxColor;
20     int minLifetime, maxLifetime;
21     core::dimension2df minSize, maxSize;
22     int angle;
23     video::ITexture *particleTexture;
24
25     FireworkInfo(){}
26     FireworkInfo(int minP, int maxP, video::SColor minC, video::SColor maxC,
27                  int minL, int maxL, core::dimension2df minS, core::dimension2df maxS,
28                  int a, video::ITexture *texture)
29     {
30         minPart=minP;
31         maxPart=maxP;
32         minColor=minC;
33         maxColor=maxC;
34         minLifetime=minL;
35         maxLifetime=maxL;
36         minSize=minS;
37         maxSize=maxS;
38         angle=a;
39         particleTexture=texture;
40     }
41 };
42
43 class Fireworks
44 {
45     public:
46         Fireworks(IrrlichtDevice *device, FireworkInfo info, btDynamicsWorld *world,
47                    video::ITexture *texture, core::vector3df pos, core::vector3df dir, float thrwStr);
48         ~Fireworks();
49
50         FireworkState getState();
51         btRigidBody *getRigidBody();
52         void checkHeight();
53         void countDown(float dt);
54
55     private:
56         /// Binary state machine, state holder.
57         FireworkState state;
58         /// Collision filter.
59         int fireworkCollidesWith;
59         /// Current height of the shell object.
```

```
60     float height;
61     /// Firework burst.
62     float burst;
63     /// Duration of the firework before being destroyed.
64     float lifetime;
65     /// Forward direction for movement.
66     core::vector3df direction;
67     /// Graphics object representing the fireworks.
68     scene::IMeshSceneNode *shell;
69     /// Shell motion state.
70     btDefaultMotionState *motionState;
71     /// Shell rigid body.
72     btRigidBody *shellRigidBody;
73     /// Particle system's emitter.
74     scene::IParticlePointEmitter *emm;
75     /// Particle system.
76     scene::IParticleSystemSceneNode *parSys;
77 };
78
79 #endif
80
```

```
1 #include "fireworks.h"
2
3 Fireworks::Fireworks(IrrlichtDevice *device, FireworkInfo info, btDynamicsWorld *world, video::ITexture *texture, core::vector3df pos, core::vector3df dir, float thrwStr):
4 lifetime(5), burst(0.2)
5 {
6     /// Creating firework sphere shell.
7     shell=device->getSceneManager()->addSphereSceneNode(10);
8     shell->setMaterialTexture(0, texture);
9     shell->setMaterialFlag(video::EMF_LIGHTING, false);
10    shell->setRotation(core::vector3df(0.0f, 0.0f, 0.0f));
11    shell->setPosition(pos);
12
13     /// Setting up rigid body for fireworks shell.
14     fireworkCollidesWith=WORLD_COLLISION|SCENIC_COLLISION;
15     btCollisionShape *shellShape=new btSphereShape(10);
16     btScalar mass=2;
17     btVector3 shellInertia;
18     shellShape->calculateLocalInertia(mass, shellInertia);
19     btVector3 initPos(shell->getPosition().X, shell->getPosition().Y, shell-
>getPosition().Z);
20     btTransform trans;
21     trans.setIdentity();
22     trans.setOrigin(initPos);
23     motionState=new btDefaultMotionState(trans);
24     btRigidBody::btRigidBodyConstructionInfo shellRigidBodyCI(mass, motionSt
ate, shellShape, shellInertia);
25     shellRigidBody=new btRigidBody(shellRigidBodyCI);
26     shellRigidBody->setUserPointer((void *)shell);
27     cout << "dir - X: " << dir.X << " Y: " << dir.Y << " Z: " << dir.Z << en
dl;
28     shellRigidBody->setLinearVelocity(btVector3(dir.X, dir.Y, dir.Z)*thrwStr
);
29     world->addRigidBody(shellRigidBody, FIREWORK_COLLISION, fireworkCollides
With);
30
31     /// Particle system setup.
32     parSys=device->getSceneManager()->addParticleSystemSceneNode();
33     emm=parSys->createPointEmitter(dir, info.minPart, info.maxPart, info.min
Color, info.maxColor,
34                                         info.minLifetime, info.maxLifetime, info.
angle, info.minSize, info.maxSize);
35     parSys->setMaterialFlag(video::EMF_LIGHTING, false);
36     parSys->setMaterialFlag(video::EMF_ZWRITE_ENABLE, false);
37     parSys->setMaterialTexture(0, info.particleTexture);
38 //     parSys->setMaterialType(video::EMT_TRANSPARENT_ADD_COLOR);
39
40     /// Particle gravity affector.
41     scene::IParticleGravityAffector *gravAff=parSys->createGravityAffector(c
ore::vector3df(0.0f, -10.0f, 0.0f), 10000);
42     parSys->addAffector(gravAff);
43     parSys->setEmitter(0);
44     /// Creating physics object and adding it to the world simulator.
45
46     /// Setting initial state and height.
47     state=FLYING;
48 }
49
50 Fireworks::~Fireworks()
51 {
52     parSys->setEmitter(0);
```

