Aeon Engine

Name: Albert Bennett

Student Number: S00100640

Supervisor: Neil Gannon

**Contents:**

* **Introduction**
* **Description**
* **Background Information and Resources**
* **Requirements Specification**
* **Project Layout**
* **Game Engine**
* **Engine Architecture**
  + Initial Research
  + MVC
  + COM
  + MVC vs. COM
  + Implementation
  + Problems Encountered
* **Real Time Lighting** 
  + Initial Research
  + Forward Rendering
  + Deferred Rendering
  + Lighting Pre Pass
  + Lighting Pre Pass vs. Deferred Rendering
  + HLSL Research
    - Phong Shading
    - Lambertian Shading
    - Phong Shading vs. Lambertian Shading
  + Implementation
    - Shaders
      * ClearGBuffer
      * Point Light
      * Basic Material
      * Compose
    - Lighting Pre Pass
      * Opaque Pass
      * Light Pass
      * Composition
  + Problems Encountered
* **Content Pipeline Extension**
  + Implementation
  + Problems Encountered
* **Summary**
* **References**

**Introduction**

The inspiration for Aeon Engine came from a 3D menu system I created February 2012. At the time I had created it because I wasn't allowed to work on my third year project because I had done too much with it. The original idea behind Aeon Engine was to create a UI api that works with XNA and would allow users to create 3D and 2D menu systems quickly and easily though a UI creation tool. The tool would create and package the files created by the UI api into a zip folder as well as all of the resources that was used to create the menu system e.g. models, Textures. The created package would then be unzipped when the game was being installed on a computer.

The main aim behind Aeon Engine was to allow users to create good looking, complex menu systems quickly and easily though a development tool. Over the summer of that year I had started creating the UI api and when it was finished, it did exactly what I had planned it to do minus the UI creation tool and the packaging of the created resources. When the original api was finished I tried to use it in a side project of mine, however it didn’t work to well, nor could it be effectively controlled outside of the api. This defeated the purpose of creating it in the first place. So step two was to remake it but, do it better. I tried and I tried but. It wouldn't work at least, not well enough to be used in a game. So I tried to think of a solution to make it more flexible. I eventually came to the conclusion that I needed to encapsulate it inside of a game engine. This was when I did my initial research into game engines. By summers end I had learned a lot about how game engines should work and about their architectures. With what I knew then I decided that my fourth year project should be a game engine that would have physics, 3D rendering and a UI api.

However, when I first proposed that idea to my supervisor, he felt that it was too big of a project to do so. We had to downscale it and I thought that the most interesting aspect of any game engine is the lighting. More specifically real time lighting so after further research into real time lighting I had decided that my project would focus on lighting pre pass. I also wanted my project to include a rudimentary game engine so that it could manage lighting and various other aspects of the project such as input handling.

**Description**

Aeon Engine is a 3D real time lighting sample built on top of a rudimentary game engine which has a custom COM architecture implemented.

**Background Information and Resources**

The main technology used throughout the project is XNA and high level shader language (HLSL) both of which I have programmed inside of Microsoft Visual Studios 2012. I’ve also used both Blender and Photoshop, to create the models that have been used to test the system.

There is a vast array of samples I’ve downloaded and books that I’ve read that have helped me to understand how real time lighting works in games and how to write HLSL. These are all referenced below in the reference section.

**Requirements Specification**

What is required of Aeon Engine to be able to achieve at the end of the project is, that it should be able to render multiple lights on a 3D model in real-time. It should also be able to render, a texture, normal map, specular map for any 3D model. Furthermore, the lighting technique employed should be managed by a rudimentary implementation of a game engine.

**Project Layout**

At the start of the planning stage of Aeon Engine I had decided that the project would be broken up into three distinct parts. Those are:

1. The game engine itself.
2. Real time lighting.
3. The content pipeline extension.

In each section I will explain my initial research into a particular aspect of Aeon Engine, what persuaded to implement it in that way, and my implementation of that particular aspect into Aeon Engine.

**Game Engine**

A game engine is a collection of frameworks that are designed to handle one specific module of work such as rendering or physics. However a game engine goes beyond this and is used to bring together all of its frameworks to solve a particular problem.

The first step in creating a game engine is to define what game engine architecture to use. Next is the implementation of said architecture. Then it is the creation of the various frameworks/ modules that are to be used in the game engine. The final step is to bring together all of the various frameworks/ modules that have been created.

**Engine Architecture**

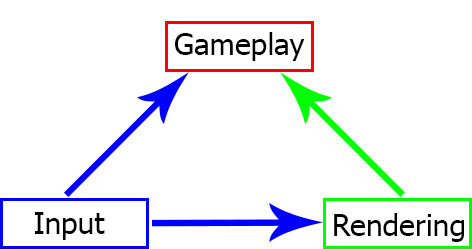
A game engine architecture is a particular pattern and/ or set of rules to follow when designing the various frameworks/ modules required by the game engine.

**Initial Research**

In initial research I did into game engine architectures. I found that there are two main architectures available for games engines, MVC (Model view controller) and COM (component object model). These two aren’t the most commonly use game architectures used but they are the two most recently discovered ones which have been implemented into some of the most commercially massive game engines available such as Unreal Engine 4 and Cry Engine 3.

**MVC**

The MVC pattern is used in game engines as an architecture to help separate out the input logic, game logic and rendering. The primary appeal to MVC is the ability to change things quickly with little reworking especially in the early stages of the games development. Below is a diagram that depicts the simplest logical representation of MVC architecture:

****

Gameplay = Model.

Rendering = View.

Player input = Controller.

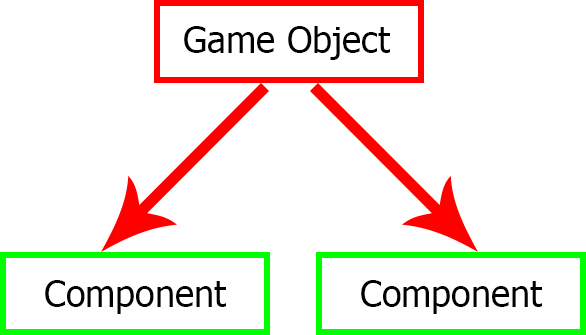
In short, the controller takes input from the player which changes the model. The controller then passes the model and all relevant data to the view to be rendered. All changes made through the model by the controller are known as interfaces.

The main advantages of using MVC are:

* Decoupled rendering: In the majority of game engines the game world is tightly coupled. However MVC decouples the game world from the rendering. By using the model to render. This in turn simplifies the process of introducing multi-threading support to the game.
* Quick to change: With an MVC pattern implemented it is easy to change aspects of the game engine with little reworking because of the level of separation that MVC provides between rendering and input. This in turn allows for greater cross platform support.

**COM**

Unlike MVC, COM (component object model) is also used as a game engine architecture to help separate out logic however. The main appeal to using COM is the ability to reuse vast amounts of code. This is achieved because COM is implemented using “components” and “game objects”, where every “game object” can have many “components”. Below is a diagram that depicts the simplest logical representation of COM architecture:

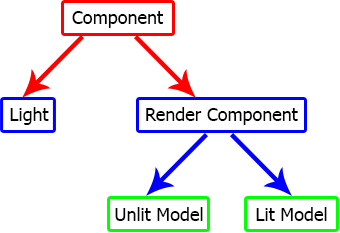


In all cases a game object will hold a collection of components that it will manage. Each component attached to a game object must do one thing and one thing only such as rendering a model. Each component in COM is in itself a separate entity such that each component shouldn’t be dependent on any other component.

