THESCENT

A final project for the Game Development: Advanced Programming Post-Graduate at Fanshawe College

Felipe Da S. Bellini

B. Eng. Control Systems

Keywords: AI, C++, Descent, Entity Component System, Game development, Graphics, OpenGL, Physics, Simulation, Software

"Bernard of Chartres used to compare us to dwarfs perched on the shoulders of giants. He pointed out that we see more and farther than our predecessors, not because we have keener vision or greater height, but because we are lifted up and borne aloft on their gigantic stature." -- John of Salisbury, Metalogicon

To Erineu and Gilda Bellini, my parents

Revision History

Date	***Version***	***Changes***
Apr. 13 - 2020	1.0.0.0	*Initial documentation.
Apr. 16 - 2020	1.0.0.1	*Added sections for Sound, Game Engine and Patterns, Physics
Apr. 20 - 2020	1.0.0.2	*Added missing sections, overview of gameplay, screenshots
Apr. 23 - 2020	1.0.0.3	*Updated text after addressing the timestep issues described in the Developer's Notes section. *Updated controls section.
Apr. 24 – 2020	1.0.0.4	*Fixed typos in the document

Contents

1. Introduction

The “Game Development: Advanced Programming” Post-Graduate, also known as GDP, at Fanshawe College, is a specialized software development program geared towards building games and engines from scratch. During its course, students are tasked with several hands-on projects in the most critical areas of the game industry, such as 3D graphics and sound, artificial intelligence, development operations, networking, physics and software development patterns.

At the end of the program, the final project, entitled “Game Jam”, is the culmination of all these subjects that were lectured. This document outlines the development process and features of Thescent, an OpenGL approach to the original Descent™ (1995) by Interplay Entertainment.

Descent was the first fully 3D game released. It was a first-person shooter, where the player was driven into a sci-fi maze-like environment and had to defeat enemy drones, rescue hostages and destroy reactors.

Thescent features a single level of gameplay that tries to match the original game while also being designed to show several systems working together to simulate a proper game.

2. Tools and Programming Languages

During the development of this project, the following tools were used:

- Microsoft Visual Studio 2019 Community (link)
- Microsoft Office 2019 Professional (link)
- Inkscape (link)
- GIMP (link)
- MeshLab (link)
- Blender (link)
- Microsoft 3D Builder (link)
- Notepad++ (link)

Programming languages:

- C/C++ 98/11/17 - Internals of the game engine (link)
- C# - Custom tools to aid during development (link)
- Lua 5.3.5 – Scripted Animations (link)
- WiX 3.11 – Installer (link)

3. Scope

The scope of this project is to:

- Provide four to six technical things related to the program's subjects
- Provide a customized engine written in C++
- Provide a portfolio-oriented sample for evaluation of the knowledge acquired during the GDP

For more information of the scope, visit the Scope document, provided by the teaching staff at Fanshawe College.

4. Technical Aspects

This section discusses the technical features applied in this project. These are separated by the courses that they mostly relate to.

Some of the features demonstrated in this project are:

- Instanced rendering
- Multiple shader program support
- An ECS Engine
- Ray casting
- Ghost objects
- Multithreading
- A package installer
- Texture effects such as tiling, offsetting over time, blending
- VFX such as transparencies with discard and alpha, particles, imposters
- Deferred rendering
- GUI support
- Lua and shader animations
- In-game events
- Sounds
- A full game

4.1. Artificial Intelligence

Thescent makes use of a threaded AI system that is located at BellEngine.AI.Custom. This system is detached to a DLL and can be swapped by another AI library that inherits from the same *ISystemAI* interface.

The *SystemAI* does not hold any sort of data, but will update entities provided a Scene is passed into the system before an update call is made. The update call will loop through all entities in the scene and check if any of those have AI behaviors. Those that do, will be checked if they are active or not and if yes, the update call of the behaviour is called.

AI behaviors can have a broad or narrow scope, where:

- Broad scope behaviors are those that are applied to an entire scene or a group of entities
- Narrow scope behaviors are those applied to a single entity

AI behaviours are used in multiple ways during the game, for NPCs to interact with the player, visual effects and even imposter illusions. The following points describe which behaviors are in play during the game.

4.1.1. Steering Behaviors

The enemy drones have steering behaviors to chase the player given a certain range and evade player shots, increasing the difficulty of the game.

Examples can be located at:

- $(SolutionDir)BellEngine.AI.Custom\AIBehaviors\src\Pursue.cpp
    - Given a target entity, causes the parent entity to follow the target
- $(SolutionDir)BellEngine.AI.Custom\AIBehaviors\src\Evade.cpp
    - Will escape target entities that are in range.
    - In Thescent, the custom evade will always try to evade the closest projectile shot by the player.

Img. Drone chasing player

4.1.2. Attack and Observe Behaviors

These custom behaviors are used to look at and cause damage to the player during the game. Given a variety of data such as the player position in 3D space, a radius of action and even the scene itself, these behaviors are capable of finding the player in the scene and interact with it in some way.

Examples can be located at:

- $(SolutionDir)BellEngine.AI.Custom\AIBehaviors\src\LookAt.cpp
    - Given a target position or an entity, causes the parent entity to 'observe' continuously
- $(SolutionDir)BellEngine.AI.Custom\AIBehaviors\src\LookAtMovingDirection.cpp
    - Forces the parent to always be oriented towards its current normalized velocity (or direction of movement)
- $(SolutionDir)BellEngine.AI.Custom\AIBehaviors\src\ShootAt.cpp
    - Given a target entity, a scene, a projectile and a recharging rate; it'll clone and shoot projectiles towards the player if certain conditions are met

4.1.3. Visual FX Behaviors

One interesting aspect of AI and control systems is that it can accommodate a limitless range of possibilities; meaning that any process can be turned automated in some way.

In Thescent, there are two particular cases that fit well this line of thought:

- The “Anticloak” item found in the first room of the game is an imposter plane, but what drives its world transform is the *LookAt* AI behaviour. This causes the item to always face the player no matter what – instead of hardcoding an imposter effect for that item.
- The skybox distortion is also driven by a *LookAt* behavior. One of the original ideas behind this was that the Space Station was to have a force field surrounding it and although there are many ways to approach this, a hidden object was placed at the centre of the skybox. This object has a *PathFollowing* AI that goes around circles. The skybox then, observes this object continuously which causes distortion visual effect to run.