```
53     shell->remove();
54     parSys->remove();
55 }
56
57 FireworkState Fireworks::getState(){return state;}
58
59 void Fireworks::checkHeight()
60 {
61     int newHeight=shell->getPosition().Y;
62     if(height>newHeight)
63     {
64         parSys->setPosition(shell->getPosition());
65         shell->setVisible(false);
66         parSys->setEmitter(emm);
67         emm->drop(); // Deleting emitter since it's no longer needed.
68         state=FIREWORK;
69     }
70     else{height=newHeight;}
71 }
72
73 btRigidBody* Fireworks::getRigidBody(){return shellRigidBody;}
74
75 void Fireworks::countDown(float dt)
76 {
77     if(burst<=0){parSys->setEmitter(NULL);}
78     else{burst-=1*dt;}
79     lifetime<=0?state=CLEANUP:lifetime-=1*dt;
80 }
81
```

```
1 #ifndef INCLUDE_EVENT_CLASS
2 #define INCLUDE_EVENT_CLASS
3
4 #include <iostream>
5 #include <irrlicht.h>
6 #include "dummy.h"
7
8 using namespace std;
9 using namespace irr;
10
11 enum Mouse_Button{M_LEFT=KEY_LBUTTON, M_RIGHT=KEY_RBUTTON};
12
13 struct SAppContext
14 {
15     irr::IrrlichtDevice *device;
16 };
17
18 class EventReceiver:public irr::IEventReceiver
19 {
20     public:
21     EventReceiver(SAppContext &context);
22     ~EventReceiver();
23
24     virtual bool OnEvent(const irr::SEvent &event);
25     /// Return the state of a specified keyboard key.
26     virtual bool IsKeyDown(EKEY_CODE keyCode);
27     /// Return the action of a specified keyboard key.
28     int IsKeyPressed(EKEY_CODE keyCode);
29     /// Reset variable that holds keyboard key actions (must be called at the
30     /// end of the event loop in main.cpp)
31     void resetKeyPressed();
32     /// Return the state of the left mouse button.
33     bool IsMouseButtonDown(Mouse_Button button);
34     /// Return the state of the right mouse button.
35     int IsMouseButtonPressed(Mouse_Button button);
36
37     void resetMousePressed();
38
39     bool isCursorMoving();
40
41     void resetCursorMove();
42
43     private:
44     SAppContext &Context;
45     bool isKeyDown[KEY_KEY_CODES_COUNT];
46     int isKeyPressed[KEY_KEY_CODES_COUNT];
47     bool cursorMoving;
48     bool isLMouseDown, isRMouseDown;
49     int isLMousePressed, isRMousePressed;
50 };
51 #endif
52
```

