Mirandela, Junho de 2018

Relatório de Projeto

_Nameless Engine

A C++ Game Engine / Framework

Relatório de Projeto apresentado à Escola Superior de Comunicação, Administração e Turismo de Mirandela para obtenção do grau de licenciado em Design de Jogos Digitais.

ALUNO: **Mateus Valente Santos Silva Branco**
ORIENTADOR: **João Paulo Sousa**

Este relatório de Projeto não inclui as críticas e sugestões feitas pelo júri.

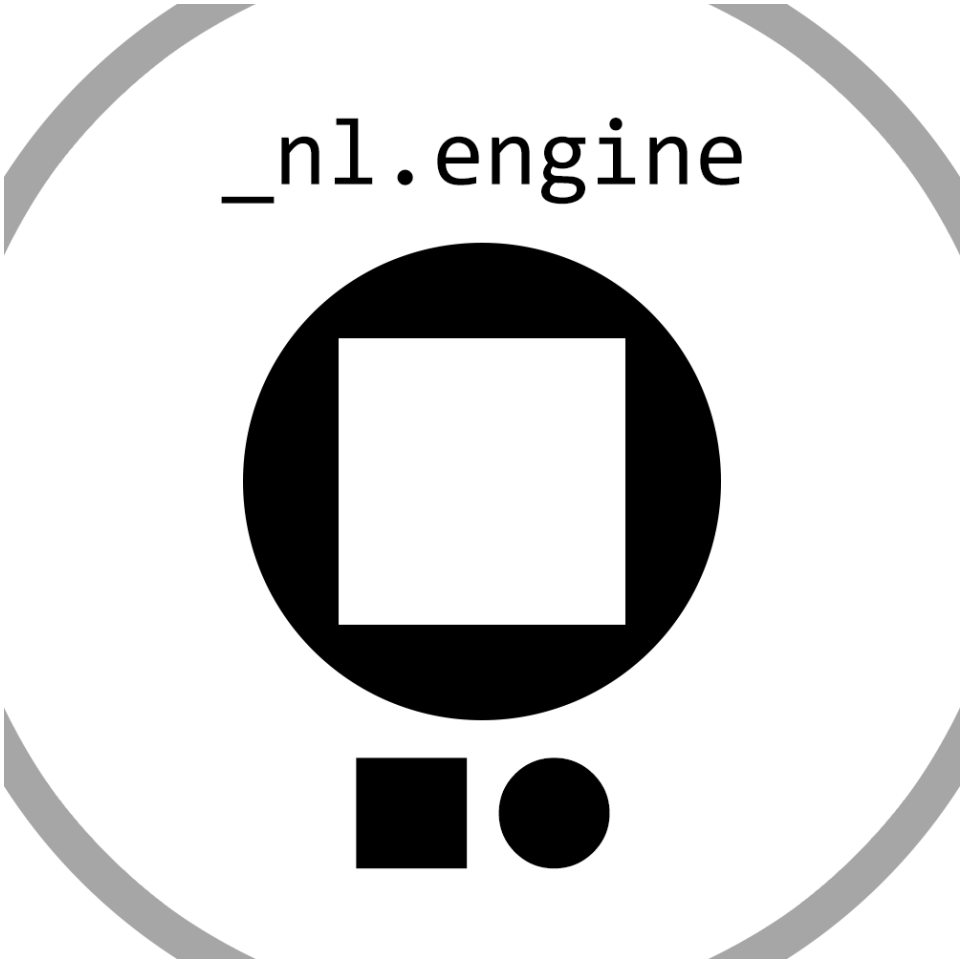
INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Comunicação, Administração e Turismo

Dedicatória

Dedico este projeto, aos meus gatos, á minha namorada, ao meu irmão e á minha irmã.

Agradecimentos

Agradeço a todos os que me ajudaram a testar e me deram feedback durante o desenvolvimento do Nameless Engine.



Resumo

Este relatório tem com intuito documentar a minha experiencia, como programador de software, relativa ao desenvolvimento do meu projeto : “Nameless Engine, a C++ Game Engine / Framework”.

Começarei por expor, de uma perspetiva mais abrangente, especificas fases do desenvolvimento do projeto, partindo de seguida para uma análise em profundidade de cada tópico. Isto de modo e transmitir ao leitor uma inicial intuição que ajudará na construção do mapa mental requerido para abordar consequentes tópicos e ideias.

Mais exatamente, como tópico inicial, decido abordar e elucidar razão por detrás da decisão de escolha da Licença de software, e arquitetura de software adotada para o motor de jogo.

Seguidamente tenciono abordar o desenvolvimento e metodologia de funcionamento dessa mesma arquitetura em função do motor de jogo e seus componentes e constituintes.

Após essa exposição, explico os paços tomados, momentos de refatoração de código, e imprevistos emergentes durante o desenvolvimento, até à obtenção de uma estrutura relativamente estável para a pipeline de renderização e ciclo de jogo.

Consequentemente parto para a exploração de detalhes técnicos relativos ao tópico de renderização, respetivamente: programas gráficos criados (glsl shaders), e suas condicionantes, de modo a adotar a pipeline de materiais fisicamente baseada (Physically Based Rendering); E a escolha de implementação das pipelines gráficas modernas “Deferred rendering” vs ”Forward Rendering” para a construção final da imagem.

Concluo o relatório com uma reflecção acerca da visão global do projeto.

Lista de abreviaturas / Siglas

IDE - Integrated development environment.

OpenGL – OpenGraphicsLibrary.

GLSL – OpenGL Shading Language.

PBR – Physically Based Rendering.

SFML – Simple and Fast Multimedia Library (SFML) v2.1.0.

GLEW – The OpenGL Extension Wrangler Library (GLEW) v2.4.2.

GLM – OpenGL Mathematics (GLM) v0.9.8.0.

GUI – Graphical User Interface.

GPU – Graphics processing unit.

CPU – Central processing unit.

MVP – Model-View-Projection

FXAA – Fast approximate anti-aliasing. (Anti-aliasing algorithm)

BRDF – Bidirectional reflectance distribution function.

BRDF_2D_LUT - BRDF 2D Look up Table.

TBN Matrix – Tangent Bitangent Normal Matrix

Índice

3 Ciclo de Jogo e Ciclo de Renderização

(…cont)

Luz e sombra

Deferred PBR Shader

4 Tech-Demo Notes

Key-Bindings:

Notas relativas ao Tech-Demo

5 List of Implemented Features

Currently Supported 3D Model Files:

3D Model Export specifications:

Índice de ilustrações

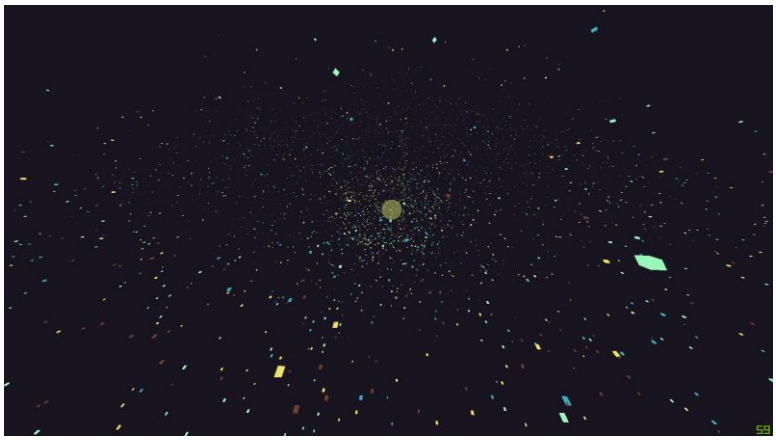
Introdução

Nameless Engine.