Img. “Anticloak” item is an imposter with an observe player AI

4.2. Algorithms and Gems

A complex use-case of multithreading was applied to the game. There are, at the time of this paper was written, 6 threads in the game, these being:

- Main Thread (main.cpp’s while loop or *GameWorld’s Update(…)*)
  The main loop thread or program thread. This thread is responsible for promoting updates to the *VideoSystem*, *SceneSystem*, *UserControlSystem* and *GUISystems*. Most of these systems are linked to OpenGL, which can only be used in the main thread context.

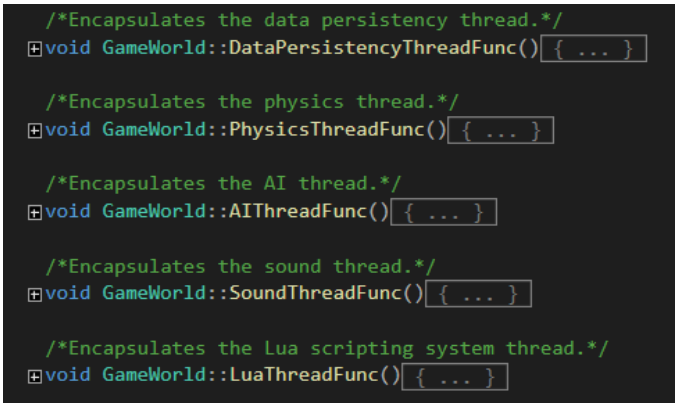
- *AIThreadFunc()*
  The AI thread is responsible for updating the AI components on every entity in the game. Thread protection was involved here because the AI system often changes physics properties of the entities, as, for instance, setting the rotation or velocity of a rigid body. This is a C++ 11 thread, created and detached by Thescent's *GameWorld*.

- *DataPersistencyThreadFunc()*
  This thread will listen for "commands" to load data. Most importantly, it loads the level in the background, including all models, textures and entities that are stored in files. This allows Thescent to not hang during initialisation and show a clean Loading Screen when it opens. This is a C++ 11 thread, created and detached by Thescent's *GameWorld*.

- *LuaThreadFunc()*
  This thread is responsible for updating all the scripted sequences in the game. Since scripts can often changes properties that are contained by other systems, such as physics, it also offers some level of cross-thread protection. This is a C++ 11 thread, created and detached by Thescent's *GameWorld*.

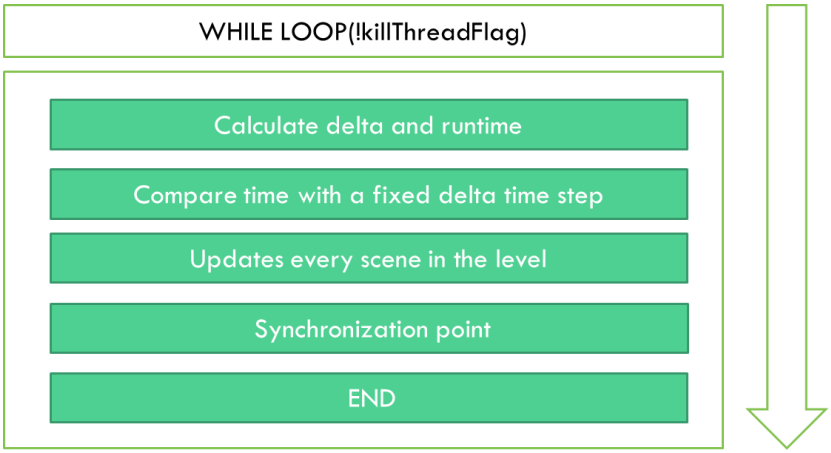
- *PhysicsThreadFunc()*
  This is the most complicated of all threads. The physics' related objects are constantly queried and updated by multiple threads and therefore need to be tightly protected by locks in the framework. The physics thread is responsible for integrating and propagating collisions between objects in the game. This is a C++ 11 thread, created and detached by Thescent's *GameWorld*.

- *SoundThreadFunc()*
  The sound thread is responsible for playing sounds in the game. It has the least amount of impact of the detached threads and runs concurrently to all others. This is a C++ 11 thread, created and detached by Thescent's *GameWorld*.



Img. Threading methods

A typical thread in the game works as in the frame below.



Img. Thread layout

The detached threads in the game are started once the DataPersistencySystem finishes loading the Level from the xml file – in the method *OnFileLoadCallback(…)*.

The systems that are running in these threads have a single synchronisation point. *WaitForEntitiesToBeDisposed*(…) is an important method that will hold the thread until all the disposing objects have been successfully removed and deleted from the scene. For Thescent this means projectiles, dead enemies, etc.

In addition to that, every thread has its own exception handling for any exceptional errors that might happen during runtime.

Cross thread protection occurs primarily with the use of critical sections and/or atomics, as in the pictures below.

```
#ifdef __linux__
 #include <mutex>
#elif _WIN32
 #define NOMINMAX
 #include <windows.h>
#else

 #endif

 #define RIGIDBODY_STR "RigidBody"

class RigidBody : public IRigidbody
 {
 protected:
#ifdef __linux__
    std::mutex mutex;
#elif _WIN32
    CRITICAL_SECTION lock;
#else

#endif
```

Img. Use of critical sections

```
#include <atomic>

class SoundComponent : public ISoundComponent
{
private:
    std::string path;
    std::atomic<bool> isActive;
    std::string name;
    std::atomic<int> sourceType;
```

Img. Use of atomics

4.3. Animation

The animations are divided into two categories here, shader-based animations and Lua scripted-sequences.

4.3.1. Shader-Based Animations

Shader-based animations are animations that require meshes to have bones, these are transformed over time and their matrices are sent to the shader which effectively modifies the vertices in 3D space, causing a ‘static’ mesh to be able to move during runtime.

In Thescent, these types of animations are present in the hostages. These are entities that have a *SkinnedMeshComponent* attached to them and the appropriate *ShaderFeature* for sending data to the vertex shader. The *SkinnedMeshComponent* was derived from the works of Professor Michael Feeney at Fanshawe College and adapted to the ECS framework of BellEngine.

The model used, “Peanut Man”, has an internal state machine that changes according to the distance to the player. The hostages in the game are contained within a field of energy, or cage, that is a transparent, but visible, game entity.

Once the hostage’s state changes, the shader animations are kicked and will loop until the next animation is played. Animations will run until completion, and thus do not overlap on top of one another.

The reason for these animations was to have the hostages struggling to get out of the cage and/or try to ask the player for help. Once the hostages are rescued, the animations are halted.