```
1 #include "event_receiver.h"
2
3 EventReceiver::EventReceiver(S ApplicationContext &context):
4     Context(context), cursorMoving(false), isLMouseDown(false), isRMouseDown(false),
5     isLMousePressed(-1), isRMousePressed(-1)
6 {
7     for(u32 n=0; n<KEY_KEY_CODES_COUNT; ++n){isKeyDown[n]=false;}
8     for(u32 n=0; n<KEY_KEY_CODES_COUNT; ++n){isKeyPressed[n]=-1;}
9 }
10 EventReceiver::~EventReceiver(){}
11
12 bool EventReceiver::OnEvent(const SEvent& event)
13 {
14     // Remember whether each key is down or up
15     if(event.EventType==EET_KEY_INPUT_EVENT)
16     {
17         isKeyDown[event.KeyInput.Key]=event.KeyInput.PressedDown;
18         isKeyPressed[event.KeyInput.Key]=event.KeyInput.PressedDown;
19     }
20     if(event.EventType==EET_MOUSE_INPUT_EVENT)
21     {
22         switch(event.MouseInput.Event)
23         {
24             case EMIE_MOUSE_MOVED:
25             {
26                 cursorMoving=true;
27             }break;
28
29             case irr::EMIE_LMOUSE_PRESSED_DOWN:
30             {
31                 isLMouseDown=true;
32                 isLMousePressed=true;
33             }break;
34
35             case irr::EMIE_LMOUSE_LEFT_UP:
36             {
37                 isLMouseDown=false;
38                 isLMousePressed=false;
39             }break;
40
41             case irr::EMIE_RMOUSE_PRESSED_DOWN:
42             {
43                 isRMouseDown=true;
44                 isRMousePressed=true;
45             }break;
46
47             case irr::EMIE_RMOUSE_LEFT_UP:
48             {
49                 isRMouseDown=false;
50                 isRMousePressed=false;
51             }break;
52
53             default:
54             break;
55         }
56     }
57     return false;
58 }
59
60 bool EventReceiver::IsKeyDown(EKEY_CODE keyCode){return isKeyDown[keyCode];}
61
```

```
62 int EventReceiver::IsKeyPressed(EKEY_CODE keyCode){return isKeyPressed[keyCode];}
63
64 void EventReceiver::resetKeyPressed(){for(u32 n=0; n<KEY_KEY_CODES_COUNT; ++n)isKeyPressed[n]=-1;}
65
66 bool EventReceiver::IsMouseButtonDown(Mouse_Button button)
67 {
68     if(button==M_LEFT){return isLMouseDown;}
69     else{return isRMouseDown;}
70 }
71
72 int EventReceiver::IsMouseButtonPressed(Mouse_Button button)
73 {
74     if(button==M_LEFT){return isLMousePressed;}
75     else{return isRMousePressed;}
76 }
77
78 void EventReceiver::resetMousePressed()
79 {
80     isLMousePressed=-1;
81     isRMousePressed=-1;
82 }
83
84 bool EventReceiver::isCursorMoving(){return cursorMoving;}
85
86 void EventReceiver::resetCursorMove(){cursorMoving=false;}
87
```

```
1 #ifndef INCLUDE_DUMMY_CLASS
2 #define INCLUDE_DUMMY_CLASS
3
4 #include <iostream>
5 #include <math.h>
6 #include <irrlicht.h>
7 #include <btBulletCollisionCommon.h>
8 #include <btBulletDynamicsCommon.h>
9 #include "collision_filter.h"
10 using namespace std;
11 using namespace irr;
12
13 enum Direction{FORWARD=0, LEFT=1, BACKWARDS=2, RIGHT=4};
14 enum Type{TYPE01=0, TYPE02=1, TYPE03=2, TYPE04=3};
15
16 core::vector3df Cross(core::vector3df a, core::vector3df b);
17
18 struct Inventory
19 {
20     int fwType[4];
21     void setAll(int n){for(int i=0; i<4; i++){fwType[i]=n;}}
22     void setFw(int f, int n){fwType[f]=n;}
23 };
24
25 class Player
26 {
27     public:
28         Player(IrrlichtDevice *device, btDynamicsWorld *world, scene::IMesh *mesh,
29                 video::ITexture *texture, core::vector3df forwardVector, core::vector3df upVector);
30         ~Player();
31
32     /// Set player arms.
33     void setArms(scene::IAnimatedMeshSceneNode *leftArm, core::vector2d leftIdle, scene::IAnimatedMeshSceneNode *rightArm, core::vector2d rightIdle);
34
35     /// get player's mesh node.
36     scene::IMeshSceneNode *getNode();
37
38     /// get the player's rigid body.
39     btRigidBody *getPlayerRigidBody();
40
41     /// get the player's camera object.
42     scene::ICameraSceneNode *getCamera();
43
44     /// get camera forward vector
45     core::vector3df getForwardVector();
46
47     /// set selected firework.
48     void setSelectedFW(Type t);
49
50     /// get selected firework.
51     int getSelectedFW();
52
53     /// Set value in inventory item. !!! Probably wont be necessary !!!
54     void setInventoryItem(int f, int n);
55
56     /// Get value in inventory item.
57     int getInventoryItemValue(int f);
```