Durante a minha experiencia de três meses na Finlândia, como estudante no programa Erasmus, foi me possível dedicar a maior parte do meu tempo a desenvolver as minhas habilidades como programador numa multitude de projetos e aulas praticas.

Um projeto em específico, que considero crucial para a minha introdução no campo de programação gráfica, foi a implementação um pequeno motor gráfico, para uma cadeira de Física, usando C++ e OpenGL, com o intuito de simular a gravitação de corpos num sistema solar virtual. Para mim, a criação desse programa demostrou-se um verdadeiro desafio às minhas capacidades, visto que a minha experiencia com técnicas de programação gráficas era realmente limitada. No final, concluo que atingi um resultado não muito satisfatório, mas ao mesmo tempo foi uma aprendizagem fundamental para me preparar para futuros projetos.

Figura 0.1: Simulação de gravitação de corpos celestiais.



Fonte: KAIROS Engine – Solar System Demo (Mateus Branco).

Algum tempo depois da minha chegada a Portugal, a minha curiosidade pela área de computação gráfica e a minha vontade de programar continuavam em alta.

Neste momento estava também bastante investido na investigação de modelos de autómatos celulares e decidi então implementar o famoso "Conway's game of life" com a criação de um pequeno "motor gráfico de autómatos celulares", que me deixasse testar qualquer tipo de regras para autómatos celulares.

Figura 0.2: Implementação de Conway's game of life.

Fonte: Celular Automata Engine (Mateus Branco).

Inicialmente "Nameless Engine" teve a sua origem em mais um dos meus pequenos exercícios de programação, iniciado durante a segunda metade do mês de fevereiro de 2018. Inspirado num jogo da minha infância ("Deep Fighter", ano 2000), tencionava criar um simples motor de jogos com "gráficos retro", para jogos tridimensionais. Mas com a data de Pré-projecto a aproximar-se pensei: "porque não pegar nesta ideia e expandi-la para servir de projeto final de licenciatura?". Então assim o fiz.

Avançando 2 meses, estava eu pronto, no início do mês de Abril, com um plano bem estruturado para a continuação do desenvolvimento do Nameless Engine.

Com desenvolvimento deste motor de jogos, explorei em profundidade conceitos como: design de software, técnicas modernas

de computação gráfica, o funcionamento da Biblioteca gráfica OpenGL e linguagem GLSL, e amplifiquei os meus conhecimentos e habilidade na utilização da linguagem de programação C++.

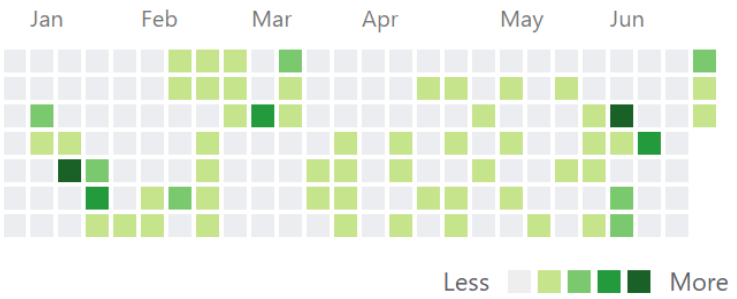
A minha linguagem de programação de escolha para este projeto foi C++ pois após ter experimentado uma multitude de linguagens, até ao dia de hoje, esta foi que mais me interessou estudar e desenvolver e também me permite exercer um nível de controlo e versatilidade adequado. Visto que programação gráfica é um trabalho que geralmente requer bastante manipulação de memória a um nível relativamente baixo, C++ pareceu-me uma das melhores opções para isso.

Ao longo do projeto tentei implementar prioritariamente funções e GLSL shaders da versão de OpenGL 4.5 pois tinha como objetivo testar e aprender a utilizar a versão mais recente da biblioteca OpenGL. Infelizmente na adoção desta posição sacrifiquei a compatibilidade do meu motor de jogo em máquinas com hardware que não tenham a versão de OpenGL 4.5 ou superior instalada.

Como IDE optei por usar "Visual Studio Community 2017" pois é o IDE que utilizo na maioria dos meus projetos.

Ao longo do desenvolvimento deste projeto aprendi também a utilizar a plataforma online de controlo de versões de código "GitHub", que utilizei para criar pontos de restauro quando necessário, deste modo assegurando a estabilidade da última versão do meu programa antes de cada implementação de novos componentes ou durante momentos refatoração de código.

Figura 0.3: GitHub - Uploads de código



Fonte: https://github.com/mattateusb7 (Mateus Branco).

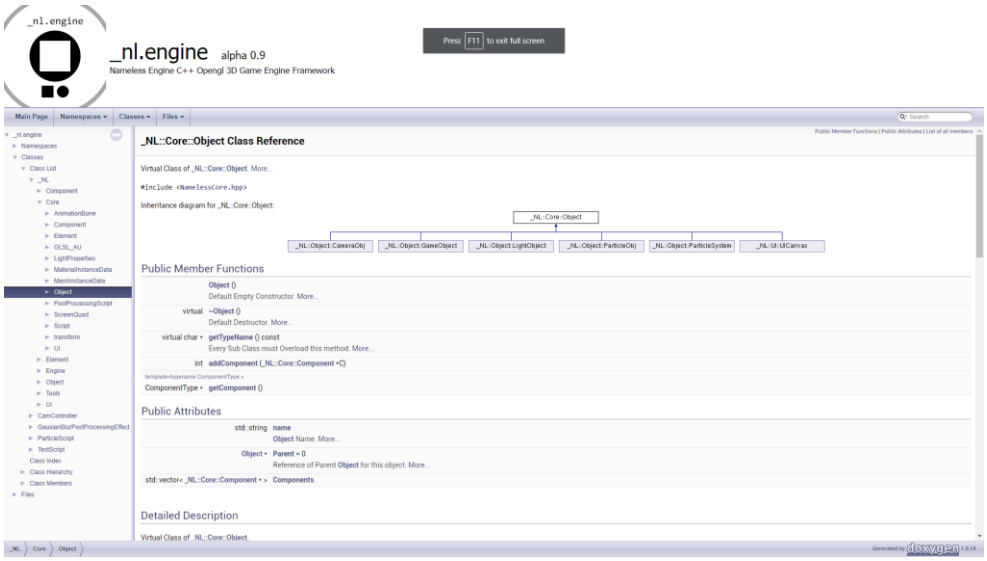
"GitHub" também oferece o serviço online "GitHubPages" que utilizei para a hospedagem de um pequeno website que criei, com o intuito de distribuir versões estáveis do produto ao público, nas fases de testing do software e uma vez que o projeto esteja concluído, e para a documentação do meu software.