Img. Hostage inside the cage

4.3.2. Scripted-Sequence Animations (LUA)

Another aspect of Thescent is the addition of scripted sequences. An example of that are the hatches in the game. Once the player executes an action, the doors will allow the ship to visit new areas of Thescent.

This was accomplished with the use of Lua; a flexible programming language maintained by PUC-RIO in Brazil. Lua is widely used by the game industry to integrate scripted sequences and cutscenes into products.

Lua scripts can be either focused on a single entity or can be applied to multiple entities in the game. In BellEngine, developers can either attach scripted animation components to entities and/or have the animations in the scene itself. The later was used for the Thescent. All instances of the OpenDor.lua script, for instance, are driven by completely different actions in the game; however, their code is the same and more importantly, unaware of the game logic.

Img. Door that can be opened with a lua script

Scripted sequences are handled by the BellEngine.Scripting.Lua, which contains a *ScriptingSystem* that runs on it's own thread. The system continuously processes scripts until they detach themselves from the *LuaWorld*.

The connection between the game and Lua happens through the declaration and registering of *static int(luaState*)* functions. These can be used by any scripted sequence in the game. Some of the functions available are listed in the table below.

Function	***Description***
SYS_GetDeltaTime	Returns the frame's deltatime to the script. Useful for interpolating over time. Which is the approach used by the doors.
SYS_Dispose	Causes the script to be removed from the list of scripts being currently processed by the scripting system.
ENT_SetPosition	Sets the position of an entity in 3D space. This shows a clean example of how Lua can interact with the physics in the game.

…	Other functions can be found at BellEngine.Scripting.Lua

4.4. Component Integration

BellEngine integrates third party technologies to accomplish some common tasks such as reading XML files and using functions from OpenGL. The table below describes which libraries are included in the engine.

Since Thescent is a technical demo, free and/or open-source solutions were preferred during the development of the project.

Third party technology	***Desciption***
Assimp	Is used to load meshes into BellEngine's own mesh container. This includes vertices, triangles and bone animations. Can be found at BellEngine.Mesh.Assimp project.
Bullet	Physics library mainly used for robotics. Although BellEngine offers support for that, Thescent does **not** use Bullet for its physics. Can be found at BellEngine.Physics.Bullet.
Dear ImGUI	C GUI framework built with OpenGL. Provides access to menus and buttons for enhanced presentation. Can be found at Game.
GLAD	Provides access to low level OpenGL functions. Used for things that are graphics-related, like operations on the shaders and the video card. Can be found at Game, BellEngine.Core, BellEngine.Common.
GLFW	Provides an OpenGL framework for the game window, input/output callbacks, context control and much more. Can be found at Game, BellEngine.Core, BellEngine.Common.
Lua C binder	Official C style library provided by the developers of Lua. Provides functions to link C/C++ programs with the Lua framework. Lua is used for scripted animations in Thescent. Can be found at BellEngine.Scripting.Lua.
OpenAL	C style audio API. Provides native support for playing, recording and configuring sounds in the game. Can be found at BellEngine.Sound.OpenAL.
RapidXML	C++ library for reading XML files. The levels, scenes and entities are stored as XML data in Thescent. Can be found at Game.Persistency.XML.
SOIL	C library is used for loading textures. Can be found at BellEngine.Core.

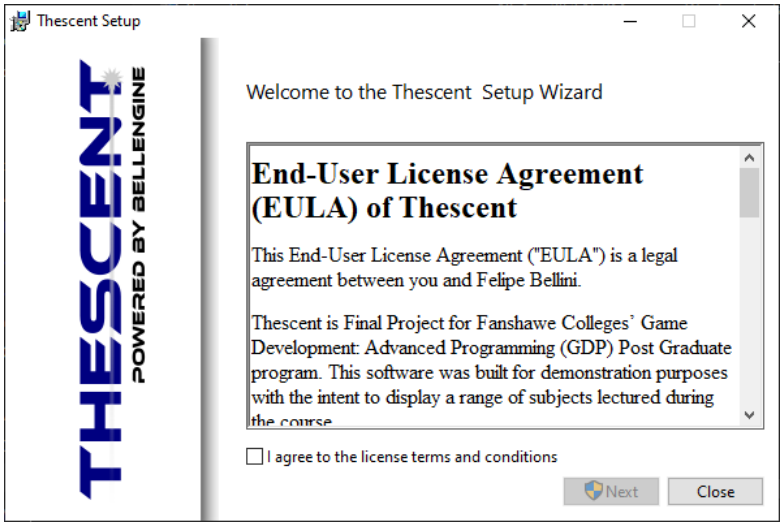
Links for these technologies can be found in the supporting libraries section, all copyright is attributed to the teams behind their respective projects.

4.5. Configuration and Deployment

4.5.1. A Professional Game Installer

Installers are a clean and effective way to distribute deliverables to consumers, while also helping products stay organized.

This project features an installer built with WiX Toolset. WiX is a free professional framework for developing complex installers for Windows. Its programming language exists in the form of XML nodes and has full support for Visual Studio.



Img. Setup intro

In order to build both installer projects of Thescent, users will need:

- WiX Toolset v.3.11 SDK (from the official website)
- WiX Toolset Visual Studio 2019 Extension (from the Visual Studio Marketplace at Visual Studio->Extensions->Manage Extensions)

```
<CustomAction Id="Cleanup"
              Directory="CompanyFolder"
              ExeCommand="cmd /C &quot;rmdir /s /q &quot;.\Thescent&quot;&quot;"
              Execute="deferred"
              Return="ignore"
              HideTarget="no"
              Impersonate="no" />
<InstallExecuteSequence>
  <Custom Action="Cleanup" After="RemoveFiles" >
    REMOVE="ALL"
  </Custom>
</InstallExecuteSequence>
```

Img. - Excerpt from the code showing a custom action

Thescent's installer is provided in two steps, an MSI package that delivers the game and the engine (*Game.Setup*) and an EXE bundle (*Game.Setup.Full*) that delivers the aforementioned MSI plus the runtimes for Microsoft Visual C++ 2019 (VC++140-142) and OpenAL (1.1).

- *Game.Setup* shows examples of custom actions, shortcut creation and both dynamic and static binary packaging.
- *Game.Setup.Full* shows how to create professional packages for the end consumer.

Users only need to have the EXE installer, as it is the complete deliverable.

4.6. Game Engine and Patterns

BellEngine is a C++ engine that follows the principles of the Entity Component System and the Object-Oriented Programming. The engine is divided into several subsystems, among them promoting:

- Assembly detachment through the use of DLLs and common interfaces to provide clean and intuitive ways to expand or replace systems
- Multithreading, as often times games split their processing to improve performance
- Events/Callbacks to link systems together without referencing one another
- A common *ISystem* interface so all systems follow the same OOP structure

This engine borrows some ideas of popular game engines as Unity and Godot, although in no way aims to serve as a replacement for any well-established engine.

