Spring 5-9-2010

Development of a 3D Game Engine

Nicholas Alexander Woodfield
University of Connecticut - Storrs, Starnick10287@aol.com

Development of a 3D Game Engine

Nicholas Woodfield

Dr. Thomas J. Peters

Senior Design Project – Honors Thesis

26 April 2010

University of Connecticut

Contents

1. Introduction

This paper presents the development history and specification of a 3D game engine titled "Spark Engine". The term "graphics engine" is used to describe a complex software suite that provides a platform for scene management and rendering, allowing such functionality to be reused for multiple projects. A "game engine" expands on this concept and specifically provides tools and game modules to simplify game development. Spark Engine was developed to aid in the creation of a Windows based game for a senior design project.

The genesis for this project began with an interest in the development of shaders. Shaders are small programs that are executed on graphics hardware and allow developers to program the graphics processing unit (GPU) programmable rendering pipeline. Graphics hardware has moved from a fixed-function approach to a programmable rendering pipeline in recent years. This trend is due to the flexibility that shaders provide: a developer is able to create custom rendering effects by reprogramming the pipeline and is not limited to the algorithms provided by the rendering API. Originally this honors project was intended to explore and implement shader concepts, such as normal mapping and real time lighting.

Due to the interest in shader programming, the platform chosen for the design project was Microsoft's XNA Framework. The XNA Framework is a .NET platform for the C# language, and is built on top of Microsoft's DirectX rendering API. While gaining experience with the C# language was desirable, the primary reason for choosing XNA is that the platform has first class support for shaders. Every geometric model rendered by XNA is required to use a shader, which meant the backend support for using them was already available. This would allow more effort to be focused on the implementation of shader algorithms. All shaders developed for this project were written in Microsoft's High Level Shading Language (HLSL).

The idea for developing a game engine grew out of the needs for the design project, as XNA is not a game engine. It provides some tools for building content and access to DirectX, but it does not provide the means to either organize 3D scenes or efficiently render them. This meant the framework does not provide the type of functionality that a graphics engine would normally supply, such as a scene graph data structure or the functionality to cull geometry that need not be rendered. Early on in the design project, many features under development resembled the core

components of a graphics engine. This was the beginning of Spark Engine and prompted a shift in focus. While a significant portion was still to be devoted to shader development, the honors project expanded to include engine development.

This paper is intended to be a series of discussions for the major systems that comprise Spark Engine. These discussions range in describing how the system works, as well as how it expands on existing XNA features or provides a brand new feature. In some cases this will include deficiencies in the system's design or implementation as well as ideas for future development. While these discussions will be in detail, the nature of this paper is not to discuss implementation details. The paper is partitioned into six sections:

1. Motivation for creating the engine.
2. An overview of the XNA Framework.
3. Engine architecture discussion for all major and minor systems of the engine.
4. Performance and optimization work.
5. Future work for enhancing and expanding engine features.
6. Concluding remarks on success of meeting engine design and learning goals.

The source code for Spark Engine is open source under the New BSD License, and is freely available from a Google Code Subversion repository[1]. Engine systems and features discussed in this paper pertain to Version .6 of Spark Engine.

2. Motivation

There were a few reasons that motivated this honors project. The first being to gain experience working with aspects of computer graphics at all levels of the application, from the exciting to the mundane. This included the aspiration to investigate shaders, as well as explore how graphics engines are organized to aid the developer in efficiently rendering a scene. Experience cannot

[1] http://code.google.com/p/spark-engine/

easily be gained in those systems until they are worked with directly. No third party libraries would be used and most parts of the engine were designed and implemented from scratch. This would make the project an invaluable learning experience. Use of third party libraries would not facilitate learning how systems discussed in this paper function or how to implement them. The only exceptions to this rule were some significantly used XNA features, some of which were unavoidable such as using the XNA API to access DirectX.

When it was decided to expand the scope of this project to include engine development, the second motivating reason presented itself. If systems had to be developed for the senior design project that resembled a graphics engine, that effort should be unified under the direction of a coherent design philosophy. It made sense to a build a general piece of software which had systems that were designed to work with one another, opposed to software that was haphazardly pieced together to meet the needs of a specific application. Much of the work for the engine was to address the short comings of the XNA Framework, as many of its features are meant as starting points for hobbyists. Code that used the XNA API directly did not lend itself to be easily extended nor very elegant. This in turn led to the primary two goals in developing the engine: flexibility and reusability. Therefore there was tremendous motivation in improving the experience using XNA for developers, such as the senior design project team.