Figura 0.4: Main Website

Fonte: https://mattateusb7.github.io/NamelessEngine-Framework (Mateus Branco).

A criação da documentação do Nameless Engine foi auxiliada pela utilização de um software livre chamado “Doxygen”, com suporte a uma grande variedade de linguagens de programação, que analisa o código de um projeto segundo uma sintaxe específica de marcação de comentários e automaticamente gera páginas HTML organizadas em modelo de website.

Figura 0.5: Documentation Website



Fonte:https://mattateusb7.github.io/NamelessEngine-Framework/documentation/

(Mateus Branco).

1

Configuração Inicial do Projeto

Licença de Software Livre

Partindo do início, antes de começar a trabalhar no motor de jogo, realizei uma pesquisa de modo a informar-me das várias possíveis licenças de software que poderia aplicar a este projeto. Visto que tencionava publicar este projeto online e disponibiliza-lo ao público de forma gratuita para download, modificação e publicação comercial, apenas necessitava uma licença que desse a liberdade para isso e que ao mesmo tempo assegurasse a preservação dos meus direitos de detentor de Copyright. Após ter percorrido um grande número de licenças, cheguei à conclusão que as cláusulas que mais se enquadravam nos meus planos eram os da Licença de software livre MIT.

Permissions:	***Limitations:***	***Conditions:***
• Commercial use • Modification • Distribution • Private use	• Liability • Warranty	• License and copyright notice

MIT License : A short and simple permissive license with conditions only requiring preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

(GitHub License descriptions).

Repositório online e Instalação de Bibliotecas Externas

Após o licenciamento do projeto e a criação de um repositório online, para o código do projeto, no website GitHub, abri o meu IDE de preferência, Visual Studio Community 2017, e criei um novo projeto.

Como já tinha feito a minha pesquisa na fase de pre-produção, comecei por instalar e conectar algumas das bibliotecas externas que iria necessitar para o projeto, nomeadamente SFML, GLEW e GLM.

Após isso comecei a desenvolver a base para a arquitetura do motor de jogo.

2

Arquitetura do Motor de Jogo

Interface por meio de Namespaces

Primeiro que tudo, decidi criar um ficheiro chamado "NamelessCore.hpp", que usei para a declaração dos principais tipos de objetos no motor de jogo (GameObject, Component, Element, etc…).

Isto foi conseguido com a declaração de Classes abstratas puras, que foram posteriormente herdados por subclasses de objeitos. Algumas funções estáticas, constantes e finalmente structs que servem como "pacotes de informação", também chamados de "primitivos", que auxiliassem na descrição de objetos de memoria importantes, como por exemplo a informação individual e descritiva presente em cada vértice de um modelo 3D.

Visto que não tencionava criar um interface gráfico (GUI) para o motor de jogo, recorri ao uso de "Namespaces" para servir de interface de Framework, de modo a identificar e organizar cada Classe do motor de jogo segundo os seguintes nomes:

- _NL – Engloba todos os Sub-namespaces relacionados com o motor de jogo.
    - _NL::Core – Contém tudo relacionado com o ficheiro NamelessCore.hpp.
    - _NL::Object – Contém subclasses da classe abstrata pura _NL::Core::Object e classes consideradas "objetos".
    - _NL::Component – Contém subclasses da classe abstrata pura _NL::Core::Component. Engloba classes de objetos que podem ser adicionados a listas de Componentes presentes em _NL::Core::Object's.
    - _NL::Element – Contém subclasses da classe abstrata pura _NL::Core::Element. Engloba classes de objetos independentes que habitualmente são referenciados por múltiplos Componentes.
    - _NL::UI – Contém subclasses da classe abstrata pura _NL::Core::UI e classes relacionadas com UI.
    - _NL::Engine – Engloba classes de objetos relacionadas com o funcionamento logístico do motor de jogo: gestão de janelas Windows, Relógios Internos, Áudio, e Ciclo de Jogo.
    - _NL::Tools – Engloba classes de objetos relacionadas com interações com ficheiros externos.

Reutilização de Código

Graças às minhas últimas experiencias com OpenGL, foi me possível reutilizar algum código de projetos antigos, depois de algumas modificações, para acelerar o inicio do projeto. Como exemplo,

reutilizei um parser com debugger para shaders glsl, como base para uma classe de objetos que auxiliasse o manuseamento de shaders no motor de jogos (instalação e debugging), e um parser de modelos 3D do tipo .OBJ, que mais tarde sofreu bastantes remodelações.

Janelas e Ciclo de Jogo

Baseando-me nos conhecimentos que tinha sobre motores de jogos sabia que um dos requerimentos básicos que necessitava de ter implementado consistia numa classe com a função de criar e destruir Janelas Windows, com especificações customizáveis, e com capacidade de gerir o Ciclo de jogo.

Posto isto criei a classe "_NL::Engine::NLManager" com esse mesmo objetivo. Esta classe instancia objetos da biblioteca externa SFML que neste caso auxiliam a gestão de Janelas e deteção de Input de periféricos (Rato e Teclado).

Nessa mesma classe criei alguns métodos que, quando "chamados", iniciam um ciclo de instruções que apenas termina com o fechamento da janela do programa.

O ciclo de jogo não é nada mais do que esse mesmo ciclo que, com a progressão e desenvolvimento do motor de jogo, se tornou cada vez mais complexa, com cada vez mais instruções.

Cenas de jogo / Níveis / Ambientes Virtuais

Nesta fase parti para a declaração de uma classe, "WorldSpace", que representa os ambientes virtuais / cenas de jogo, com a capacidade gerir listas de referências de objetos de jogo.

Esta Lista de objetos é multidimensional (tridimensional para ser exato). O primeiro eixo representa diferentes tipos de _NL::Core::Object, correspondentes ao retorno (tipo string) da função virtual pura "getTypeName()" em _NL::Core::Object. O segundo eixo guarda a referência de todos os objetos do mesmo tipo. Finalmente o terceiro eixo contém todas as referências de instâncias de objetos que

partilham uma referência com um mesmo _NL::Core::Component da subclasse "_NL::Component::MeshRenderer". Este terceiro eixo é importante de modo a possibilitar a observação do número de objetos que utilizam o mesmo modelo 3D e assim otimizar a renderização dos mesmos com a utilização de funções de OpenGL de instanciação GPU.

Uma Arquitetura Modular

Olhando para motores de jogos modernos, como exemplo "Unity Engnine" e "Unreal Engine", reparamos que a maior parte de "objetos de jogo", quando criados, começam por ser elementos bastante simples, contudo todos eles têm a capacidade de evoluir em termos de complexidade graças à incorporação de novos componentes. Como exemplo de alguns componentes temos posição no espaço, rotação e escala, o modelo 3D a si atribuído, texturas, Logica proveniente de scripts, etc…

Tendo esta ideia como base, e com a utilização de funções template e classes abstratas puras, foi me possível implementar um sistema de Componentes similar aos dos motores de jogos que me inspiraram.