The purpose of BellEngine is to serve as a study case and platform for game development during the course of the GDP.

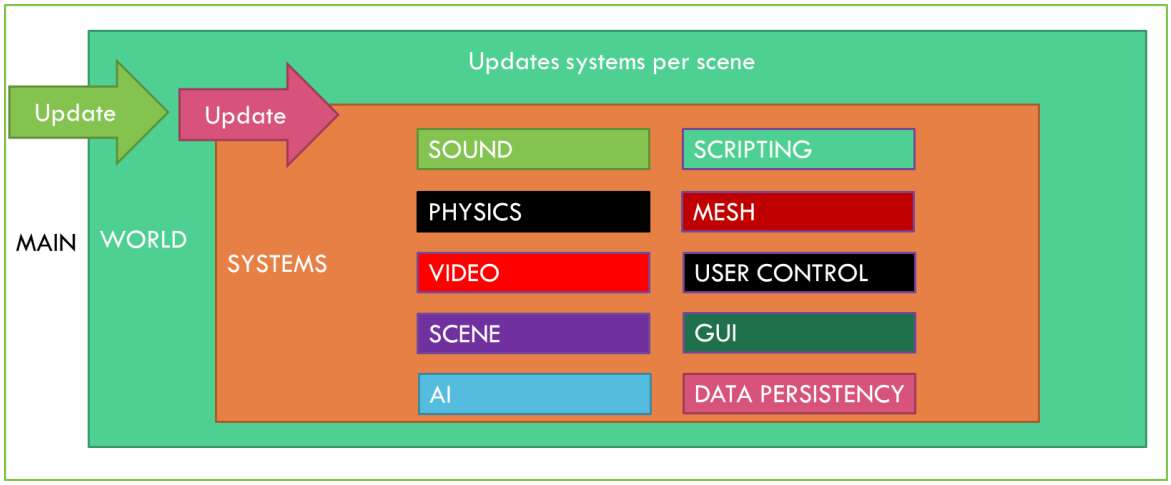
The systems provided with the current version of BellEngine are:

System name or alias	***Goal***
Core/Game.SystemGUI	Graphics User Interface features
Core/Game.SystemVideo	Drawing, shader handling and OpenGL
Game.DataPersistency.XML	Reading XML files
Game.SystemScene	Handles scenes' specific logic
Game.SystemUserControl	Handles user input for mouse and keyboard
Game.VertexAnimationSystem	Handles shader animations
BellEngine.AI.Custom	Handles AI behaviors
BellEngine.Mesh.Assimp	Reading 3D model files
BellEngine.Mesh.PlyReader	Reading 3D model files in ply format
BellEngine.Physics.Custom	Self implemented physics support
BellEngine.Physics.Bullet	Bullet based physics support

BellEngine.Scripting.Lua	Scripting system that handles Lua scripts
BellEngine.Sound.OpenAL	Sound system based on OpenAL

The *GameWorld* creates all it's systems and registers itself in all necessary the callbacks so it can then link system behaviors during the game (IE: Linking a sound to an AI event).

When the game loop starts, it calls the *Update(…)* method of the *GameWorld*, passing the current time which calls the update of each system. The *GameWorld* also fires separate and detached threads for the systems that are multithreaded as these contain their own update cycles.



Img. Gameloop architecture

Each system has two main methods that are called during the loop, *SetScene(…)* and *Update(…)*.

- Given that a level contains a collection of scenes, *SetScene(…)* takes each scene separately as the target of the update
- *Update(…)* takes the game runtime (or thread time) and the frame`s delta time and applies that to the entities inside the selected scene

A system will only update one level per frame. If a second level is loaded, then it must become the active level. By default, only one level is loaded and kept in memory for performance.

4.7. Graphics

4.7.1. Instanced Rendering

Instance rendering is an OpenGL feature that allows applications to draw sequences of objects with a single draw call. This is accomplished with the use of *glDrawElementsInstanced* or *glDrawArraysInstanced* and some modifications to the shader(s).

This project uses instance rendering to render the particles (the core) that are seen inside the reactor. The approach used was divided as follows:

- An instance rendering shader was created with an array of *mat4*
  - $(SolutionDir)Game\assets\shaders\VertexInstanceRendering.glsl
- On the CPU side, the model matrices for all particles are pre-calculated
  - $(SolutionDir)Game\src\GamesystemVideo.cpp - *DrawParticles(...)*
- Matrices are passed to the shader using the *InstancedVertexShader* class
  - $(SolutionDir)Game\Shaders\src\InstancedVertexShader.cpp
- The shader uses *gl_InstanceID* to access the matrices and draw the objects onscreen

Img. Reactor with textured imposters

4.7.2. Deferred Rendering

Deferred rendering is a technique where the application renders objects offscreen in a buffer other than the main buffer (or buffer 0). These can be stencil buffers, framebuffers, etc.

Thescent makes use of this technique to render the main game. The main scene (Scene01) is rendered as a texture on a quad and displayed in front of the cockpit and crosshair of the ship.

Since scenes are separate in this architecture, every scene has its own camera and therefore will use the view and projection matrices of their respective cameras, unless specified otherwise.

This section also includes a shader pass to simulate the “Anticloak” vision. This is a shader pass that is applied to the main scene texture and simulates the effect of looking through a green glass.

Img. Anticloak device used to see the boss

Shader passes can be blended to create a variety of different results.

```
/*Processes all the fullscreen texture passes and effectively blends them*/
/*Users can add many passes for the same texture to combine effects*/
vec4 resultBuffer = vec4(0.0, 0.0, 0.0, 1.0);
vec4 result = vec4(0.0, 0.0, 0.0, 1.0);
for(int i = 0; i < passProcessInfo.passArrayCount; i++)
{
    result = vec4(0.0, 0.0, 0.0, 1.0);
    switch(passProcessInfo.passArray[i])
    {
        case 0: /*Reserved*/ break;
        case 1: Pass_FBOTextureEmplacement(result); break;
        case 2: Pass_InClassBlur(result); break;
        case 3: Pass_Blur(result); break;
        case 4: Pass_NightVision(result); break;
        case 5: Pass_TVStatic(result); break;
        case 6: Pass_DepthTest(result); break;
        case 7: Pass_SinWave_Wobbly(result); break;
        case 8: Pass_SinWave_Static(result); break;
        default: break;
    }

    resultBuffer += result;
}
```

Img. Blending passes in the fragment shader

4.7.3. Multiple Shader Programs

The engine provides support for using multiple shader programs. As an example, Thescent uses two programs by default, the first one runs the uber shaders, while the second is used by the reactor core (InstanceRendering+Imposter shaders).