```
59     /// This function updates the camera and player rotation as well as repositioning the camera and player vectors.
60     void pRotation(core::vector2df cursorDelta);
61
62     /// This function is used to move the player through the map.
63     void pMovement(f32 dt, Direction d);
64
65     /// update player's camera target to reposition the forward vector.
66     void cUpdateTarget();
67
68     /// Update player camera target.
69     void pUpdate();
70
71     float getHealth();
72     void setHealth(float h);
73
74     private:
75     /// Irrlicht player scene node.
76     scene::IMeshSceneNode *player;
77     /// Animated left arm.
78     scene::IAnimatedMeshSceneNode *leftArm;
79     /// Animated right arm.
80     scene::IAnimatedMeshSceneNode *rightArm;
81     /// Bullet rigid body pointer to player rigid body.
82     btRigidBody *btPlayer;
83     /// Rigid body motion state.
84     btDefaultMotionState *playerMotionState;
85     /// Bullet transform used to rotate and translate rigis body.
86     btTransform trans;
87     /// Irrlicht Camera scene node.
88     scene::ICameraSceneNode *camera;
89     /// Temporary node for visualizing camera.
90     scene::IMeshSceneNode *cameraReference;
91     /// boolean keeps track of the player's attacking action.
92     bool attacking;
93     /// Movement speed.
94     float velocity;
95     /// Player's health.
96     float health;
97     /// Inventory.
98     Inventory invent;
99     /// Selected firework type.
100    Type selectedType;
101    /// forward, right and up vectors for the player.
102    core::vector3df vForward, vRight, vUp;
103    /// forward, right and up vectors for the camera.
104    core::vector3df vCamForward, vCamRight, vCamUp;
105 };
106
107 #endif
108 }
```

```
1 #include "dummy.h"
2
3 /// Function that puts the cross product of two vectors.
4 core::vector3df Cross(core::vector3df a, core::vector3df b)
5 {
6     core::vector3df c(core::vector3df(a.Y*b.Z-a.Z*b.Y, a.Z*b.X-a.X*b.Z, a.X
7 *b.Y-b.X*a.Y));
8     return c;
9 }
10
11 scene::IMeshSceneNode* Player::getNode(){return player;}
12 btRigidBody* Player::getPlayerRigidBody(){return btPlayer;}
13
14 scene::ICameraSceneNode* Player::getCamera(){return camera;}
15
16 core::vector3df Player::getForwardVector()
17 {
18     float magnitude=sqrt(pow(vCamForward.X, 2)+pow(vCamForward.Y, 2)+pow(vC
19 amForward.Z, 2));
20     core::vector3df vector=vCamForward/magnitude;
21     return vector;
22 }
23
24 void Player::setSelectedFW(Type t){selectedType=t;}
25
26 int Player::getSelectedFW(){return selectedType;}
27
28 void Player::setInventoryItem(int f, int n){invent.setFw(f, n);}
29
30 int Player::getInventoryItemValue(int f){return invent.fwType[f];}
31
32 Player::Player(IrrlichtDevice *device, btDynamicsWorld *world, scene::IMesh
33 *mesh, video::ITexture *texture,
34         core::vector3df forwardVector, core::vector3df upVector):vel
35 ocity(100), health(100)
36 {
37     /// Setting up player graphics object.
38     player=device->getSceneManager()->addMeshSceneNode(mesh);
39     player->setMaterialTexture(0, texture);
40     player->setMaterialFlag(video::EMF_LIGHTING, false);
41
42     /// Player scale and initial position.
43     player->setPosition(core::vector3df(2500.0f, 400.0f, 2500.0f));
44
45     /// Initializing selected firework type and firework types amount.
46     selectedType=TYPE01;
47
48     /// Pointer to bullet (player) rigid body.
49     invent.setAll(0);
50
51     /// Creating dummy collision body.
52     int playerCollidesWith=WORLD_COLLISION|SCENIC_COLLISION;
53     core::aabbox3df boundingBox=player->getTransformedBoundingBox();
54     core::vector3d<f32> edges=player->getTransformedBoundingBox().getExtent
55     ();
56     btVector3 btPlayerScale(edges.X*0.5f, edges.Y*0.5f, edges.Z*0.5f);
57     btCollisionShape *playerShape=new btBoxShape(btPlayerScale);
58     btScalar mass=1;
59     btVector3 playerInertia;
60     playerShape->calculateLocalInertia(mass, playerInertia);
```