It also was desired for the work developed for this honors project to not just be thrown away at the project's conclusion, but to survive and be used by others. Therefore, Spark Engine was designed to be a viable and feature complete graphics platform that could be used for any graphics application, be it a game or not. This aspiration combined with implementing game-specific systems from the senior design project would propel the engine to the classification of game engine, since games were of primary interest in developing the engine. While this was very ambitious, at the conclusion of this project the majority of those aspirations did come to fruition. The engine has progressed very far in its development as the rest of this paper will discuss.

3. XNA Background

Since Spark Engine's purpose was to build upon the XNA Framework, this paper first presents several key features of XNA used by the engine. Microsoft's XNA Framework is freely available

software intended to allow easy entry into game development by hobbyists and independent developers. The framework supports the Windows, Zune, and Xbox 360 platforms. Development for the Xbox 360 is a major feature of the XNA Framework, as it opens that platform up for anyone to develop a game and sell it through the Xbox Live Marketplace.

The XNA Framework is not a game engine but rather a collection of tools and technologies to facilitate game development and foster cod reuse between different target platforms. Currently XNA only supports DirectX 9.0c, but is entirely shader driven. Everything that is rendered must be done so via a shader. This made it a perfect choice for investigating shaders, and forced Spark Engine to also be shader-driven. All modern game engines today are designed with shaders in mind, and building the engine in this style was desired.

At the time of this writing, Spark Engine is intended for use only on the Windows platform. Support for the Xbox 360 would require additional development, which was not needed for the senior design project. Although Xbox 360 support was a secondary reason for adopting the XNA Framework, as it was an area of interest.

Content Pipeline

The XNA Framework's content pipeline is one of the signature pieces of the entire framework. Game asset management was one of the reasons for the framework's creation, to bring together a set of easy to use tools to process texture and 3D models into a game ready format. This game ready format is the platform independent binary .xnb file, which allows for game content to be loaded quickly at run time. The content pipeline is entirely an offline process as the processing for content in most cases would be unfeasible during runtime and result in long load times.

This is an area of the XNA Framework that Spark Engine used significantly. The engine directly extended it to include its own model importers and processors, but changed little in the overall build process. Section 4.6 goes further into detail how the pipeline works as well as how the engine extends it.

Rendering Capabilities

The XNA Framework is built over DirectX 9.0c, which supports both shaders and the fixed-function pipeline. Later versions of DirectX have done away with the fixed-function pipeline in favor of a fully programmable pipeline with shaders (introduced with DirectX 10). This is one of the reasons why XNA is entirely shader driven, in order to facilitate a move to DirectX 10 sometime in the future. At the time of this paper, the next version of XNA (XNA 4.0) has begun this transition. The version of XNA used by Spark Engine is XNA 3.1.

Although rendering is entirely shader-based using the XNA API, it still supports some limited fixed-function effects. A notable example is fixed-function fogging, which is enabled with the use of the fog render state (See Section 4.2 for further details on render states). The engine fully uses the XNA API in its rendering system, as it provides access to DirectX 9.

Shader Effect API

In addition to the rendering capabilities the engine uses, the XNA Framework's shader `Effect` API provided the foundation for the engine's material system. The API builds directly upon DirectX's `Effect` class and allows shader programs to be compiled, bound to the graphics device, and shader constants to be updated. XNA provides a `BasicEffect` class that provides the default rendering effects for 3D models. However, using BasicEffect directly often resulted in ugly drawing code. While the `Effect` API was used by the engine, the material system (Section 4.3) was developed to address the shortcomings of working with the API directly and provide an automatic means with applying shaders to 3D models.

Game Component System

In order to promote game development, the XNA Framework provides a game component API. This allows for a portion of game logic (such as a scene graph) to be packaged in a component that could be reused by other developers. The system also supports the idea of game services, which are interfaces that are implemented by components. Game services are singleton-like objects that are

queried via their interfaces (e.g. a scene graph manager), which allows for different implementations to be replaced in a game without requiring code change. This is actually how Spark Engine is attached to an XNA application at start up; the engine can be considered a large game component. Section 4.7 has further details on application usage with the engine.

While this can be a useful piece of the XNA API, it often can be misused. For example, an animated character can be its own game component while a scene graph can be another component. This can cause issues in managing the order of how systems are updated or rendered. Therefore the game component API was seldom used in Spark Engine's development, instead favoring implementation of original systems that were unified under one design.