Shaders are loaded from the shader_programs.xml file. This generates a *ShaderContainer* that wraps the shader programs.

```
<?xml version="1.0" encoding="utf-8"?>
<ShaderContainer Type="GLShaderContainer">
    <!-- Available programs, which are really shader combinations -->
    <ShaderPrograms>
        <Program>
            <Parameters>
                <Name Type="string">ShaderProgram01</Name>
                <ShaderCount Type="unsigned int">2</ShaderCount>
                <Shader0Path Type="string">VertexShader.glsl</Shader0Path>
                <Shader0Type Type="unsigned long long">1</Shader0Type>
                <Shader1Path Type="string">FragmentShader.glsl</Shader1Path>
                <Shader1Type Type="unsigned long long">2</Shader1Type>
                <AvailableFeatures Type="string">BASIC_VERTEX_DRAW SKINNEDMESH COLORIZE INTERE
                <AvailableFBOPasses Type="string">ALL</AvailableFBOPasses>
            </Parameters>
        </Program>
        <Program>
            <Parameters>
                <Name Type="string">ImposterInstanced</Name>
                <ShaderCount Type="unsigned int">2</ShaderCount>
                <Shader0Path Type="string">VertexInstanceRendering.glsl</Shader0Path>
                <Shader0Type Type="unsigned long long">1</Shader0Type>
                <Shader1Path Type="string">FragmentShaderImposter.glsl</Shader1Path>
                <Shader1Type Type="unsigned long long">2</Shader1Type>
                <AvailableFeatures Type="string">INSTANCED_DRAW IMPOSTER</AvailableFeatures>
                <AvailableFBOPasses Type="string">ALL</AvailableFBOPasses>
            </Parameters>
        </Program>
    </ShaderPrograms>
</ShaderContainer>
```

Img. XML showing both programs

When the game is running the *GameSystemVideo's WriteShaderFeatures(…)* will look for the program the contains all the *ShaderFeatures* attached to the game entities and will set the program id accordingly.

A *ShaderFeature* is a generic structure to keep shader data during runtime (example: the current offset of a texture, if the object is covered by a texture or just a color).

```
IShaderProgram* program =  this->shaderContainer->FindShaderProgramByAllFeatures(shaderFeatures);
if (!program) continue;

ResolveCamera(program->GetID());

IGLShader* sd = glShaders[SHADER_INDEX_VERTEX_INSTANCED];
if (sd)
{
    ModelDrawInfo modelDrawInfo;
    if (FindModelDrawInfoByMeshName(mesh->GetName(), modelDrawInfo))
    {
        char buf[100];
        sprintf_s(buf, 100, "vertexShaderInfo.shaderFeatureArray[%llu]", vertexShaderCount);
        GLint shaderFeatureArray = glGetUniformLocation(program->GetID(), buf);
        glUniform1i(shaderFeatureArray, shaderID);
        //-------------------------------------------------------------------------------------

        InstancedVertexShader* shader = (InstancedVertexShader*)sd;
        shader->SetProgramID(program->GetID());
        shader->SetDrawInfo(modelDrawInfo);
```

Img. Swapping shader programs during runtime

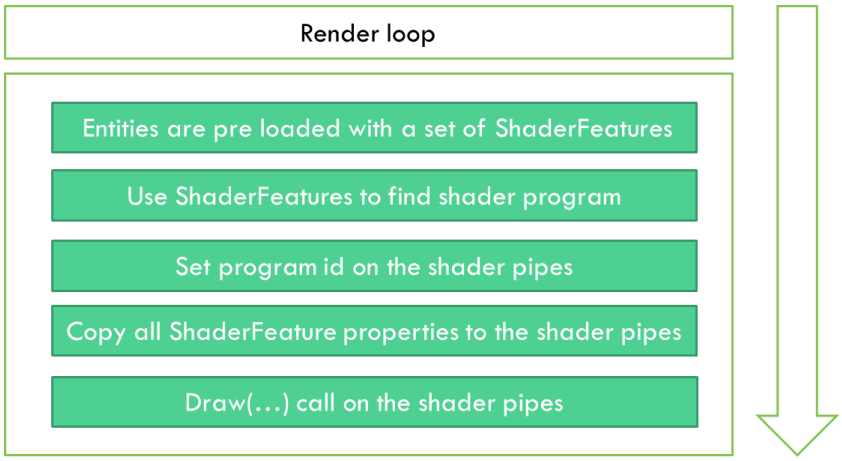
4.7.4. Uber Shaders and C++ Shader Classes

An uber shader is a GLSL shader that contains logic for all effects needed by the game. The uber shaders currently drives most of the graphics of Thescent.

These shaders make use of layout locations for the VAO/VBO buffers, structs, uniforms and other OpenGL features. The uber shader also is capable of blending effects, such as chained passes on a fullscreen texture.

On the CPU side, the shaders are separated in four different structures that combined, give the final result:

- ShaderFeatures
  - Objects that hold the current shader data of a given entity
- Shader
  - A Generic shader class to store the contents of the GLSL file and the id after compilation
- ShaderProgram
  - Stores the full program containing vertex+fragment+geometry and the proper associated Shader objects
- Individual shader classes
  - The CPU shaders, or pipes, are C++ classes used to store the current locations of the uniforms and effectively wrap the draw calls to the OpenGL API. They create an interface between the video system and the GLSL shaders. This way we can reduce the amount of calls to the pipeline every frame.



Img. Render pipeline of the shaders

4.7.5. Transparencies

Transparencies correspond to a partial or complete omission of a color or texture, producing a see-through or glass effect.

There are multiple use cases for this concept in Thescent.

- The windows, from where we can see the skybox and parts of the space station in the entrance hall and both left and right rooms
  - These use discard in the shader to not draw the black portions of the textures
- The reactor is a “glass” looking capsule, where the player can see the particles moving inside
  - This uses the alpha off the diffuse colour
- The player itself is a transparent sphere

    - This uses the alpha off the diffuse colour
- The hostage cage is a translucent box
    - This uses the alpha off the diffuse colour

In order for these transparencies look realistic to the player, the video system has to order all the entities in the scene according to their position and distance to the camera, so the see-through effect can appear natural. The transparent objects are then always rendered last.