```
58     /// Setting initial position
59     core::vector3df position=player->getPosition();
60     btVector3 btPosition(position.X, position.Y, position.Z);
61     btTransform playerTrans;
62     playerTrans.setIdentity();
63     playerTrans.setOrigin(btPosition);
64     playerMotionState=new btDefaultMotionState(playerTrans);
65
66     /// RigidBody setup.
67     btRigidBody::btRigidBodyConstructionInfo playerRigidBodyCI(mass, player
MotionState, playerShape, playerInertia);
68     btPlayer=new btRigidBody(playerRigidBodyCI);
69     btPlayer->setAngularFactor(btVector3(0.0f, 0.0f, 0.0f));
70
71     btPlayer->setUserPointer((void *)player);
72     btPlayer->setActivationState(DISABLE_DEACTIVATION);
73
74     /// Adding rigid body to the world.
75     world->addRigidBody(btPlayer, PLAYER_COLLISION, playerCollidesWith);
76
77     /// (Bullet Physics) Initializing transform that will hold player's pos
ition and rotation in the world.
78     trans.setIdentity();
79     trans=btPlayer->getCenterOfMassTransform();
80
81     /// Creating a new camera and attaching it to the dummy.
82     camera=device->getSceneManager()->addCameraSceneNode(player, core::vect
or3df(player->getAbsolutePosition().X, player->getAbsolutePosition().Y+5.0f
,
83
84         player->getAbsolutePosition().Z));
85     camera->setTarget(core::vector3df(camera->getAbsolutePosition().X*50.0f
, 0.0f, 0.0f));
86     camera->setFarValue(10000);
87
88     /// Creating a node in order to see where the camera attached to the du
mmy is placed.
89     cameraReference=device->getSceneManager()->addSphereSceneNode(20, 32, c
amera);
90     cameraReference->setMaterialTexture(0, device->getVideoDriver()->getTex
ture("./assets/EpikCubeTexture"));
91     cameraReference->setMaterialFlag(video::EMF_LIGHTING, false);
92
93     /// Setting intial values for the player vectors.
94     vForward=forwardVector;
95     vUp=upVector;
96     vRight=Cross(vForward, vUp);
97
98     /// Setting intial values for the camera vectors.
99     vCamForward=camera->getTarget();
100    vCamUp=vUp;
101    vRight=Cross(vCamForward, vCamUp);
102 }
103 Player::~Player(){}
104
105 void Player::pRotation(core::vector2df cursorDelta)
106 {
107     core::vector2df vectorXZ(sin(cursorDelta.Y), cos(cursorDelta.Y));
108     trans=btPlayer->getCenterOfMassTransform();
109
110     /// Rotating player rigid body.
```

```
111     btQuaternion btPlayerRot=btPlayer->getOrientation();
112     btQuaternion pRotation;
113     pRotation.setEulerZYX(0.0f, core::DEGTORAD*cursorDelta.X, 0.0f);
114     btPlayerRot*=pRotation;
115     trans.setRotation(btPlayerRot);
116     btPlayer->setCenterOfMassTransform(trans);
117
118     // Rotating camera node.
119     camera->setRotation(core::vector3df(0.0f, 0.0f, camera->getRotation().Z
120 -cursorDelta.Y));
121     float rotY=player->getRotation().Y*core::DEGTORAD;
122     float rotZ=player->getRotation().Z*core::DEGTORAD;
123
124     // Repositioning player's forward vector.
125     float rot=rotZ+rotY*cos(rotZ);
126     // float Rot=
127     vForward.X=cos(rot);
128     vForward.Z=-sin(rot);
129
130     // Calculating player's right vector.
131     vRight=Cross(vForward, vUp);
132
133     // Repositioning camera's forward vector.
134     vCamForward.X=vForward.X*cos(camera->getRotation().Z/360*2*3.1416/180)*
50;
135     vCamForward.Z=vForward.Z*cos(camera->getRotation().Z/360*2*3.1416/180)*
50;
136     vCamForward.Y=sin(camera->getRotation().Z/360*2*3.1416)*50;
137
138     // Calculating camera's right vector.
139     vCamRight=Cross(vCamForward, vUp);
140     // Calculating camera's up vector.
141     vCamUp=Cross(vCamRight, vCamForward)/50;
142 }
143 void Player::pMovement(f32 dt, Direction d)
144 {
145     btVector3 frwd(vForward.X, vForward.Y, vForward.Z);
146     btVector3 right(vRight.X, vRight.Y, vRight.Z);
147
148     switch(d)
149     {
150         case FORWARD:
151             trans.setOrigin(btPlayer->getCenterOfMassTransform().getOrigin()+
fr
152 wrd*velocity*dt);
153             break;
154
155         case RIGHT:
156             trans.setOrigin(btPlayer->getCenterOfMassTransform().getOrigin()-ri
ght*velocity*dt);
157             break;
158
159         case BACKWARDS:
160             trans.setOrigin(btPlayer->getCenterOfMassTransform().getOrigin()-fr
wrd*velocity*dt);
161             break;
162
163         case LEFT:
164             trans.setOrigin(btPlayer->getCenterOfMassTransform().getOrigin()+
ri
ght*velocity*dt);
165             break;
166     }
```