Esse sistema funciona graças a subclasses de Objetos herdados da classe mãe _NL::Core::Object, que ganham a possibilidade de expandirem a sua complexidade com a adição de um número ilimitado de subclasses de Objetos herdados da classe _NL::Core::Component.

Para adicionar Componentes a objetos, é apenas necessário chamar a função "addComponent(_NL::Core::Component*)", de subclasses de objetos que herdem a classe abstrata pura _NL::Core::Object (ex.: _NL::Object::GameObject), usando uma referencia a um objecto de subclasses que herdem a classe abstrata pura _NL::Core::Component (ex.: _NL::Component::Transform).

Relativamente à questão de Scripting no motor de jogo desenvolvi uma abordagem que tecnicamente não requere a utilização de uma Linguagem Interpretada como Javascript, Python ou LUA, mas ao invés, utiliza C++ compilado antes de RunTime.

Para conseguir este efeito, criei uma classe template abstrata pura, _NL::Core::Script<Owner>, com 2 funções virtuais puras, "Start()" e "Update()", e uma função virtual "End()".

Conjuntamente criei também uma subclasse template da classe abstrata pura _NL::Core::Component, denominada _NL::Component::CppScript<ScriptType> com uma referencia a subclasses de objetos herdados da classe mãe _NL::Core::Script<Owner>.

A classe _NL::Core::Script<Owner>, necessita de ser template de modo a conter uma referência, "_this*", definida pelo tipo de objeto que pretende implementar os seus métodos virtuais puros, "Owner" , e que referencie o objeto que os pretende chamar para que consiga aceder também aos métodos do objeto quando executando o Script.

O processo anterior é automatizado pela classe _NL::Component::CppScript<ScriptType>, sendo que quando o utilizador deseja criar um novo componente script, apenas necessita de criar uma nova subclasse que herde a classe _NL::Core::Script<Owner>, e de seguida instanciar um novo objeto da classe _NL::Component::CppScript<ScriptType> substituindo o "ScriptType" pela nova classe criada.

Os Scripts dos objetos são executados automaticamente durante o ciclo de jogo.

- O método "_NL::Core::Script<Owner>::Start()" é executado quando o booleano "_NL::Core::Script<Owner>::Awake" é igual a 0, mudando o seu valor para 1.
- O método "_NL::Core::Script<Owner>::Update()" é executado todos os frames de jogo em que o booleano "_NL::Core::Script<Owner>::Awake" é igual a 1.

3

Ciclo de Jogo e Ciclo de Renderização

Mesh Renderer

NL::Component::MeshRenderer é a classe (componente) responsável pela definição e gestão de informação relativa a objetos de jogo com o propósito de serem renderizados nos ambientes virtuais de jogo (_NL::Engine::WorldSpace).

O ciclo de jogo, em maior profundidade

O ciclo de jogo, como referido no capítulo anterior, é gerido pela classe "_NL::Engine::NLManager", sendo definido por uma série de instruções ordenadas que passarei a descrever em maior pormenor:

Primeiramente a função _NL::Engine::NLManager::RunScene chamada, usualmente após a declaração de todos os objetos de jogo dentro da função principal do programa ("Main"), de modo a dar início ao ciclo de Jogo.

Esta função começa por inicializar algumas definições de OpenGL e após isso, entra num ciclo "while".

1. Este ciclo, o Ciclo de jogo, apenas retorna falso / termina caso a janela ativa é fechada ou o booleano "_NL::Engine::NLManager::bEndCurrentScene" é igual a 1.

2. Uma vez dentro do ciclo de jogo, a cada frame, este começa com iteração ao longo de todos os objetos da cena atual de jogo verificando se contém componentes do tipo _NL::Component::CppScript<ScriptType>, e executando os seus respetivos métodos de Start(), Update() ou End().
3. Depois disto, o método _NL::Engine::NLManager::UpdateParticleSystems é chamado e todos os Objetos do tipo _NL::Object::ParticleSystem executam as suas funções de atualização de partículas.
4. De seguida o método _NL::Engine::NLManager::UpdateObjectLists é chamado e todas as listas de referências de objetos para o frame atual, são atualizadas.
5. Se existir pelo menos um _NL::Object::CameraObj na cena atual, o ciclo de Renderização começa, com o chamamento do método NL::Engine::NLManager::RenderCurrentScene();
6. Após o frame atual ser apresentado no ecrã, o relógio de jogo é incrementado, e assim termina a lista de instruções, repetindo novamente partindo da primeira.

Ciclo de Renderização

No passo número 5 do ciclo de jogo, caso existir pelo menos um _NL::Object::CameraObj na cena atual, o ciclo de renderização é iniciado, seguindo também um número fixo de passos.

1. A função de OpenGL "glBindFramebuffer" é chamada, selecionando o Framebuffer padrão (ID 0) como alvo de renderização
2. A função de OpenGL "glClear" é chamada com as mascaras de bits: "GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT" de modo a limpar qualquer memória presente no Framebuffer selecionado (ID 0).

3. Todas as referências de luzes da cena atual são atualizadas e enviadas para um "GL_SHADER_STORAGE_BUFFER", para serem posteriormente utilizadas nos shaders de cálculo de luz
4. Para cada Câmara na cena atual, o seguinte número de instruções é executado de modo a renderizar toda a informação necessária para o frame atual:
    a. O Framebuffer secundário da câmara é selecionado como alvo de renderização.
    b. O Framebuffer selecionado é limpo com a função de OpenGL "glClear" usando as mesmas máscaras de bits anteriormente mencionadas.
    c. O Viewport de renderização é actualizado consoante as definições da Câmara atual.
    d. Se a Cena de jogo atual contiver uma Skybox válida e ativa, a Skybox é renderizada para o framebuffer selecionado.
    e. O Framebuffer Primário da câmara é selecionado como alvo de renderização.
    f. O Framebuffer selecionado é limpo com a função de OpenGL "glClear" usando as mesmas máscaras de bits anteriormente mencionadas.
    g. Nesta fase o Framebuffer Primário está preparado para renderizar todos os objetos da cena de jogo. De modo a otimizar esta fase, apenas objetos contendo os componentes _NL::Component::MeshRenderer e _NL::Component::Transform que forem considerados ativos e válidos serão contabilizados para serem enviados para a GPU nos programas de renderização glsl (shaders).
        1. Se o Objeto possuir um shader válido no seu componente _NL::Component::MeshRender, este por sua vez torna-se o shader activo para renderização.

2. Para cada instancia de objeto de tipo de objetos é calculado uma Matriz de Modelo (ModelMatrix) individual, que consiste numa matriz 4x4 contendo informação relativa a como transladar, escalar e rodar os vértices do seu respetivo modelo 3D segundo o seu componente _NL::Component::Transform e o de um possível Objeto Pai (Object parenting).

3. Após isto, a Matriz de Modelo é enviada para o programa glsl (shader), justamente com a Matriz de Visão (ViewMatrix) e a Matriz de Projeção (ProjectionMatrix). A Matriz de Visão e projeção é construída pela câmara ativa. Tudo isto para ser possível contruir o MVP Matrix no Shader.

4. Se o objeto possuir texturas, estas são também enviadas para o shader.

5. Finalmente, a Skybox correspondente à cena de jogo atual envia também para o shader texturas necessárias para o cálculo de luz ambiente e reflexão da skybox.