This logic is used by the GamesystemVideo and resides on the Core's SystemVideo.

Img. Example of discard transparency

4.7.6. Particle Components

Particles are tiny objects used to simulate visual effects such as flames, sparks, water drops, etc. This project has as an *IParticleEmitter*, that is an *IComponent* that can be attached to any entity in the game.

In Thescent, this is used by the reactor core, where we have one entity "REACTOR" with the IParticleEmitter attached. The particle emitter then bursts a sequence of quads, that are imposters, with a red lightning texture on top of it.

4.7.7. A GUI Overlay

A graphical user interface is a common asset is software development, and allows projects look altogether more professional. For games it may come in the forms of game menus, HUDs and other ideas.

This feature is carried over by the *GameSystemGUI* and is internally processed by the ImGui C API.

Another way to approach this, is to have the menu on a separate framebuffer – or a combination of framebuffers - then render that to a texture. To capture user input, the game would then ray cast to the quads onscreen to figure out which button the user was selecting. This solution was not picked because of some performance issues that were happening in the beginning of the project and also the availability of ImGui, which integrates well with both GLAD and GLFW.

GUIs are used when:

- At the start of the game, to show that the level is loading in the background
- Whenever the player pauses the game, the main menu pops up with some options
- By the HUD in the game, which is represented by the health, items and weapons of the ship

Img. GUI overlay showing the menus

4.7.8. Combination of Lights and Meshes to Create Visual FX

Since lights are also *IComponents* in this architecture, it is possible to combine lights and entities, which can be seen in the projectiles. Once a projectile is fired, it will illuminate its way, making up for an interesting 3D weapon effect.

Both the Laser Blaster and the WTW Cannon make use of this feature.

4.7.9. Offsetting Vertices in the Shader

The reactor is capable of “exploding” once the player shoots at it or collides against it. The effect that can be seen is a red wave that crosses the reactor room and blows the corridor door away into space.

This effect offsets the vertices of the reactor capsule along its normal, causing it to grow over a few seconds.

```
float vertex_offset = 0.0f;
if (enable_offset_along_normal)
{
    if (shaderFeature->parameters.FindParameterByName("VERTEX_OFFSET", p)) vertex_offset = strtof(p.value.c_str(), NULL);

    /*Check for an internal parameter in the shaderfeature. This is where the last offsets will be stored.*/
    if (!shaderFeature->parameters.HasParameter("LASTVERTEXOFFSET"))
    {
        /*Updates offfset*/
        vertex_offset += vertex_offset * (float)frameTime.GetDeltaTime();
        /*Does not exist. Create variable and store in the container.*/
        shaderFeature->parameters.Add("Vector2D", "LASTVERTEXOFFSET", std::to_string(vertex_offset), true);
    }
    else
    {
        /*Exists, replaces the old value with the current one.*/
        if (shaderFeature->parameters.FindParameterByName("LASTVERTEXOFFSET", p))
        {
            float lastOffset = strtof(p.value.c_str(), NULL);

            /*Updates offfset*/
            vertex_offset = lastOffset + vertex_offset * (float)frameTime.GetDeltaTime();

            shaderFeature->parameters.SetValue("LASTVERTEXOFFSET", std::to_string(vertex_offset));
        }
        else { /*Not possible!*/ }
    }
}
shader->SetVertexOffset(vertex_offset);
```

Img. The explosion effect driven by *GameSystemVideo`s WriteShaderFeature(…)*

4.7.10. Texture Effects

There are some occasions in the game where texture effects are used in the shader, some of them are listed below:

- The reactor room uses texture blending to blend a regular ceiling/floor textures with the red spot where the reactor is located
- The WTW Cannon projectile uses a spherical texture that is offset over time
- The FBO texture, explained here

4.8. Physics

BellEngine.Physics.Custom is a self implemented physics handling system that runs in a separate thread, started once the level object is completely loaded. Similarly to the other systems in this engine, it does not hold any data but only processes it over time, given it is fed the current time information - *Update(…)* and one scene - *SetScene(…)*.

4.8.1. System Configuration

This system is also configured with the use of *SetFlags(…)*, that takes an *unsigned long long* and check for bits down/raised during the gameplay. Meaning that it is possible to disable collision response among a subset of colliders.

To illustrate this, the *GameWorld* will disable some of the default collision response options, such as Box-Box, Sphere-Sphere, Sphere-Capsule and others. This uses an OR (|) operation to generate the configuration value. The *SystemPhysics* will deserialize this data by performing the inverse bitwise operation with an AND (&).

```
this->systemPhysics = this->systemFactory->GetPhysicsSystem((SystemFactory::PhysicsSystem_e)gameConfig->physicsOptions.engine);
if (this->systemPhysics)
{
    this->systemPhysics->SetFlags(DISABLEDEFAULTRESPONSE_SphereToSphere | DISABLEDEFAULTRESPONSE_BoxToBox | DISABLEDEFAULTRESPONSE_SphereToCap
    this->systemPhysics->BindCollisionCallback(std::bind(&GameWorld::OnPhysicsCollisionCallback, this, std::placeholders::_1));
    this->systemPhysics->Activate(this->gameConfig->physicsOptions.isActive);
}
```

Img. Example of configuration

4.8.2. Ray Casting

Ray casting was implemented for every collider in the physics system. This test is outlined by the *ICollider* interface and returns true if a ray intersects a collider in the game, false otherwise.

```
virtual const bool Raycast(const Vector3D& rayStart, const Vector3D& direction, const float& rayLength) = 0;
virtual const bool Raycast(const Vector3D& rayStart, const Vector3D& direction, const float& rayLength, ParameterContainer& output) = 0;
```

Img. Ray casting on the *ICollider* interface

This technique is used in game for the underlying “auto-aim” for the player, as follows:

- If there are no enemies around the player, when a projectile is fired, the direction of the impulse is towards a fixed point ahead of the player
- If an enemy is in close range and in the field of view, the player will ray cast to the enemy to make sure it is in target, then fire the projectile

```
/*Check if player is facing the enemy*/
if (camera->IsFacingEntity(enemyEntity))
{
    enemyRB = enemyEntity->GetComponent<IRigidbody>();
    /*Raycast to the enemy to make sure it is within the AIM of the player*/
    if (enemyRB->GetCollider()->Raycast(playerPosition, playerRB->GetForwardInWorld(), radius))
    {
        enemyPosition = enemyRB->GetPosition();
        currentDistance = glm::distance(playerPosition, enemyPosition);
```

Img. Player ray casting towards an enemy

Ray Casting is also used by the *ShootAt* AI Behaviour in the AI system, which uses this technique to find our if there are obstacles between an NPC and the player before shooting towards the player.