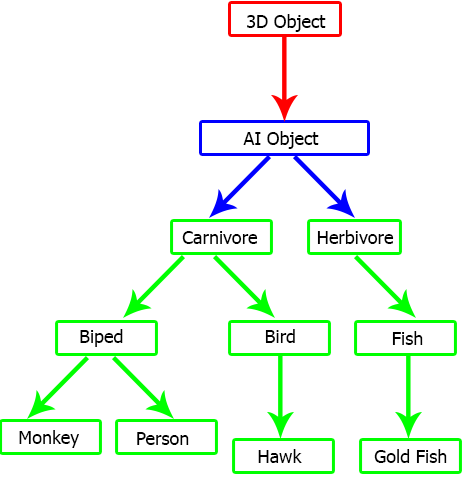
The main advantages of using COM are:

* Code reuse: Due to its design, game engines following the COM architecture will have a shallow level of inheritance as opposed to the traditional deep level of inheritance thus reducing the amount of reworking need to make changes to certain aspects of the game engine. It also means that the majority of code used can be reused even in other projects. The diagrams below illustrate both a shallow level of inheritance and a deep level of inheritance.

Shallow inheritance



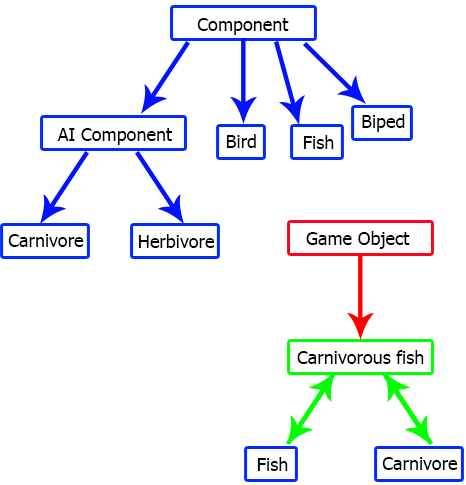
Deep inheritance



From the diagram above it would be near impossible for an object which has fish like characteristics to be a “carnivore” because the object “fish” inherits from herbivore and not carnivore. When a problem like this presents itself inside of a deep inheritance there are two possible solutions:

1. Create another object that inherits from carnivore which has all of the same characteristics as “fish”. The problem with this approach is that you will be copping most of the same code from the “fish” object. All though it’s reasonably quick to implement this solution the amount of work required maintaining the code has now gone up considerably and so has the amount of code.
2. Rearrange the code and inheritance model to now support a “fish” being a “carnivore”. This approach could take a long time to implement. The end result being an increased amount of objects, which in the long term could lead to memory problems, increased complexity and reduced flexibility in the project. It also means that maintenance of the code has also risen due to the increased number of code files.

However, consider the introduction of a shallow level of inheritance one that can be achieved by implementing COM architecture. Now the problem has been simplified because of components. Each component is in itself a separate entity which is created to perform a particular task. Now an object which has fish characteristics will have several separate components defining it. Below is an example of how a carnivorous fish can be implemented in a game engine following COM architecture:



From the diagram it is clear to see how implementing COM architecture into a game engine can greatly increase the amount of code that can be reused, the flexibility of the project and also, how using components can reduce the complexity and code maintenance for the project.

* Modularity: It is because of component involved in COM architecture that gives it is modularity. Each component is an independent entity that in most cases isn’t relying on any other object. This means that every component (in most cases) can be replaced with any other component or none and the game engine will still run without errors.
* Quick prototyping: COM provides a strict set of rules to follow when implementing it. These are:

1. No two game objects in the same project can have the same unique identification name.
2. No two components attached to the same game object can have the same unique identification name.
3. Components must be able to send messages to other components through the game object.
4. Each component attached to a game object must be managed by that game object.
5. When a game object has been destroyed so should all of its components.
6. When a game object has been disabled all of its components should also be disabled and vice versa.

With these rules in place, when reusing components from other projects the only thing that should be changed is the namespace for the components. This means that the amount of code that needs to be rewritten can be reduced. Overall resulting in: games being quicker to prototype.

* Reduce conflict: With the implementation of COM large teams can be working on separate aspects of the same task such as a health component, ai component, rendering component for a player object. The natural separation of objects that COM provides will lead to less conflict among team members.

**MVC vs. COM**

There are merits for using both MVC and COM as a game engine architecture. When it came down to selecting one of them for my project it had to be COM because:

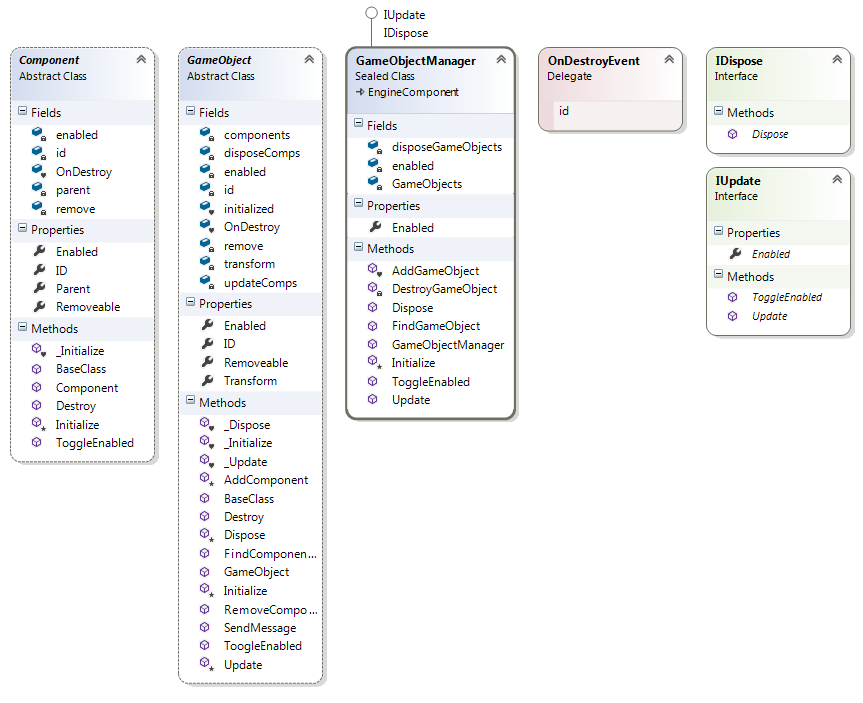
* COM is a game engine architecture that I’m familiar with mainly, because I’ve been using it since February of last year. I even made an attempt at implementing COM into the game engine I made as part of my third year project. However I only really started researching it when I was on work experience last year.
* Before I researched COM all of the game engines that I’ve previously created were all unsuccessful except for one. They all failed because whilst I was creating them, the projects got complicated quickly. Therefore for me the major appeal for implementing COM into a game engine is the simplicity that it gives to the project due to the level of separation involved in the implementation of it and also the reusability\ modularity of the code involved.

**Implementation**

My implementation of COM is different from all others that I’ve seen throughout the course of my project. Not to say that my project doesn’t follow the same rules of COM it’s just that in my project I’ve had to implement a similar system twice. The first implementation is to define game objects, components and the second time is to define engine components.

Firstly I will explain the main implementation of COM into the project and then I will explain why I felt the need to implement a similar system in a second time.