6. Agora que toda a informação possível foi enviada para o shader, é chamada a função "glDrawArraysInstanced()" que utiliza o shader activo para desenhar todas as instancias de objetos do tipo de objeto específico.

7. Após isto o shader ativo e texturas ativas são desativados, de modo a receber informação do próximo tipo de objeto.

h. Depois de todos os objetos da cena atual serem renderizados, a imagem final de frame é construída e enviada para o "stack" de pós processamento de imagem (Post-Processing Stack) da camara. Este "stack" executa um número pré-especificado de shaders instalados na camera que modificam / filtram a

imagem a ser apresentada, de modo a atingir os efeitos gráficos pretendidos.

   i. Depois de obter a imagem pretendida, esta é enviada para o monitor, apenas sendo descartada no inicio do próximo frame.

5. Depois de todas as camaras enviarem as suas imagens para o ecrã, todos os objetos _NL::UI::Canvas desenham todos os elementos dependentes por cima de tudo o anteriormente "desenhado".
6. Concluindo assim o ciclo de renderização com a apresentação do frame final no ecrã.

Pipeline de materiais PBR

Para este motor de jogos decidi optar por uma pipeline de materiais relativamente moderna (PBR), pois tencionava disponibilizar a possibilidade de atingir realismo gráfico em termos de texturas e iluminação, e também porque é uma pipeline bastante usada e efetivamente popular entre modeladores 3D.

A pipeline PBR consiste numa representação fisicamente baseada no real para a descrição de materiais no virtual.

Esta representação é transcrita por meio de fórmulas matemáticas que interagem com os valores de um conjunto de texturas.

A minha implementação utiliza 5 texturas por material, nomeadamente:

1. Mapa de Albedo:
   - Cor difusa (sem sombras) do material.
2. Mapa de Rugosidade:
   - Define a rugosidade do material.
   - Valores de 0 representam um material com uma superfície polida e refletora.

- Valores de 1 representam um material com uma superfície rugosa e não refletora.

3. Mapa de Metalicidade:
    - Define o quão metálico o material é.
    - Valores de 0 representam um material dielétrico (ex.: Plástico).
    - Valores de 0.5 representam um material semicondutor (ex.: Silício)
    - Valores de 1 representam um material metálico (ex.: Alumínio)
4. Mapa Normal:
    - Define a direção do vetor normal (perpendicular) à superfície.
5. Mapa de Oclusão de Ambiente:
    - Define a percentagem de oclusão da luz ambiente na superfície.
    - Valores de 0 representam 100% de oclusão de luz ambiente.
    - Valores de 1 representam 0% de oclusão de luz ambiente.

Figura 3.1: Resultado Final *Fonte: _NLEngine (Mateus Branco)*

Figura 3.2: Albedo | Figura 3.3: Rugosidade

Fonte: http://artisaverb.info/PBT.html

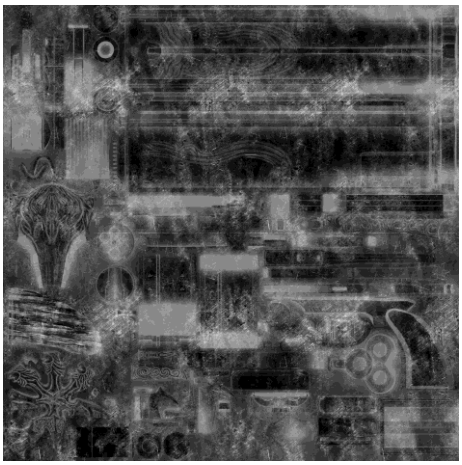
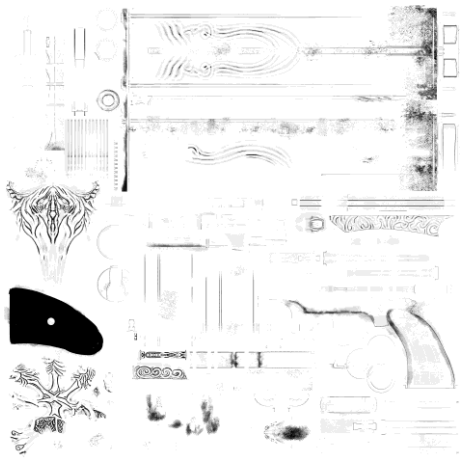


Figura 3.4: Metalicidade | Figura 3.5: Normal

Fonte: http://artisaverb.info/PBT.html



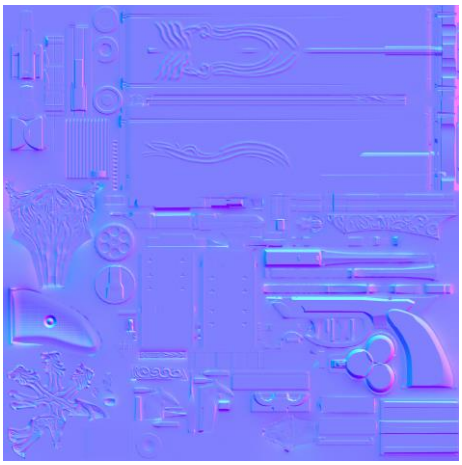
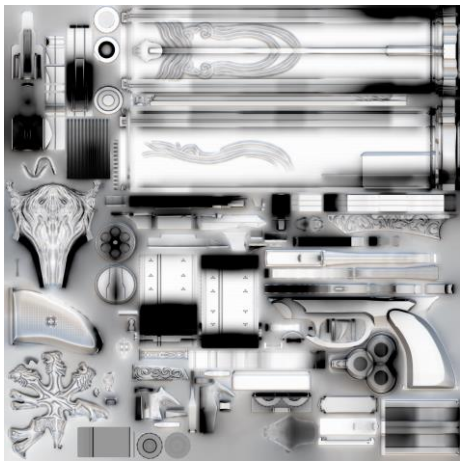


Figura 3.6: Oclusão de Ambiente | Fonte: http://artisaverb.info/PBT.html



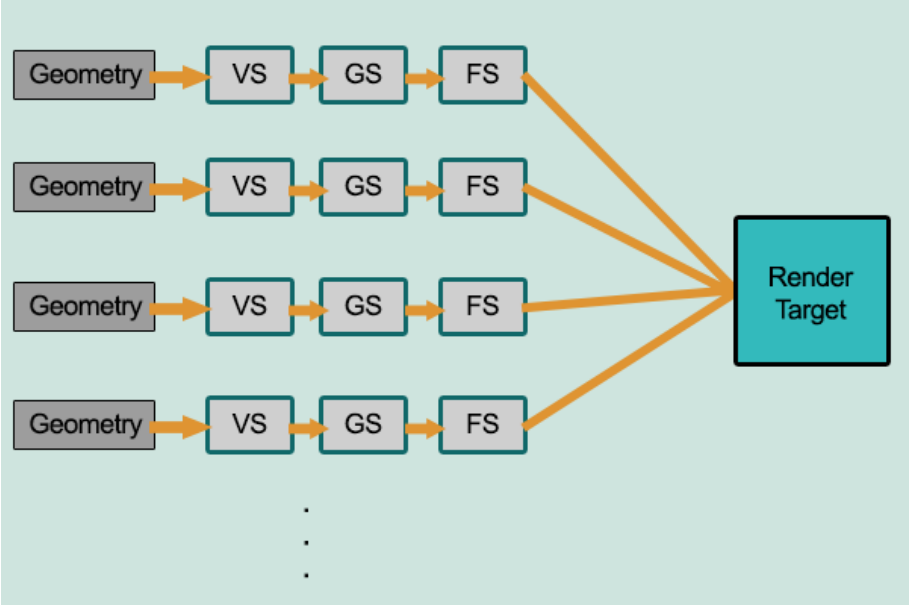
Durante o desenvolvimento do motor de jogo, os meus conhecimentos acerca de técnicas de computação gráfica foram evoluindo e com isso, ideias e planos iniciais mal formadas tornaram-se obstáculos.