4.8.3. The Update Loop

The update loop is the most critical part of the system, it takes all entities in a given scene, integrates over time (applying gravity, acceleration, friction, etc.) each enabled entity, then checks for collisions amongst their rigid bodies.

All rigid bodies in the engine have colliders that are updated during runtime as the game progresses. Colliders hold information of the rigid body shape, including:

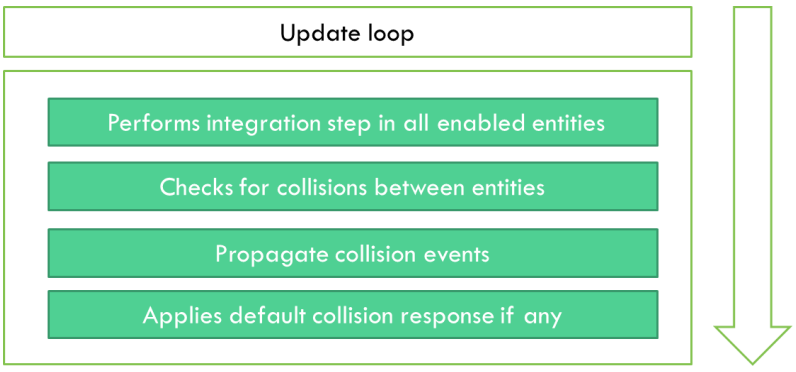
- Sphere
- Plane
- Box
- Capsule
- AABB

This information is used during the collision check portion of the loop to figure out which colliders have intersected.

After all collisions have been properly detected and captured, the loop will then provide the appropriate collision response for each collision pair, if available and propagate collision events to the game. Thescent makes heavy use of this technique as the game has logic that relies in "ghost objects" or collisions between special entities and the player. Examples of that include:

- When the player collides with the first door in the game, the Sphere-Plane collision generates an event that causes the door scripted animation to open
- When the player collides with the button in the first room, the Sphere-Box collision generates an event that causes the second door to open on the other side of the map
- When the player or the enemies collide against bullets, their health are decreased because of the Sphere-Sphere collision event generated
- When a bullet collides with the reactor, causing the reactor to explode, it is generated by a Sphere-Capsule event
- When the player leaves the station at the end of the game, the scripted animation is triggered by a Sphere-Box event, which propels the ship to be accelerated towards the space
- When the player comes in contact with the Anticloak device (Sphere-Plane), the WTW Cannon (Sphere-Sphere) and the Hostage Cage (Sphere-Box), sounds are triggered

Collision events are generated with the use of C++11 *std::function* callbacks that are registered once the system is created by Thescent's constructor.



Img. Update loop layout

Because the nature of this system is threaded, there are many protections put in place to avoid crashing errors (more in the Algorithms and Gems section). These may include the use of critical sections in the entities, rigid bodies, etc., try-catch for expected C/C++ SEH exceptions and signals for disposing entities (*IDisposable::IsDisposing()*).

4.9. Sound

The sound system uses the low-level OpenAL API to play sounds in game. This system has its own update loop that constantly checks if any entity that has a sound attached to it has its flag raised.

The engine provides a sound component that contains an *IsActive()* state. Whenever the game wants to play a sound, it calls *Activate(true)* on the desired component and the sound system will pick it up on its next iteration.

Sounds can be triggered by different kinds of in-game events. The following are examples of audio usage:

- A background theme song is played right from the start of the game and looped until the end
- When firing weapons, there will be a corresponding sound effect
- Doors make sound when opened
- Doors that are blocked will emit an “access denied” sound
- Once the hostage is rescued, the boss is placed in-game and a roaring sound can be heard
- A voice over can also be heard when the player acquires a new weapon and a new item

Thescent uses a variety of wav sounds and music, under free licensing. In addition to that, the author’s voiceover can also be heard in the game.

5. Gameplay

The main goal of the game is to rescue the hostage, defeat the enemy drones and the 'boss', then finally destroy the reactor. At the end of the game there is a small cutscene where the ship flies into open space.

Thescent features a short map composed of four main areas:

- The entry corridor
  This is the first area in the game, where the ship is positioned at the start. The corridor is closed by a door behind the player and connects directly to the reactor room just ahead.
  Once the reactor explodes at the end of the demo, the door that was previously blocked will be blown into space. If the user moves along the corridor again, an animation sequence will be played, showing the ship fleeing the station.

- The reactor hall
  In this hall players will face the turret enemy and the boss at later stage. At the start of the game players can also see the reactor, protected by iron bars at the farthest point of the room.
  Once the boss is defeated, these bars are lifted and the player can either shoot the reactor or collide with it to cause an explosion.

- The left room
  This room can be accessed directly from the reactor room. Here players can find the first enemy drone, a button to open the left room door and the Anticloak game item.

- The right room
  This room can be accessed directly from the reactor room, once the player has pressed the button to open its door. In this room, players will find the second enemy drone, the hostage and the WTW Cannon.

5.1. Controls

General		
Key	***Modifier***	***Action***
ESC	-	Exit simulation
P	-	Pause / show main menu

Player		
Key	***Modifier***	***Action***
W	-	Move forward
A	-	Strafe left
S	-	Move backward
D	-	Strafe right
Q	-	Move up
E	-	Move down
SPACEBAR	-	Halt ship
N	-	Toggle special vision to see invisible objects
1	-	Select Laser Blaster
2	-	Select WTW Cannon

Mouse Scroll Up	-	Zoom-In Target
Mouse Scroll Down	-	Zoom-Out Target
Mouse Movement	-	Move the Descent's crosshair
Mouse Left Click	-	Shoot with selected weapon

5.2. Game Menu

The game features a main menu, technically described here. The menu tree is as follows:

- PLAY
  Allows the player to start the game or go back to a paused game

- OPTIONS
  - Toggle Physics colliders
    Will show the physics shapes around the game entities

  - Show developer window
    Show a small window with the current GUI framerate and other buttons to assist the author during development.

- CONTROLS
  Displays the key bindings for Thescent

- CREDITS
  Lists information about the project, the authors and the professors at Fanshawe College.

- EXIT
  Leaves the simulation

6. Developer's Notes

During the development of Thescent several issues were found and solved in the game – some are still to be worked on. This section features some highlights and comments.