Below is a class diagram showing the various classes that I’ve created in order to implement COM into my project the first time:

****

When implementing COM into any game engine the most important class that must be created is the component class. From my research the component class must be an abstract class meaning that an instance of it can’t be created, instead in order for something to be a component it must inherit from “Component”. Also a component must have:

* A unique identification name.
* An owner game object (but in some cases this isn’t necessary).

Also it should be able to perform the following tasks:

* To destroy itself.
* To be able to be enabled or disabled and to allow the toggling of these.

The above are the fundamentals for implementing components in to a game engine. I’ve also included some other features to “Component”. These are more helper functions than anything else. One such item is a delegate entitled “OnDestroy”.

The second most important class to be implemented when attempting to implement COM into a game engine is the game object class. The primary function of a game object class is to manage all of the components attached to it. Initially I had some problems with this. The main problem I had was how to manage the different components, if they were all supposed to do different tasks. An example of this is that if I had two components, one which is supposed to be rendered only and one which is supposed to be updated only. The solution came in the form of interfaces. When the game object decided to add a component to it, it would check to see what interfaces the component inherits from and then it would add the component to the appropriate collection(s) .Where the game object would hold a number of collections each one containing components that will be managed differently at different points in the games update cycle.

From the research I had done on COM, showed me that like components, game objects must also be an abstract class. When I was implementing COM into Aeon Engine my game object class entitled “GameObject” it had all of the fundamentals for a game object from COM, these are:

* A unique identification name.
* A transformation matrix.
* A collect of all of the components attached to it.
* A means of enabling and disabling it.
* A means to send messages to all of the components attached to it.
* A method to update it (but this is debateable).
* A means to removing a specific component.
* A means to destroy it.
* A means to find a specific component.

I also felt the need to add some extra functionality to “GameObject” such as a delegate to signal the destruction of a game object and an initialized property. These are used primarily by the “GameObjectManager” class to help manage it.

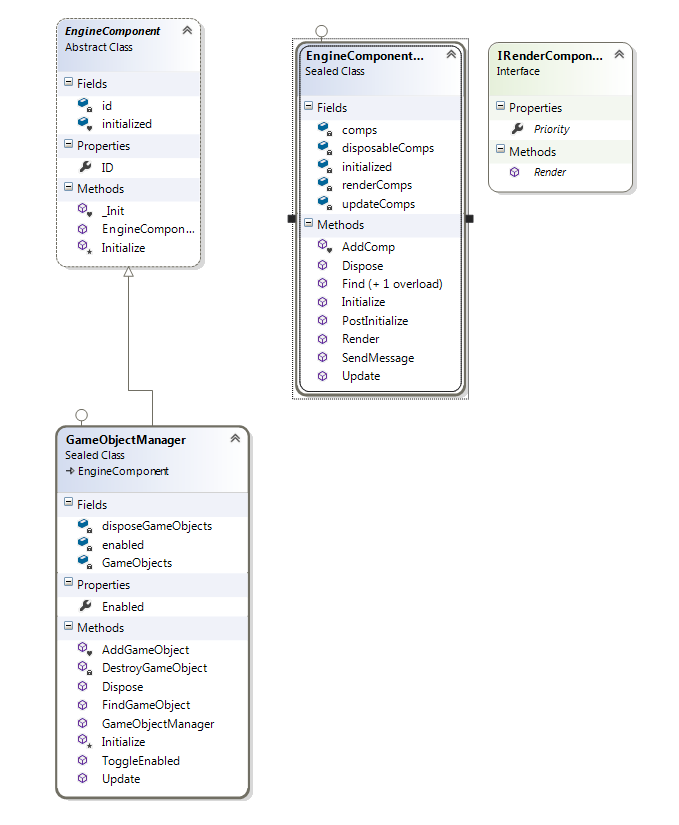
The “GameObjectManager” class is used to manage every game object in the game and it is managed by the “EngineComponentManager” class which is explained further down.

When a game object is initialized it adds itself to the “GameObjectManager” class. This is to help with the automation of the game engine and also to check to see make sure that no two game objects have the same identification name. Therefore this means that there is one central location for every game object in the entire game. Helping separate out the game engine even more so. Also it has the added advantage of being able to search through every game object in order to find a specific game object or even to send a message to all/ a specific game object.

A message in COM terms generally refers to the name of a method and the parameters for the requested method that is required to be invoked by a particular engine component/ component that can also return the result of the invoked method if required.

I felt the need to implement a similar system to COM a second time into my project because to me a framework/ module (engine component) should be managed automatically. It shouldn’t have to rely on the engine to update/ render it nor should it be a regular component like a render component. It should be its own separate entity.

This part probably could be avoided because there are so few actual frameworks in place. However I still felt that it was a necessity for Aeon Engine. Below is my implementation of COM a second time:



My solution, to better automate Aeon Engine is to implement two new classes “EngineComponent” and “EngineComponentManager”.

An engine component is similar in many ways to a component except, an engine component has far less functionality. “EngineComponent” defines a framework\ module that is managed automatically by the “EngineComponentManager” class. When an “EngineComponent” is instantiated it adds itself to the “EngineComponentManager”. In here, the “EngineComponentManager” manages, the initializing, updating, rendering and disposing of every “EngineComponent” attached to it. This might seem somewhat simplistic; however the “EngineComponentManager” is managed by another class entitled “Framework” which is used to manage the game engine. “Framework” is used to create\ bring together all of the other modules\ frameworks needed for the game engine. It does this by first creating all of the necessary modules such as the “GameObjectManager”. Next “Framework” loads in a file with a specific name. This file contains all of the data required to set up the remaining modules\ frameworks required. Finally the input manager is created.

I’ve decided to create the remaining modules\ frameworks through the use of an external file because, this makes the game engine itself more modular, instead of having to hard code the creation of various modules\ frameworks. This means that any engine component that is correctly defined in the external file is created and managed automatically.

However, the solution above poses a major problem that being; every module\ framework is different and some modules might need to be rendered whereas others need to be updated and some might even need to do both. I devised a solution to this problem being, the implementation of interfaces. In Aeon Engine there are several interfaces included, which help solve this problem those are:

* IUpdate: This is used by engine components and components that are required to be updated in some form.
* IRenderComponent: This particular interface is used to define an “EngineComponent” which can be rendered. It contains a property “Priority” which notifies the “EngineComponentManager” what order the “EngineComponent” must be drawn.
* IDispose: When IDispose is inherited from, it allows for the disposing of various aspects of an “EngineComponent”\“Component”\ “GameObject” when that particular instance is about to be destroyed.

The “EngineComponentManager” class is the only class in Aeon Engine that has to be managed directly. Like “GameObject”, “EngineComponentManager” manages all of the “EngineComponent” attached to it in a similar way. However, unlike “GameObject”, “EngineComponentManager” has to manage the rendering of “EngineComponent”. This done in the “Render” method, all “EngineComponent” in the “renderComp” collection is rendered based on the value that they have for “Priority”. Thus the higher “Priority” is the later the “EngineComponent” will get rendered.

**Problems encountered:**

When implementing COM into Aeon Engine I had many problems. These include:

1. Problems with the destruction of components.

**Problem:** The destruction of components.

When a component was destroyed and it had been added to a collect outside of the game object it was stilling included in that collection. The problem first appeared when I had started implementing lighting pre pass into Aeon Engine.