Deparei-me com uma destas adversidades quando comecei a introduzir elementos de Iluminação no motor de jogo. Inicialmente tencionava utilizar a pipeline de gráficos "Forward Rendering" mas com a tentativa da introdução de um grande numero de luzes em tempo real verifiquei que, nesta situação, a utilização desta pipeline não é de facto uma abordagem tão eficiente como a utilização de "Deferred Rendering".

A maior contraste mais entre Deferred Rendering e Forward Rendering é o facto de que, começando pela pipeline de Forward Rendering, toda a geometria passa por um "Vertex Shader", "Geometry Shader" e "Fragment Shader" e toda a informação é calculada na sua totalidade e enviada para um único final Render Target.

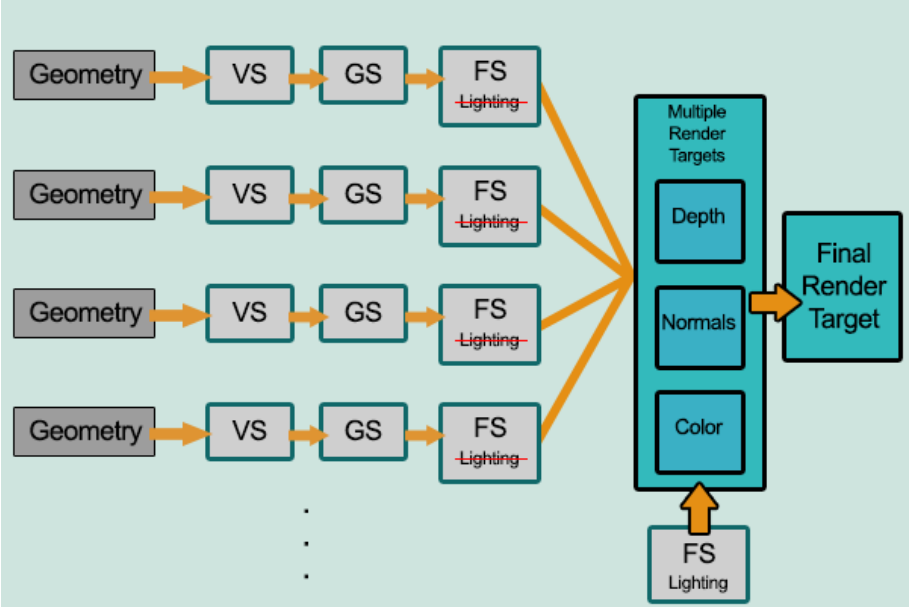
Ao invés, na Deferred Rendering Pipeleine, após a passagem pelos shaders iniciais, é gerado um conjunto de Render Targets que guardam diversos Render Targets divido essa informação calculada correspondente à geometria. Depois disto esta informação é devidamente enviada para um "Fragment Shader" final onde de dá a composição da imagem previamente calculada e os cálculos de luz e sombra "per pixel" em vez de "per fragment", aumentando a sua eficiência.

Figura 3.7: Forward Rendering Pipeline Diagram



Font: https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342

Figura 3.8: Deferred Rendering Pipeline Diagram



Font: https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342

Se eu quisesse, seria possível manter a pipeline original mas teria que recorrer à implementação de mapas de luzes pré-calculados, que também não estava nos meus planos.

Outro fator dependente a ter em conta consistia na escolha de uma das diversas técnicas existentes para o método de projeção de sombras que também dependeria do tipo de pipeline que escolhesse.

Após algum tempo e pesquisa, cheguei á conclusão que a melhor abordagem para atingir os meus objetivos, tanto para a utilização de luzes dinâmicas como para a projeção de sombras e controlo sob a imagem (Post-processing), seria a opção de Deferred Render Pipeline.

Sendo assim, decidi remodelar a minha pipeline gráfica de Forward Rendering para Deferred Rendering a meio do projeto.

Pouco tempo depois as remodelações foram completadas com sucesso, com a acomodação de certos problemas e o aparecimento de outras complicações.

Como exemplo mais relevante temos a opção de standard Multisampling e Anti-aliasing que tinha anteriormente implementado. Acabou por ser removido pois deixou de ser compatível com a nova pipeline. Uma das formas de contrariar este problema seria a implementação de um shader de post-processing de FXAA.

Luz e sombra

Após as remodelações, consegui então implementar 3 tipos de Luzes:

1. Point-Light: Luz com coordenadas no espaço relevantes, controlo de intensidade e Cor da Luz.
2. Directonal-Light: Luz sem coordenadas no espaço relevantes, controlo de intensidade, Cor e direção da Luz.
3. Spot-Light: Luz com coordenadas no espaço relevantes, controlo de intensidade, Cor, direção e angulo mínimo e máximo da Luz.

Avançando para o final do projeto, as sombras acabaram por não ser implementadas, infelizmente, por falta de tempo.

Deferred PBR Shader

Ao longo do desenvolvimento do motor de jogo, fui programando diversos shaders de modo a processar e apresentar informação necessária para cada frame.

Foquei-me principalmente na construção de um "Default Shader" com a capacidade de receber qualquer modelo 3D, ler a sua respetiva posição, rotação e escala, iluminação incidente proveniente de luzes individuais e ambiente (skybox) e o seu material atribuído de modo a projeta-lo no espaço de forma previsível.

De modo a acomodar a pipeline de materiais PBR, necessitei de fornecer diversas texturas ao PBR shader, nomeadamente:

- Material do objeto:
    - 2DTex – AlbedoTexture;
    - 2DTex – RoughnessTexture;
    - 2DTex – MetalnessTexture;
    - 2DTex – NormalTexture;
    - 2DTex – AmbientOculusionTexture;
- Skybox da cena actual de jogo:
    - CubeTex – AmbientIrradianceTexture;
    - CubeTex – PreFilterTexture;
    - 2DTex – BRDF_2D_LUTTexture;

Todas estas texturas são necessárias para o cálculo de diversos parâmetros dentro do PBR shader.

As texturas do Material são provenientes de ficheiros de imagem carregados por meio da ferramenta _NL::Tool::TextureLoader e conectados aso Elemento _NL::Element::MaterialInstance, que por

sua vês define o material no componente _NL::Component::MeshRenderer.