4. Engine Architecture

The engine is composed of several important systems that handle the tasks involved with updating and rendering a scene. This section discusses each of those systems in detail and how they interact with one another. The scene graph, rendering system, and material system form the core features of the engine and are explicitly coupled together. Other systems such as input and the animation library are extensions to the core. Reusability and extendibility were the two prevailing goals in designing the engine as a whole, and as a result most engine systems can be extended easily or replaced entirely.

4.1. Scene Graph

The engine's scene graph is its most defining characteristic and is the core component that the rest of the engine is built around. A scene graph is a collection of nodes in a tree, where a node has a single parent but multiple children and where leaf nodes represent geometry. This allows for the scene to be organized spatially, which leads to optimizations when rendering the scene. For example, objects not in view can be culled to reduce data sent to the graphics card. A scene graph also allows for inheritance between parent nodes and their children. If a scene object is rotated 45 degrees, and its parent rotated 30 degrees, then the object would be rotated by a total of 75

degrees. The rotation performed on the object is its inherent or local property; while the object's final rotation that its parent influences is its world property. Other properties can be inherited similarly, such as lights and render states. Thus the scene graph can be viewed in more than one context.

Since the XNA Framework does not provide a scene graph, a basic implementation was worked on in the first stage of the senior design project. This was to make development easier for the senior design project team to easily setup the scene and not have to worry about the mundane details involved with updating and rendering geometry. Therefore this feature gives the engine a tremendous advantage over the XNA Framework. It eliminates much of the repetitive code that often is found in XNA tutorials and leads to a high level of abstraction. The developer only has to worry about the scene graph's properties, such as its transformation or material rather than worrying about how to ensure those details are applied when the scene graph is updated and rendered.

The design of the scene graph is based on David Eberly's Wild Magic Engine from his book *3D Game Engine Design* as well as other similarly designed graphics engines such as the Java-based jMonkey Engine. In that design, the scene graph has a strong hierarchy where every scene object extends an abstract super class called `Spatial`. When discussing scene graph objects in a general fashion, the term spatial will be used. The `Spatial` class represents any object in 3D space, whether it is geometry or a logical entity, and has the following properties:

- Local and world transformations
- Render state list
- World bounding Volume
- Controllers
- Scene hints
- Parent node

The engine has two prominent scene objects that inherit from the `Spatial` class: the `Node` Class (internal node) and the `Mesh` class (leaf node). Although internal nodes do not contain geometry, they are required to have properties such as render state information to allow for geometry to inherit these states. For example, a render state applied to the root of the scene graph

is inherited by all nodes and meshes in the graph. Although `Node` is a subclass of `Spatial`, the parent of every `Spatial` is of type `Node`. Nodes ensure that all of their children are updated and drawn during those respective passes and provide functions for managing children.

`Mesh` objects on the other hand represent actual data – the geometry and material effects that will get rendered. Since they extend `Spatial` and not `Node`, they explicitly are not allowed to have children. A `Mesh` object has three important properties:

- A mesh data object
- A material
- A model bounding volume

While the `Mesh` class represents geometry, it in fact resembles that of a container class. The `MeshData` class holds the actual geometric data and owns the vertex and index buffers. This allows for multiple meshes to share the same geometric data, but at the same time be entirely separate entities with their own transformations, material s and controllers. The `MeshData` class supports all the primitive types that XNA does – point list, line list, triangle list, and triangle fan. Therefore a `Mesh` may be a composed of triangles, or simply a list of points. Regardless of the primitive type, the engine simply treats it as a batch of geometric data.

Since XNA is shader-driven, every `Mesh` object is required to have a material in order to be drawn by the graphics card. Materials contain reference to a shader effect object and a list of material definitions that feed data to the shader's uniform parameters. The material system is discussed further in detail in the Section 4.3.

In addition to the world bounding volume property in the `Spatial` class, the `Mesh` class has a bounding volume that is built from the geometric information from the `MeshData` object. This is called the model bound and encapsulates the entire geometry. A `Mesh` object's world bounding is a transformed model bound. Bounding volumes are propagated up the tree, thus the world bounding volumes for nodes are merged with the world bounding of their children. So in this way, the scene graph also represents a Bounding Volume Hierarchy (BVH) which forms the backbone for collision, picking, and culling.