**Solution 1:**

The first solution to this particular problem was simple I added a variable to every “Component” and “GameObject” entitled “OnDestroy”. “OnDestroy” is a delegate that when called, signals the destruction of an item. After it is called a property entitled “removable” changed its value from false to true. This is checked by the class in which the collection in question resides in, so that it can remove the “Component” \ “GameObject”. This solution is only when the “Component” is unaware of what collect it has been added to.

**Solution 2:**

This solution is event simpler that the one before it. The logic behind this solution is that every “Component” of a certain type is pre added to a list and when it is destroyed it removes itself from that list. This solution is only used when dealing with the render section of Aeon Engine.

There are other problems I had when implementing COM into Aeon Engine but, these are explained in the content pipeline section of the report. These problems were encountered when I had tried to create engine components from a file. Along with those problems there were others encountered when I was implementing the real time lighting into Aeon Engine which is related to COM, but those problems are explained in the real time lighting section of the report.

**Real Time Lighting**

Real time lighting is the means by which games illuminate models in real time.

**Initial Research**

Before 1995 3D graphics was unheard of in games. However, this all changed with the release of Quake. Due to that fact that, Quake was the first game to contain 3D graphics. With this, more games were released rendering 3d models. When those games became popular it persuaded several hardware vendors to start offering 3D accelerated graphics cards at consumer prices.

Over the years many 3D accelerated graphics cards have become available, one of the first being TNT by NVidia, which was shortly released after Quake in 1995, to the Quadro 6000 also by NVidia, released in 2011. NVidia’s GeForce graphics cards had some of the most revolutionary technology in them. The GeForce is considered as the first graphics processing unit (GPU), capable of preforming transformations and light calculations that were previously done on the CPU. However it was the Geforce3 that was the first GPU to feature hardware assisted vertex and pixel shading. These “shaders” allowed developers to manipulate geometry and pixel data directly on the hardware. By switching from processing graphics on the CPU, to the GPU this allowed for more cutting edge effects to render in real time.

It wasn’t until 2004 when the demand for more realistic graphics in games became apparent. As a result many lighting techniques and calculations have been invented to try and convoy ever more realism in games. Multi pass lighting techniques so far have been proven the most effective, when convoying realism in games and many modern day games employ these techniques. As such there are three main techniques available to games by which they can illuminate 3D models in real time. These are:

* Forward Rendering.
* Deferred Rendering.
* Lighting Pre Pass.

Some game companies such as Insomniac have employed more that one of the above techniques to solve problems with multi-pass lighting such the introduction of forward rendering to help render transparent geometry.

**Shader: -** A custom shading and lighting procedure, used to define the rendering of a pixel\ vertex.

The term shader was first introduced in 1989 by a program created by Pixar entitled “RenderMan”. “RenderMan” takes in the entire description of a scene, camera position, and model geometry through to the final render; however it wasn’t used until the 1995 release of Toy Story.

**Forward Rendering**

Forward rendering is rarely used in games these days. It was common place on consoles such as the PlayStation 2 and Xbox, because of the limited performance of their graphics cards.

For every model that was going to be rendered to the screen, required a shader to be applied to it. This shader would define how that model would be affected by a proposed light source. Thus, forward rendering only allows for one light source to be applied to a model at a time, meaning that many programmers employed the use of light maps to give the appearance of multiple light sources being applied to a model. This however had its down falls such as incorrect shadowing of models.

Advantages:

* Low performance required.
* A large variety of material s can be created. This is because in forward rendering the shader has to access the lights properties and material properties. Accessing light information in lighting pre pass and deferred is impossible because of how the shader have to be structured in order to achieve multiple lights in a scene.

Disadvantages:

* Only one light source can be applied at a time.
* Incorrect shadowing when using light maps.
* Light calculations are done with every pixel, because of the shader model available it was impractical to check if a particular pixel can be lit.

When technology improved the advantages for using forward rendering as a method for rendering models was lessened. Eventually newer, more effective methods for rendering models in real time were required.

**Deferred Rendering**

Deferred rendering is the most common technique used when rendering 3D models in real time. Due to major technological changes better graphics cards are now available. These graphics cards can now output geometry and other graphical information on to several textures in the one draw call.

Advantages:

* Multiple light sources can now be applied to a scene without much of a performance hit.
* Simpler management of complex light resources.

Disadvantages:

* Transparency is difficult to implement and is still very limited.
* Too many geometry draw calls.
* The variety of materials are limited because, lighting information is not accessablt directally to the material shader.
* Unable to used multiple materials.

**Lighting Pre Pass**

The main downfall of deferred render is the amount of geometry draw calls needed. As such Lighting pre pass was introduced as a more effective method for rendering 3D models. Both deferred rendering and lighting pre pass have an opaque, depth pass and a lighting pass. However, it is in the lighting pass where things are different. The main differences are the calculation of the light map. Although lighting pre pass has one more pass to it. All of the pass are less hardware intensive that those of deferred rendering. The first pass in lighting pre pass is to calculate opaque data only this being specular data, depth data and normals.

Advantages:

* Lower memory and bandwidth usage. This is because it isn’t necessary to have four render targets anymore and even in some case lighting pre case can be achieved with a single render target.
* Less draw calls to lightning than other methods.
* Better variety for materials. In deferred rendering you are limited to using a single pass per light, lighting pre pass allows for the application of a single pass per mesh.

Disadvantages:

* Transparency is still difficult to implement.
* Only really effective with a large number of visible lights in a scene.
* Less flexible than deferred rendering. This is because one single value can only be stored by the shader for calculating specular power.

**Lighting Pre Pass vs. Deferred Rendering**

I choose to implement lighting pre pass over deferred render for one reason. That being, when compared to deferred rendering, lighting pre pass has far less geometry draw calls per light. Also the geometry draw calls that are involved in lighting pre pass are far less CPU extensive than those of deferred rendering.

**HLSL Research**

When I had finished my initial research into real time lighting I had noticed that among the different techniques available there are also many different ways in which light can be calculated through shaders. Up until then I had only ever heard of phong shading and toon shading. As I delved more in to the subject of light calculations I discovered other calculations employed by game. One such calculation that I had discovered was lambertian shading. This type of shading seemed to be more adverse among commercial games as opposed to phong shading. So having discovered more about both types of shading I’ve decided to show what I have found and what light calculation I decided to use in the light shader of Aeon Engine. Although there are many more light calculations used in games, most of them are derivatives of phong or lambertian shading, such as toon shading.

**Phong Shading**

Phong shading is a calculation that interpolates the surface normal. Then the surface of the normal vector is calculated as an interpolation of the normal vectors. Finally each normal is then normalized and an illumination calculation is used to determine the colour at the current texture co-ordinate.

**Lambertian Shading (Lambertian Reflectance)**

Lambertian shading is used as a means to calculate diffuse reflectance. This is the amount of light that gets reflected from all direction from the surface of an object. It also means that the reflectance of a vertex will not change if it has been transformed around it normal.

**Phong Shading vs. Lambertian Shading**

I decided to implement phong shading to the shaders used by Aeon Engine because, phong shading is because, and phong shading gives a better effect when rendered that lambertian shading. Also phong shading is more easily implemented into lighting pre pass as the luminance model used by phong shading is better adjusted to lighting pre pass.

**Implementation**

I had difficulty when implementing lighting pre pass into Aeon Engine the result of which has lead me to do a number of revisions to the techniques that I’ve employed into the lighting pre pass. Instead of describing all of the failed attempts I’ve made at implementing lighting pre pass into Aeon Engine, I will be discussing my implementation of the most successful revision of lighting pre pass into Aeon Engine.