As texturas provenientes da Skybox são pré-calculadas com o chamamento do método _NL::Object::Skybox::createEnvironment() e _NL::Object::Skybox::createSkybox().

O PBR Shader consiste em 2 Vertex Shaders e 2 Fragment Shaders.

- *VertexShader 1: "Ddeferedvertexshader.glsl"*
- *Fragment Shader 1: "Ddeferedfragmentshader.glsl"*
- *VertexShader 2: "DdeferedScreenQuadvertexshader.glsl"*
- *Fragment Shader 2: "DdeferedScreenQuadfragmentshader.glsl"*

O primeiro VertexSahder é executado durante o ciclo de renderização e recebe a seguinte informação respetiva ao modelo 3D:

IN:

- *Vetor3: Posição dos vértices;*
- *Vetor3: Vetor Normal dos Vértices;*
- *Vetor3: Vetor Tangente dos Vértices;*
- *Vetor2: Coordenadas de Textura dos Vértices;*
- *Matriz4x4: Matriz Modelo da instância objeto;*
- *Vetor3: Posição da Camara;*
- *Matrix4x4: Matriz de Visão da Camara;*
- *Matrix4x4: Matriz de Projeção da Camara;*

Essa informação é então processada e preparada para ser enviada para o primeiro Fragment Sahder:

OUT:

- *Vetor3: Posição do Fragmento no Mundo/Espaço;*
- *Vetor2: Coordenadas de Textura do Fragmento;*
- *Matrix3x3: Matriz TBN;*

A posição do fragmento no ecrã é calculada pela multiplicação de matrizes4x4 por um vetor3 XYZ pela seguinte ordem:

*Matriz Projeção * Matriz Visão * Matriz Modelo * Posição do vértice;*

A posição do fragmento no "Mundo/Espaço" é calculada pela seguinte multiplicação:

*Matriz Modelo * Posição do vértice;*

É também necessário calcular uma matriz que transforme pontos relativos ao mundo para pontos relativos à superfície do modelo 3D, de modo a que as normais da superfície estejam sempre bem alinhadas independentemente da posição, rotação, ou escalamento do objeto. Para isso é necessário calcular o chamado TBN Matrix.

Esta Matriz é calculada da seguinte forma:

1. Calcula-se a Matriz Normal:
    - Ignorando a quarta dimensão da Matriz Modelo, calcula-se o inverso desta nova Matriz3x3 e efetua-se a transposição da mesma de seguida.
2. Normaliza-se a operação de multiplicação entre a Matriz Normal e o vetor tangente do vértice, para obter o VetorXYZ "T" (Tangente).
3. Normaliza-se a operação de multiplicação entre a Matriz Normal e o vetor Normal do Vértice, para obter o VetorXYZ "N" (Normal).
4. Iguala-se o VetorXYZ "T" à normalização da operação:
    - *T - ProdutoEscalar (entre T e N) * N;*
5. Realiza-se o Produto vetorial entre T e N, para obter o VectorXYZ "B"(Bi-Tangente).
6. Calcula-se a Matriz TBN Final:
    - Utiliza-se os vetores "T" "B" e "N" para construir uma Matriz3x3 e calcula-se de seguida a transposição desta nova Matriz3x3.

A Coordenada de Textura é simplesmente enviada para o Fragment Shader sem sofrer alterações.

Com isto terminam-se as operações relativas ao Primeiro Vertex Shader.

Visto que estamos a utilizar uma Pipeline Gráfica de Deferred Rendering, o Primeiro Fragment Shader recebe estes valores e utiliza 8 Render Targets (Texturas), para guardar os cálculos relativos à criação do Material PBR. Estes valores guardados em Texturas são depois Enviados para o segundo Fragment Shader.

As 8 Texturas geradas a partir do Primeiro Fragment Sahder são as seguintes:

OUT:

- *vetor4 Tangent Space Fragment Position Color;*
- *vetor4 Tangent Space Eye Position Color e Material Color Alpha;*
- *vetor4 Tangent component of TBNMatrix e Material Roughness;*
- *vetor4 Bi-Tangent component of TBNMatrix e Material Metalness;*
- *vetor4 Normal component of TBNMatrix e Material Ambient Occlusion;*
- *vetor4 Normal Texture Color e Albedo Red Color;*
- *vetor4 Diffuse Color e Albedo Green Color;*
- *vetor4 Specular Color e Albedo Blue Color;*

Olhando para estas texturas verificamos que há partilha de elementos na mesma textura (ex.: Tangent component of TBNMatrix e Material Roughness). Isto foi decidido de modo a aproveitar o máximo espaço das Texturas (Vetor 4 XYZW) no menor número de RenderTargets (neste caso acabou por ser 8).

No Segundo Vertex Shader são simplesmente passadas as coordenadas do viewPort da câmara (4 vértices) e geradas Coordenadas de Textura para os mesmos, de modo a desenhar a imagem final num Retângulo.

Esta imagem é calculada no Segundo Fragment Shader, utilizando as 8 texturas criadas no primeiro, contendo os valores necessários para a composição da imagem com materiais PBR e para os cálculos de Luz finais. Depois disto a imagem está terminada e pronta para continuar o seu percurso no ciclo de renderização.

4

Tech-Demo Notes

Key-Bindings:

-F1- CockpitCamera

-Number 0: FreeCamera

-Number 9-7: change skybox

-Number 6: disable skybox

-Rshift: exposure up

-RControl: exposure down

-Rshift + Dash: gamma up

-RControl + Dash: gamma down

-when on CockpitCamera

-F3- Top camera

-F4- BottomCamera

-WS - Forward / Backward

-AD - Turn Left / Turn Right

-QE - Strife Left / Strife Right

-LShift LControll - Up / Down

-Spacebar : Breaks

-when on Top or bottom camera:

-F5 - Normal Vision

-F6 - NightVision

-when on Night Vision:

-LeftRight Mouse Button : Zoom IN/OUT

-when on FreeCamera

-Number 1: no light

-Number 2: white pointLight

-Number 3: multicolor pointLight

-Number 4: White spotLight

-WASDQE-Movement

-LControl- SpeedUp

Notas relativas ao Tech-Demo

Com este tech-demo tenciono demonstrar a maior parte dos elementos implementados no motor de jogo, consistindo numa cena de jogo construída a partir de modelos 3D carregados pelo motor de jogo, um objeto controlável pelo jogador, por meio de scripting, e objetos de cenário independentes controlados por meio de scripting, a capacidade de movimentação com uma camara livre, vários tipos de iluminação, materiais PBR, skyboxes, elementos de UI interativos, pós-processamento de imagem e múltiplas ViewPorts.

5

List of Implemented Features

Legenda:

- [x] : Implemented
- [/] : Started but not finished
- [ ] : Not Started.

- [x] Custom Multi-Resolution.

- [x]Window/Borderless/Fullscreen Modes

- [x] Down sampling Rendering.

- [x] Linear or Nearest Image Filtering

- [x] Deferred Rendering.

- [x] Deferred Lighting.

- [x] Point, Directional and spotlight Light types.

- [x] Custom PBR Materials.

- [x] Multiple Viewport Rendering.