```
166     /// updating rigid body's positions.  
167     btPlayer->setCenterOfMassTransform(trans);  
168 }  
169  
170 void Player::cUpdateTarget(){camera->setTarget(camera->getAbsolutePosition()  
171 )+vCamForward);}  
172 float Player::getHealth(){return health;}  
173 void Player::setHealth(float h){health=h;}  
174
```

```
1 #include <irrlicht.h>
2 #include <btBulletCollisionCommon.h>
3 #include <btBulletDynamicsCommon.h>
4
5 class DebugDraw : public btIDebugDraw
6 {
7     public:
8         DebugDraw(irr::IrrlichtDevice* const device):mode(DBG_NoDebug), driver(device->getVideoDriver()), logger(device->getLogger()){}
9
10        void drawLine(const btVector3& from, const btVector3& to, const btVector3& color)
11        {
12            //workaround to bullet's inconsistent debug colors which are either from 0.0 - 1.0 or from 0.0 - 255.0
13            irr::video::SColor newColor(255, (irr::u32)color[0], (irr::u32)color[1], (irr::u32)color[2]);
14            if (color[0] <= 1.0 && color[0] > 0.0)
15                newColor.setRed((irr::u32)(color[0]*255.0));
16            if (color[1] <= 1.0 && color[1] > 0.0)
17                newColor.setGreen((irr::u32)(color[1]*255.0));
18            if (color[2] <= 1.0 && color[2] > 0.0)
19                newColor.setBlue((irr::u32)(color[2]*255.0));
20
21            this->driver->draw3DLine(
22                irr::core::vector3df(from[0], from[1], from[2]),
23                irr::core::vector3df(to[0], to[1], to[2]),
24                newColor);
25        }
26
27        void drawContactPoint(const btVector3& PointOnB, const btVector3& normalOnB, btScalar distance, int lifeTime, const btVector3& color)
28        {
29            static const irr::video::SColor CONTACTPOINT_COLOR(255, 255, 255, 0); //bullet's are black :(
30
31            //    this->drawLine(PointOnB, PointOnB + normalOnB*distance, CONTACTPOINT_COLOR);
32            const btVector3 to(PointOnB + normalOnB*distance);
33
34            this->driver->draw3DLine(
35                irr::core::vector3df(PointOnB[0], PointOnB[1], PointOnB[2]),
36                irr::core::vector3df(to[0], to[1], to[2]),
37                CONTACTPOINT_COLOR);
38        }
39
40        void reportErrorWarning(const char* text)
41        {
42            this->logger->log(text, irr::ELL_ERROR);
43        }
44
45        void draw3dText(const btVector3& location, const char* text) { }
46        void setDebugMode(int mode) { this->mode = mode; }
47        int getDebugMode() const { return this->mode; }
48
49    private:
50        int mode;
51        irr::video::IVideoDriver* const driver;
52        irr::ILogger* logger;
53    };
54}
```

```
1 #ifndef COLLISION_FILTER
2 #define COLLISION_FILTER
3
4 #define BIT(x) (1<<(x))
5
6 enum CollisionFilters
7 {
8     NO_COLLISION=BIT(0),
9     WORLD_COLLISION=BIT(1),
10    FIREWORK_COLLISION=BIT(2),
11    PLAYER_COLLISION=BIT(3),
12    SCENIC_COLLISION=BIT(4),
13    MONSTER_COLLISION=BIT(5)
14 };
15
16 #endif
17
```