**Shaders**

In the final version of Aeon Engine there are a total of four shaders that had to be created in order to implement lighting pre pass. These are:

* ClearGBuffer
* PointLight
* Lpp
* Compose

**ClearGBuffer: -** As I have already mentioned a Gbuffer is a set of render targets used to define the necessary data for rendering models in multi-pass lighting. The “ClearGBuffer” shader clears the data already present on the depth, colour and opaque render targets. If their data wasn’t cleared at the being of every render irregularities would occur in the final result such as being able to see what was rendered in the previous render call.

The “ClearGBuffer” works by setting the colour, normal and depth data of the active render targets set to the graphics device to their default values, with normal data being set to 0.5 (this representing a normal that isn’t going to appear as though it is being subtracted or added to the surface of a model) and depth being set to 1 (this is to make sure that there is a value for the farthest depth in the scene).

**PointLight: -** This shader is responsible for defining how geometry is illuminated in Aeon Engine. It is in this shader where phong shading is implemented. A point light is a light that has a radius, position, colour and intensity. Point lights are used to define spherical lights in games such as hanging light bulbs, runway lights etc. In order for point lights in lighting pre pass to work they require opaque data and depth data. This is so that they can make the screen space rendered geometric lights seem to have been applied directly on to the model in the scene and also to allow for the illumination calculations of the normal data for the scene, this so that the model when rendered appears, to have more triangles forming it then what it actually has.

The first thing that happens in the shader is that the position in world space is converted to screen space.

Screen Space position = lp\*lw\*cv\*cp

Where:

* Lp = light’s world position.
* lw = light’s world matrix.
* cv = Currently active camera’s view matrix.
* cp = Currently active camera’s projection matrix.

A screen space position is used to figure out what data has to be modified by the shader .Next a texture coordinate is calculated using the screen space position. Using the screen space position, a colour is taken from the normal data inputted. This colour is then converted in to a useable normal which has a value in the range 0 -> 1. At 0 the normal will appear as though it is being subtracted from the face of a model and at a value of 1 the normal will appear as though it is being raised of the face of the model. Next depth, specular intensity and specular power are received from the inputted textures. Then the actual specular value for the scene is calculated.

Specular = si \* pow (saturate (dot(r, e)), sp)

Where:

* si = Specular intensity.
* r = Direction of reflection between the proposed surface normal and the difference between the point light’s world position and the screen space position.
* e = The direction that the user is viewing the object from.
* sp = Specular power.
* pow = Is a function by which the power of a value is increased by the power of another given value.
* saturate = Is a function that clamps a given value between 0 and 1.
* dot = Is a function that find the dot product of two values.

These are then finally used to calculate phong shading for the scene.

Phong = attenuation \* intensity \* (albedo, specular)

**Attenuation: -** The amount of falloff the light has.

**Intensity: -** How bright or intense the light appears to be.

**Albedo: -** The colour of the light.

**Specular: -** How shiny the object appears to be.

**Lpp: -** This shader is the shader which is used to render models. The texture parameters for this shader are set when; the model is being built using the custom model processor defined in the content pipeline extension of the report. Lpp has two techniques in it; one for rendering opaque data and one for render colour data. The colour rendering technique doesn’t do anything special. It’s just a basic texture shader where a colour is returned based on a given texture co-ordinate. However it is in the opaque render technique where the opaque and depth render targets get their data from.

The very first thing to happen in the shader is the calculation of the depth of the model in screen space. This is done by using an inputted position gotten from the shader and transforming it into a screen space position. This is accomplished in exactly the same way as above. The screen space position is then used to calculate the depth of the model being rendered.

Next a matrix is created that will hold values for tangents that are used to create normals, which are captured by the opaque render target. Tangents are used to calculate normals because, using tangent space to calculate normals gives normals similar variations to each other, which in turn will make the normals appear smoother when rendered. A tangent has to be calculated using three vectors coordinates, the face normal, the actual tangent to the surface of the face which is then orientated to the current texture co-ordinate and bitangent. The bitangent is perpendicular to the tangent that has just been calculated. With the tangent calculated, normals that are decoded from the shader’s normal map can now be calculated in tangent space. A normal from a normal map is not just simply the colour of a particular texture co-ordinate in a normal map. It is the normalized decoded from of that colour. A normal can be decoded from a colour by multiplying 2 by half of the colour value and normalizing the result.

After the normals have been calculated the alpha value from them are set to the green value of the specular map. This is to save space, because the alpha value of both depth values and normals aren’t used, so instead of having an extra renter target in the project, the specular power and intensity are stored this way for future use by the point light shader.

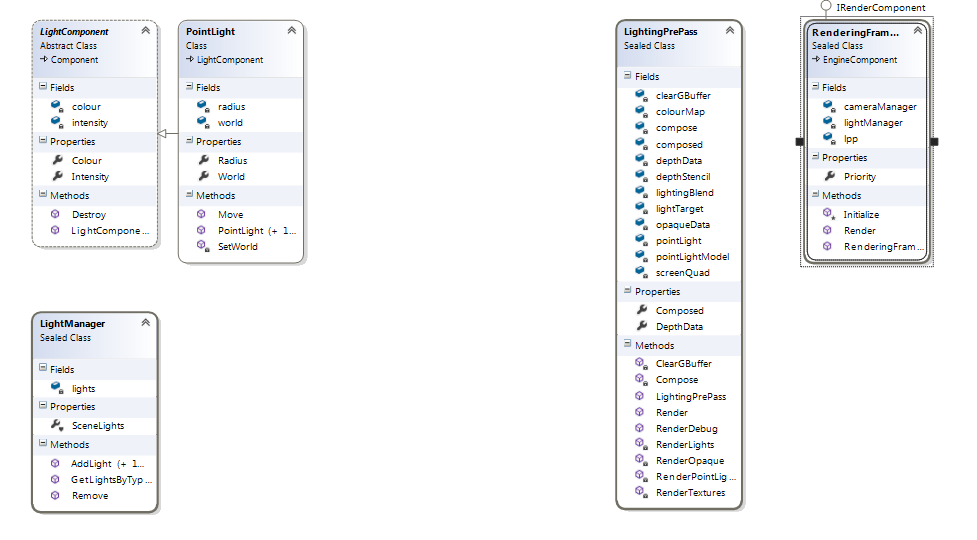
Finally, the final value for depth is calculated and its alpha value is set to the blue value of the specular map.

**Compose: -** This shader is used to combine the light map and the colour map, to form the final image. The colour map is found by rendering only the textures applied to all of the models.

Compose calculates the final composition by first multiplying the colour by the light colour plus the alpha value for the light colour and then returning this value as a colour.

**Lighting Pre Pass**

This section contains the actual code implementation of lighting pre pass in Aeon Engine. The classes required to implement lighting pre pass into Aeon Engine are shown below:

****

**Light Component**

This class is a component; I had wanted every point light to be a type of component so that it can be attached to a game object.

**PointLight**

This class defines the properties of a point light in Aeon Engine. It inherits from light component so that it can be attached to a game object. It being attached to a game object meant that it would move and be transformed by the game object.

**Lighting Manager**

This class is used to manage the point lights in the game, so that they can be quickly accessed by the lighting pre pass class.

**Render Framework**

This class is used to create and manage the various classes required to render a scene in Aeon Engine, such as lighting manager.