6.1. Issues faced during development

- Difference between Intel and Nvidia graphics cards
  This project was mainly developed using an Intel Graphics card, which are quite forgiving regarding how shaders are setup in code. In several occasions, the game would run perfectly fine on the Intel card but give a black screen, or artifacts, on Nvidia-based computers. Most of these issues were resolved by reading tons of articles on the web about the matter.
  One of these was about non-initialized color variables in the shader, which would lead to noise in Nvidia cards.

- Slow performance when updating graphics
  One of the problems Thescent faced was caused by a major bottleneck in the VideoSystem. This happened because the game didn't have customized C++ shader-classes to hold the values of the uniforms. So, the game would call glGetUniformLocation every frame for every shader and every entity in the game.
  This was fixed by implementing these pipes. Now, uniforms are only queried if the shader program id changes.

- Threading occasional null pointers
  Threading is a sensitive subject in every piece of software, this is no different for Thescent. Because of the short time to develop the POC, some synchronization errors will still appear, even though there are quite a few data protection strategies being applied to the game.
  Because this issue is expected behavior for this project, the code makes use of the familiar try/catch formula to handle SEH exceptions.
  This, however, does not affect gameplay outside Visual Studio at all.

- Frame-time errors
  There was a big issue regarding the timestep, where the game would speed up when left idle for some time. This was solved after reading the excellent article about timestep strategies at Gaffer On Games.
  Now the code locks the frame time to a maximum of 60fps for all threads in the game. With the exception of the main thread, if a 'frame' was faster that the expected, a thread will consume the remaining time by sleeping.

- The importance of proper engine work
  Because of the fast pace of the program, students often fail to build a proper engine from the start. This proved to be problematic until the beginning of 2019, when this version of BellEngine was written.
  In this case, the engine shifted from mostly OOP to ECS, which allowed for code decoupling and the multilayered system present in the final version.

- Graphics and animation are slow processes of development

These are complex subjects of study and require time and a lot of dedication to get it right. OpenGL and shaders were designed in a time most developers used pure C / assembly code to come up with games, so it is understandable that the interfaces to them requires a great deal of attention if there is no upper-level framework to handle that.

- Physics is a though subject to simulate
  Physics is complex by nature and ins software development it is no different. Here we can cite problems with floating point errors in integration systems, collision checks that failed and collision responses that do not feel natural to the player.
  This is was one of the areas that Thescent had more work and certainly more once the system was detached and threaded later.

6.2. For the future

- More Lua scripting
  Lua is a very flexible and useful tool for developing scripted sequences and other game features. As a developer, I wanted to do more with that and I'll keep studying after the GDP. Also, I can read the documentation that is written in Portuguese, which helped many times when I was looking for information on the matter.

- Shaders
  There was some shader-material that I wanted to cover originally and that did not make to the final version.
  The last feature that was implemented was the multi shader program support, which wasn't thoroughly tested and might need some reviewing later on. Because of the short time I had, I also could not separate all the sub-shaders in the uber shaders. In order to comply the requirements, I left two shader programs active in code, which effectively show that the game has the multi program support.
  Another aspect was the different applications of the geometry shader, which was also an interesting subject to study and that'll pursue.

- C# Support
  I'm interested in doing a conversion of BellEngine to C#. Now with .NET standard and .NET Core, cross platform development was made much easier and be entirely done on Visual Studio or Mono. Some libraries, such as GLFW, already offer implementation in C# for that matter.

7. Supporting libraries

The following third-party libraries were used in this project and all copyrights and credit are to be given to their respective teams.

- GLFW - https://www.glfw.org/
- GLAD - https://glad.dav1d.de/
- Dear ImGUI - https://github.com/ocornut/imgui
- OpenAL - https://www.openal.org/
- Bullet (part of the solution although not used in the final release) - https://pybullet.org/wordpress/
- Assimp - https://www.assimp.org/
- Lua C binder - https://www.lua.org/download.html
- RapidXML - http://rapidxml.sourceforge.net/
- SOIL - http://www.lonesock.net/soil.html

8. Supporting Hardware

This project was developed on a Dell Inspiron 7348 2-in-1, equipped with:

- Intel Core I7 5500U
- Intel HD Graphics 5500
- 1x8GB DDR3L
- Samsung EVO 500GB SSD
- 1920x1080 Touchscreen

9. References

In addition to the official documentation for each component used in this project, the following resources were also helpful during development:

Books:

- Real-Time Collision Detection, by Christer Ericson - Elsevier, 2005
- OpenGL – Build high performance graphics, by Muhammad Mobeen Movania, David Wolff, Raymond C. H. Lo, William C. Y. Lo

Links:

- http://wwwtyro.github.io/
- http://ogldev.org/www/tutorial33/tutorial33.html
- https://learnopengl.com/Advanced-OpenGL/Instancing
- https://www.khronos.org
- https://www.shadertoy.com/
- https://stackoverflow.com/
- https://docs.unity3d.com/Manual/index.html
- https://www.turbosquid.com/
- https://free3d.com/
- http://soundbible.com/
- https://incompetech.com/
- https://gafferongames.com/
- https://www.tomdalling.com/blog/modern-opengl/08-even-more-lighting-directional-lights-spotlights-multiple-lights/

10. License

Thescent is a Final Project for Fanshawe Colleges' Game Development: Advanced Programming (GDP) Post Graduate program. This software was built for demonstration purposes with the intent to display a range of subjects lectured during the course.

Thescent and BellEngine are free to be used and distributed, granted attribute to the author is given whenever necessary, as per the MIT license below. Attribute is also to be given to Interplay Entertainment, as this project attempts to replicate the works of Descent (1995).

Moreover, the third-party libraries used in this project are held under their respective licenses and no copyright infringement is intended with the distribution of this software.

For more information, contact the author.

11. Credits

To the teaching staff at Fanshawe College, School of IT, in alphabetical order:

Staff member	***Courses***
Feeney, Michael (Program coordinator)	Graphics I and II, Physics I, Game Engines and Patterns, Animation
Gustafson, Lukas	Configuration and Deployment, Networking
Kelly, James	Artificial Intelligence
Lara, Oscar	Multimedia Fundamentals, Game Component Integration
Lucas, James	Physics II
Maclam, Daniel	Algorithms and Gems

12. The Author

Felipe Bellini is a Control Systems engineer specialized in software development for mobile and personal computers using a variety of technologies. Over his career he has worked with software development teams all over the world, with the most diverse areas of expertise.

In addition to that, he is a martial arts teacher with over a decade of experience under the International Traditional Kung Fu Association.

For more information, contact the author at:

- Email
- Portfolio
- LinkeIn
- Github
- YouTube