- [x] Post-processing stack for each camera.

- [x] Custom Post-processing Effects.

- [x] Skybox Generation from cubemap (6 images) or from a single Image.

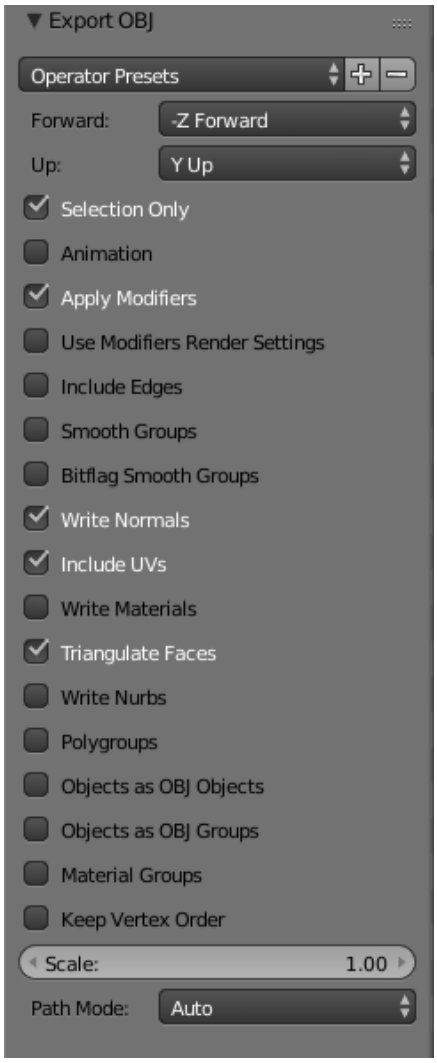
- [x] HDRI Images for Skybox.
- [x] Precomputed Environmental Lighting from skybox.
- [x] Reflexion of skybox in polished materials.
- [x] Seamless transition between Multiple scenes.
- [/] Object Instantiation and destruction.
- [x] Component based Architecture.
- [x] Scriptable Objects.
- [x] Texture Loader.
- [x] OBJ and COLLADA Model Loader.
- [x] UI Canvas.
- [x] UI Images.
- [/] UI Text.
- [ ] UI Buttons.
- [/] Custom Shaders.
- [/] Nvidia PhysX Implementation.
- [ ] Shadow Casting.
- [/] Model Animator.
- [/] Particle Systems with Scriptable Particles.
- [/] 3D Audio
- (and maybe something more that I may have forgotten)

Currently Supported 3D Model Files:

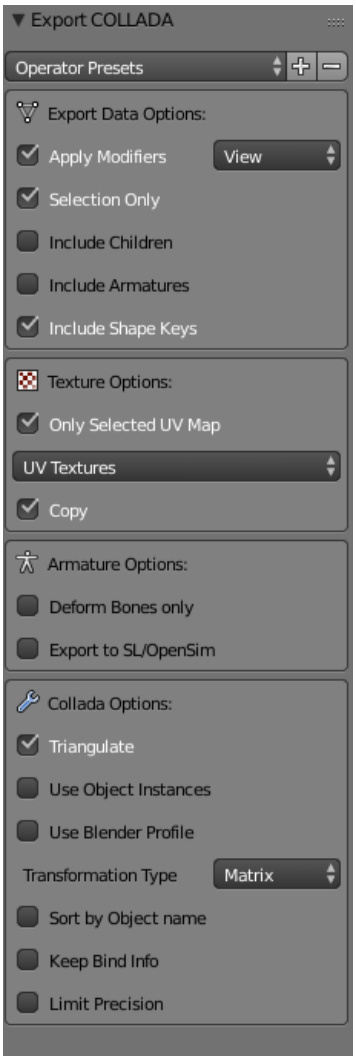
- ".obj" (OBJ files), for static meshes.
- ".dae" (COLLADA files) for static and animated meshes.

3D Model Export specifications:

.OBJ:



.DAE:



Conclusão

Olhando para todo este projeto, verifico que sinto um grande prazer em ter trabalhado no mesmo. Considero que foi um excelente ponto final no meu percurso académico pois senti que me levou à aplicação de múltiplas áreas que estudei ao longo desta licenciatura, levou-me ao limite das minhas capacidades e evoluí para além delas.

Comparando os meus conhecimentos como Programador no início do projeto para agora, sinto que os meus conhecimentos e aplicação prática evoluíram muito positivamente. Aprofundei áreas de computação que me interessavam e ganhei competências de perseverança e rotina que definitivamente me irão ser uteis no meu futuro de trabalho.

Embora não tenha terminado por completo tudo o que tinha planeado no início do projeto, parece-me que estive muito perto de o fazer. De certa forma, muitas das coisas que tinha planeado inicialmente não me agradavam a meio do projeto, visto que os meus conhecimentos acerca das mesmas ganharam profundidade, sendo assim houve algumas modificações e melhoramentos em que decidi usar o meu tempo.

Se não fossem algumas distrações e imprevistos durante os meses de projeto acredito que o teria concluído a 100%, mas mesmo assim, não estando a 100% acredito que será o suficiente para demonstrar o meu empenho e dedicação por este projeto, e paixão por esta arte que é Programar.

Glossário

[1] Classe Abstracta Pura – Classe pensada para ser obrigatoriamente usada como classe Mãe, contendo pelo menos uma função Virtual Pura.

[2] Função Virtual Pura – Função membro declarada numa classe Mãe que necessita de ser obrigatoriamente redefinida por subclasses de objetos que herdem da função Mãe.

[4] Framebuffer – Porção de RAM capaz de armazenar um mapa de bits que define a imagem de exibição de um display de vídeo.

[5] Pipeline – elementos de processamento de informação conectados em série.

[6] Anti-aliasing – técnica de computação gráfica com o objetivo de reduzir bordas irregulares em polígonos no ecrã.

[7] Multisampling – Para cada pixel na extremidade de um polígono, são tirados múltiplos samples de modo a corrigir bordas irregulares. Também conhecido por multisampling alntialiasing (MSAA).

[8] Rasterização – processo de conversão de uma imagem vetorial para pixels, de modo a possibilitar a sua apresentação num ecrã.

[9] Vertex Shader – Fase da pipeline gráfica em que um shader programado processa informação relativa a vértices individuais.

[11] Fragmento – produto do processo de Rasterização de Geometria num Vertex Shader.

[10] Fragment Shader – Fase da pipeline gráfica em que um shader programado processa informação relativa a fragmentos provenientes de rasterização.

Bibliografia

[1] http://blog.selfshadow.com/publications/
s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf

[2] https://www.khronos.org/registry/
OpenGL/specs/gl/glspec45.core.pdf

Webgrafia

[1] https://github.com/mattateusb7/NamelessEngine-Framework

[2] https://mattateusb7.github.io/NamelessEngine-Framework/documentation/

[3] https://learnopengl.com/

[4] http://artisaverb.info/PBT.html

[5] https://thebookofshaders.com/

[6] https://choosealicense.com/licenses/

[7] https://www.sfml-dev.org/

[8] https://glm.g-truc.net/

[9] http://glew.sourceforge.net/

[10] https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342