**Lighting Pre Pass**

This class is responsible for rendering the scene. In it there are a number of render targets. Those are:

* “colourMap”: this render target is used in the final stage of lighting pre pass in order to render the colour data of the models in the scene.
* “depthData”: this render target is used to have depth data from the scene saved on to it.
* “opaqueData”: this render target is used specifically to save opaque data retrieved from the scene.
* “final”: this render target is used to save the composed scene.

In Aeon Engine, lighting pre achieved through a number of steps those are defined below.

**Opaque Pass**

This is the most fundamental stage when attempting to achieve lighting pre pass in games.

In this stage the opaque data for the scene is rendered for use in the lighting pass.

Before the opaque data can be rendered the Gbuffer has to clear of any existing data, and default data is to be set on the colour, depth and opaque render targets, so that no irregularities occur when using the render targets at a later stage. An alter approach to this is to set the active render target of the graphics device to be the same as above, then set the clear colour of the graphics device to transparent followed by rendering a full screen quad. However this approach would set the data from all of the render targets to the same colour which will give the final result to have black sections on it if there was incorrect data taken from a different pass that involved the render targets at a later stage.

After the Gbuffer has been cleared the depth render target and the opaque render target is set on the graphics device and the opaque data only is rendered from each model.

**Gbuffer: -** This is a set of render targets whose function to capture data from the various passes involved with the lighting pre pass.

**Render Target: -** This is a feature of modern GPU’s that allows a 3D scene to be saved to a texture instead of to the back buffer of the graphics device. This feature allows a shader to manipulate the texture directly.

**Full Screen Quad: -** This is a set of two trangles which take up the full size of the screen. The primary reason to render a full screen quad is so that, a pixel shader is applied once per pixel of the render target.

**Lighting Pass**

In this pass the light volume for the entire scene is calculated. This is done by first setting the active render target for the graphics device to the light target. Next a spherical model has its vertex buffer attached to the vertex buffer of the graphics device. Then the parameters of the point light shader are set such as, the scene’s opaque data and the radius of the light. Directly after the shader has its current technique applied and when all lights have been rendered a full screen quad is rendered.

This is to build up a lighting volume which when the point light shader has been applied the light volume will appear as a set of lights affecting the scene.

**Composition**

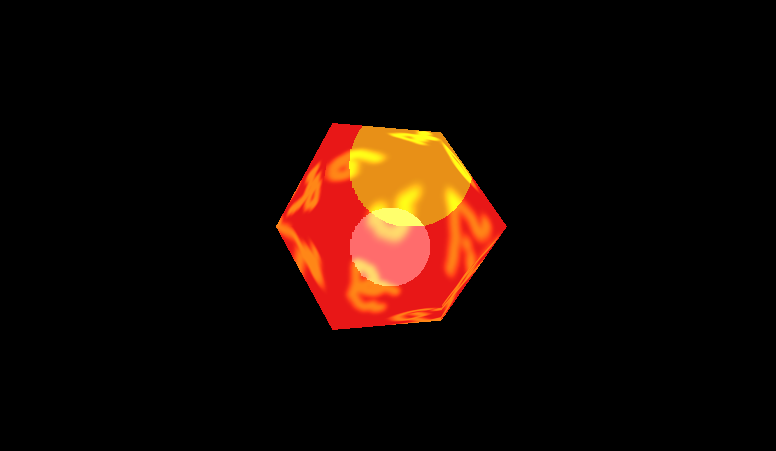
In every sample I’ve seen the final stage of the lighting pre pass is that the light map is sent back to every model and the model is then re-rendered this time with its texture and the applied lighting. In previous revisions of the project I’ve completed this stage only to find out that only one model can be rendered in a scene. I had an idea that it was this stage that caused the problem. Therefore I had decided to change this step instead; I had decided to render the colour data for every model in the scene to a texture such as I’ve been doing before. Next I decided to use a compose effect. This effect would take in both the lighting volume and the colour map and combine the two into a single final texture.

**Problems Encountered**

Throughout the implementation of lighting pre pass into Aeon Engine there was only one problem that I had which meant that I had to restart the implementation process over again.

**Problem: -** Depth render target is not being having any data set.

In both deferred rendering and lighting pre pass the depth buffer is essential for making the lighting appear three dimensional. Below is what happens if the depth target doses not have any data set:



**Solution: -** I couldn’t find a solution to the problem in that particular revision of the project. I had tried and tried but nothing seam to work. At the time the shaders that I was using for the project were edited versions of the shaders from Catalin Zima’s deferred renderer converted to XNA 4.0 by Roy Triesschijn. Instead what I had done was that, I restarted the project again. This time I researched more into shaders and how they worked. I focussed more on looking at how other people achieved multi-pass lighting as opposed to reading books about the subject. Eventually I decided to create my own shaders for the project. Some of these were less that helpful. After a while a decided that enough time had been wasted on trying to achieve 3D real time lighting and instead focused on 2D shaders. At this point my supervisor suggested that I should change from 3D lighting to 2D light, because I had already wasted too much time trying out shaders and not enough time implementing them. I was relieved to be able to change to 2D lighting instead because there is no need for a depth target, and I had somewhat more success with implementing lighting pre pass into a 2D game. Below is my implementation of lighting pre pass into a 2D environment:



The sample above is rendered with a single point light set at the bottom left hand corner of the screen.

I was considering staying with this change in the project. However over the coming weeks I had started a new side project, one which I had been thinking about for a long time. It was for a 2D game. However the more I planned the idea out the more I felt that the game would be more suited for a 3D environment. This is when I had decided to make a final attempt at implementing lighting pre pass into a 3D environment.

This time I had decided to change the overall design of my project drastically. The new processes I had implemented into Aeon Engine were:

* Instead of the light map created from the lights being passed back into each model for rendering. I decided to create a colour map by rendering the albedo data from the models.
* The introduction of a composition process. This takes place after the light map and colour map have been created. The shader combines the two images into one giving a final result.
* Models are now required to be built using a custom processor. The primary reason for this is to configure the default material applied to a model with a custom one. This is so that the textures being applied to the model are set to the rendered effect at run time, as opposed to having to specify what textures are allocated to what parameter in the shader. Also using as custom model processor allows me to check before hand to see if the model that is being processed has the appropriate vertex data for the shader such as; a tangent and texture co-ordinates.

Several changes had to be made to my shaders most of which were necessary for implementing the new processes. Such as the main shader used to render models is now only has to render its textures at the appropriate stages of in the lighting pre pass, instead of applying the light map to itself. The end result of all of the new changes can be seen below, with the final version of Aeon Engine that has the new processes and shaders implemented into it:



The top most small image indicates the opaque data taken from the render target assigned to capture it. The opaque data in this case is the model normals and specular power. The small image directly below depicts the depth data from the scene along with the specular intensity of the model. Both the specular power and specular intensity are stored as the alpha value of the mentioned render targets. The middle small image depicts the light map built up from the active lights in the scene. The final small image shows the colour map, which is the result of the rendering of all of the textures of the models in the scene. The big image, in the image above is the final composition of the scene.

**Content Pipeline Extension**

A content pipeline extension project is a project type available in XNA. It is use when the available content pipeline that XNA provides is unable to process certain content types required by the game or when a particular file type is required to be processed in a way that is unavailable to XNA.

**Content processor: -** A content processor is used to define how a file using it is going to be built at the project’s build time. Not all files added into an XNA game project need to have a content processor active.

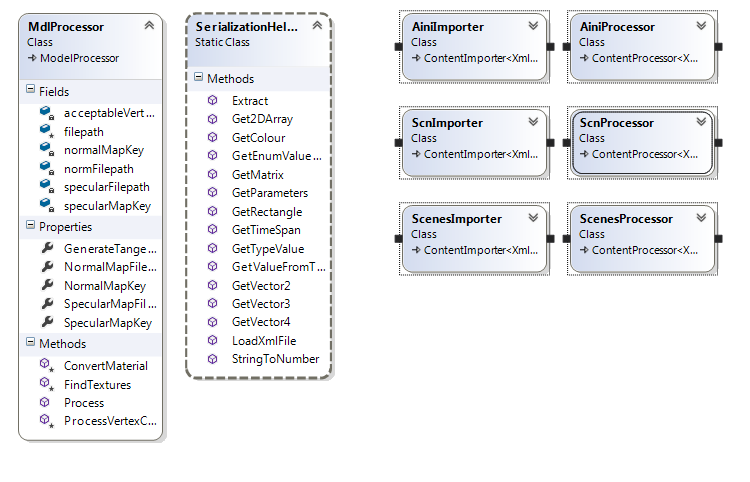
**Content importer: -** A content importer is used to define how a file using it, is imported into the project. All files that are added into an XNA game project must have a content importer selected in its properties.

**Implementation**

I needed to employ the use of a content pipeline extension project in order to configure the material data of a model that is going to be render by Aeon Engine so that, the applied effect on the model doesn’t need to be configured at run time. Instead by use of a custom content processor, the material data of a model is configured at build time thus increasing the performance of Aeon Engine. Also I needed to implement custom content processors and custom content importers, so that Aeon Engine could also:

* Create the required engine components at run time.
* Create and manage scenes, also at run time.

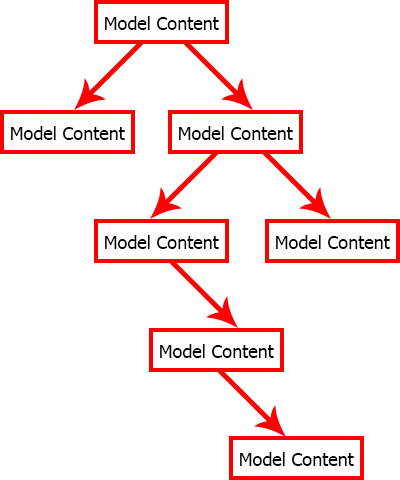
Below is my implementation of the content pipeline extension project (AeonPipeline) used by Aeon Engine.

****

The primary reason for implementing a content pipeline extension project into Aeon Engine is to configure the material data of a model. This is done in the content processor entitled “MdlProcessor”.

In the MdlProcessor, the very first thing that happens is the the file name for the model is stored; this is used later for discovering textures. Next the model content is searched for textures. This is done by first checking if the model has geometric data, if it does then the model’s opaque data is searched through to see if it contains a texture with the same name as the one specified in the MdlProcessor. If not the file name received at the start of the processing is modified, that the MdlProcessor can check to see if in the same file location as the model a texture exists with a specific name, if not a default texture is applied to the model. The purpose of this is that when the new shader is being applied to the model the discovered textured can be used to set the shaders texture parameters. The above process has to be done twice, once to find a normal map and again to find a specular map. Next the model content is searched again twice, this is to see if there is a texture already applied to the model with the same identification key as the one specified in the MdlProcessor. If so, that texture is removed and the found texture is applied with the specified key.

Finally for every other child of the model content that was being checked, goes through the same process. This is because in XNA models are structured in what is known as a tree structure. This means that every model being processed can have many child models; and so on the diagram below depicts how this works:



The final steps in configuring a material that is going to be applied to a model was to override the ProcessVertexChannel method and to override the ConvertMaterial method in the MdlProcessor. The ProcessVertexChannel method is used to check to see if the vertices that form the model have the correct data for the shader. Those being a texture coordinate a normal, a tangent and a binormal. The ConvertMaterial method is used to convert the existing material applied to a processed model with a specific material applied by the processor. In the MdlProcessor this is done by loading an effect material content from an external reference and then setting the texture parameters of that particular effect material content with the found textures from the FindTextures method and the colour map already applied to the model.

**Effect material content: -** This type of object is used to define a DirectX shader that is applied to a model being processed at build time.

**Model content: -** This type of object is used to represent vertex data retrieved from a model such as mesh data. However models defined in XNA are not defined in a tree structure.

**Problems Encountered**

The only problem I had when implementing AeonPipeline into Aeon Engine was that an exception was thrown saying that AeonPipeline was incompatible the game project I was using to test Aeon Engine.

The solution was straight forward. Simply put, I went into the properties window for AeonPipeline and changed the target framework in the application tab, from .Net framework 4.5 to .Net framework 4.

AeonPipeline is primarily used by Aeon Engine to solve problems that had occurred, especially those that were used to enhance the modularity of Aeon Engine. Those were:

* The creation of engine components at run time.
* Creating scenes in Aeon Engine. I know that my project doesn’t exactly need to have scene management in it but, I had decided that for the project presentation I would like it if I could show of a number of different scenes each one having a number of different lights and models. So that I could demonstrate what can be achieved with lighting pre pass.

**Problem: -** The creation of engine components at run time.

This particular problem occurred when I had revised what had been done with Aeon Engine up to end of semester one. From my findings I had decided to make Aeon engine more modular.

**Solution: -** The solution to this problem involved the creation of several new code files in both Aeon Engine and AeonPipeline these were: A class entitled “FrameworkCreation” in Aeon, a custom content importer entitled “AiniImporter” in AeonPipeline and a custom content processor entitled ”AiniProcessor” also in AeonPipeline . This solution was implemented in a number of steps:

1. I had to create a class entitled “FrameworkCreation” this would be used by AeonPipeline to define how the information from the processed file would be built like. In which it would contain two collections. One to hold assembly references, the second to hold information on the creation of engine components.
2. I had to create a custom content importer (“AiniImporter”). This is used at build time to import an XML file. I decided to create my own custom importer instead of using the XML importer XNA provides because, an XML file imported using the XML importer XNA provides has to have extra code in it so that XNA can tell what kind of content it is and what type of class in a defined project is used to represent it. Also the games content pipeline would now need a reference to the defined project so that it could process the file. However, with my own XML importer, none of these are required; instead all I have to do is make sure that the output folder for the game contains the .dll file for the referenced project.
3. The final step for enhancing the modularity of Aeon Engine is to create the content processor (“AiniProcessor”); this will get the imported file from the importer specified in the file’s properties and process it accordingly. Unfortunately this step had its own set of problems those were:
   1. Each key in a dictionary must be unique.
   2. How to get an object from a XML node.

The first of those problems was relatively easy to fix. I had to create my own type of collection that can hold an undetermined type for a key and a possibly different undetermined type for a value. What I did was I created a class that contained two list of undetermined type. It would get the types for these collections when the class (“AeonDictionary”) is initialized.

Again a problem was raised when a file was being processed by the content processor. There was no data in the created “AeonDictionary”. The reason for this was that I needed to make both collections inside of “AeonDictionary” public; this is so that when processing the file a compiled version of the file can be created.

When content is built in XNA, XNA creates a compiled version of the content that has the file extension .xnb. This is so that the content doesn’t need to build again. Using pre built content saves on time when the project is being built, but it also means that the content can’t be edited in any way.

The second problem is slightly more complicated. From the diagram above of AeonPipeline you will notice a class entitled “SerializationHelper”. The main purpose of this class is to help with the importing and processing of files. In it there is a method entitled “GetTypeValue”. This method is where I solved the problem. In short, “GetTypeValue” returns an object from a string but only a certain set of objects can be returned. The string in most cases is gotten from the inner text property of any XML node. The string must be written in a certain type, that being the string must have an underscore after the type of object required; this is then followed by the value of the required object. The following are examples of string values required by “GetTypeValue” and their outputs:

* System.String\_This is a test; this string returns a string with the value “This is a test”.
* Vector3\_0, 0, 0; this string returns a Vector3 which has a zero in each of its components.
* ADS2\_This, is, a, test; this string returns an “AeonDictionary” in which both collections are strings, it has two keys (“This”, “is”) and two values (“a”, “test”).

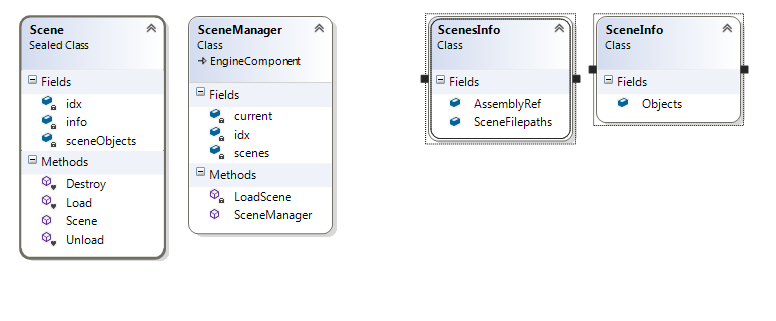
It works because the first part of the string contains the identification name for the required object and the second part of the string defines the value for the object. “GetTypeValue” first splits the string using the underscore as the separation point. Then “GetTypeValue” uses Type.GetType to find out if the first part of the string is a valid type name. If not the next thing that happens is a search through all of the common names of objects that I’ve already specified including: “Vector3” and “ADS2”. Either way if a type is found the second part of the string is then used and parsed in to set the value of the found object. If no type is found a null object is returned instead.

**Problem: -** Creating scenes in Aeon Engine.

In Aeon Engine a scene is a collection of objects created at runtime from an external file. I decided to have scenes in Aeon Engine after semester one because before this, a scene was a hard coded class, in which a number of point lights and objects were created. Testing the rendering of scenes using this method was a pain to say the least. With the success I had with being able to create engine components at runtime I decided to try and create scenes at runtime as well.

**Solution: -** The solution that I had devised for this particular problem would involve the implementation and creation of several new code files as shown below.

Included in Aeon



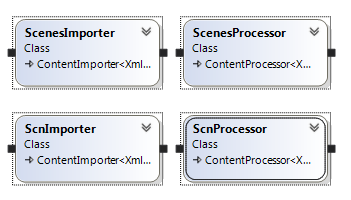
The scene class is used to store all of the objects required for rendering such as lights and models. A scene in managed and created by the scene manager class. The scene manager class creates all of the scenes defined in a specifically named external file.

The classes shown above entitled “ScenesInfo” and “SceneInfo” are used by AeonPipeline to define how the data from .scn and .scns files are to be processed. The ScenesInfo class is used by the scene manager class to find out which scenes need to be instantiated. Whereas, the SceneInfo class is used by the scene class to get its information from, such as model file paths. The ScenesInfo class holds a collection of assembly references and a collection of SceneInfo file paths. If the scene manager class successfully loads in the ScenesInfo file the first thing that happens is that the assembly references defined in ScenesInfo file are extracted and then added to the assembly manager class. Then the defined scenes are instantiated but are not initialized yet until it is needed. In which case if a previous scene has been initialized its content is unloaded and the new scene is initialized instead.

The assembly manager is used to manage assemblies referenced by a game using Aeon Engine.

It can also be used to create instances of any object in a referenced assembly.

Included in AeonPipeline

****

The above importers were implemented for the same reasons that the “AiniImporter” was implemented for, that being to reduce the amount of code per file and also to avoid the need to add references into the games content pipeline. The main problem here is the creation of objects outside of the serialization helper and as I’ve previously explained the serialization helper can only create instances of objects that are defined in it. This poses the question; how to create an instance of an object outside of AeonPipeline?

Initially this problem was difficult for me to understand and find a solution to. However, using what I know about assemblies and how Aeon Engine worked, I devised a solution.

What I had realized was that the majority of objects that were to be created using this solution would have to be created by known objects that can already be created by the serialization helper. From here I knew that I needed a new class that will hold a collection of objects and a string. The string representing the type of object that is not available to the serialization helper. This new class would then be passed into the assembly manager at run time to be instantiated. Provided that the given type name was valid. With this solution in place it now meant that I could have a vast number of objects of any type in a scene defined in an XML without having to change any of the code in the game engine. This solution was implemented into the “ScenesProcessor” and the “SceneProcessor” where I was having those problems.

**Summary**

What I’ve achieved with Aeon Engine is exactly what I had set out to do, which was to create a project that could demonstrate a technique that 3D games engines employ for rendering multiple models under multiple lights.

**References**

|  |  |
| --- | --- |
| Websites: |  |
|  | <http://gamedev.stackexchange.com/questions/12163/mvc-or-components-or-both> |
|  | <http://www.catalinzima.com/tutorials/crash-course-in-hlsl/> |
|  | <http://www.neatware.com/lbstudio/web/hlsl.html> |
|  | <http://jcoluna.wordpress.com/2011/01/18/xna-4-0-light-pre-pass/> |
|  | <http://gamedevcoder.wordpress.com/2011/04/11/light-pre-pass-vs-deferred-renderer-part-1/> |
|  | http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/ |
|  | http://gamedevcoder.wordpress.com/2011/04/11/light-pre-pass-vs-deferred-renderer-part-1/ |
|  | http://www.stack.nl/~dimitri/3dsview/phong.html |
|  | http://gamedevcoder.wordpress.com/2011/04/11/light-pre-pass-vs-deferred-renderer-part-1/ |
| Books: |  |
|  | Real Time Rendering Tricks and techniques in DirectX by Kelly Dempski. |
|  | Game Engine Gems, Volume One by Eric Lengyel. |
|  | Real-Time Shader programming by Ron Fosner . |
|  | Learn Vertex and Pixel Shader Programming with DirectX 9 by James C. Letterman. |
|  | Advanced Lighting and Materials with shaders by Kelly Dampski and Emmanuel Viale. |
| Online Samples: |  |
|  | <http://xbox.create.msdn.com/en-US/education/catalog/sample/shader_series_5> |
|  | <http://digitalerr0r.wordpress.com/2012/03/03/xna-4-0-shader-programming-4normal-mapping/> |
|  | <http://digitalerr0r.wordpress.com/2011/12/20/xna-4-0-shader-programming-3specular-light/> |
|  | <http://digitalerr0r.wordpress.com/2011/12/13/xna-4-0-shader-programming-2diffuse-light/> |
|  | <http://www.catalinzima.com/tutorials/deferred-rendering-in-xna/creating-the-g-buffer/> |
|  | <http://xbox.create.msdn.com/en-US/education/catalog/sample/normal_mapping> |
|  | <http://jcoluna.wordpress.com/downloads/> |
| Online Documents: |  |
|  | <http://www.idt.mdh.se/kurser/cd5130/msl/2006lp4/reports/drafts/mvc_and_graphics_engines.pdf> |
|  | <http://graphics.cs.cmu.edu/nsp/course/15-462/Fall04/slides/08-shading.pdf> |