A diagram of a scene graph is depicted in Figure 4-1. This scene graph has several notable features that were discussed previously. The moon is always attached to a planet, and will rotate about it regardless where the planet is positioned in the scene, since it inherits its transformation. The spaceship is free to move around the scene, since it's attached to the root node. While this example has the scene graph spatially organized, other scenes may group objects together based on common properties such as materials and textures.

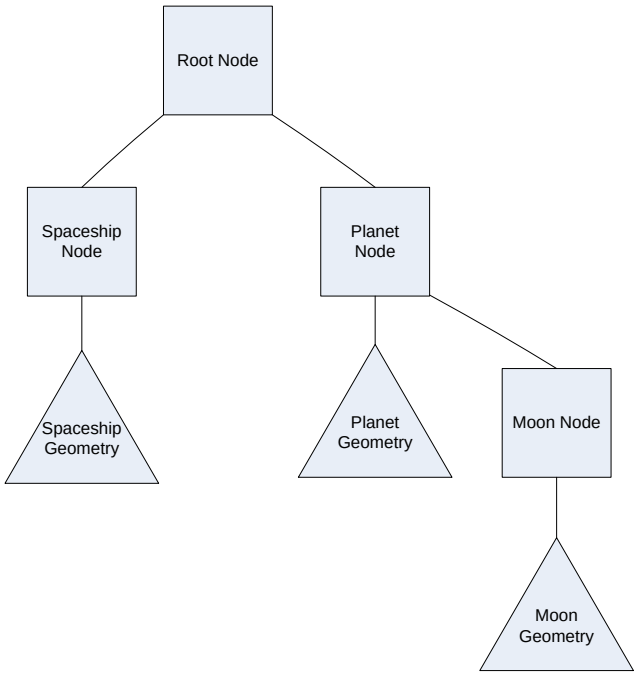

Figure 4-1 A diagram of a scene graph that is spatially organized. The moon orbits around the planet, no matter where the planet is positioned since it inherits the plane's transformation. Meanwhile, the spaceship is free to roam the scene as it is attached to the scene root. The root node typically is left at the scene's origin.

4.1.1. Properties

The scene graph is a complex data structure that has many properties and roles associated with it. Since the scene graph is a tree data structure, scene objects are hierarchical and inherit from the `Spatial` class. Therefore, `Spatial` contains many properties that are common to all scene objects regardless if they are logical nodes or geometry. This section discusses those common properties in detail.

Spatial Transformations

Every object in the scene has two sets of transformations (translation, rotation, scale), the first is local to just that object, and the second is derived from the transformations of its parent which is propagated from the root. During the update pass, once the world transformation is calculated a world matrix is computed and cached.

Controllers

Controllers are attached to a spatial, whether it is geometry or an internal node, and can modify it over time. They are local to the spatial only, and are updated when the spatial they are attached to is updated. This allows for logic to be attached to the spatial easily. A controller that rotates the spatial it is attached to about the Y-Axis 5 degrees per frame is a good example of a controller. The engine provides an interface to implement controllers, but also provides implementations such as the aforementioned rotation controller. Since controllers are the first thing to be updated during the update pass of the scene graph and as such, controllers can safely modify a spatial's transformations.

Usages for controllers can go beyond simple animations. Controllers for meshes can be used for morphing vertex data, or modifying render states. A controller could be used to vary the color of a light source or its intensity. Another example is a controller that implements a continuous-level-of-detail algorithm that dynamically changes the topology of a mesh.

Scene Hints

A `SceneHint` object is a collection of enumerations that provide clues to the engine on how to process a spatial. Scene hints can be explicitly set or inherited from the spatial's parent. Therefore there are hints that are local to the spatial and those that are derived from its ancestors. The hints are as followed:

- `CullHint` – Determines how the spatial is to be culled during rendering. It can do as it parent does, never be culled, always be culled, or be dynamic. The last hint is the default and is culled depending if the spatial is in view or not.
- `PickingHint` – Determines how the spatial should behave during collision and ray picking queries. A spatial can be queried for collision only, or ray picking only, or both.
- `RenderBucketType` – Determines to which bucket the spatial will be added during rendering. See Section 4.2 for more information on render buckets.
- `TransparencyHint` – Sets whether the spatial should use one sided or two sided rendering when it is in the transparent render bucket.
- `LightCombineHint` – Specifies how light states are to be combined. This enables many lights to be in the scene and the light states the spatial uses. Lights are also sorted based on how much they affect a spatial and on the maximum number of lights the material system supports.
- `TextureCombineHint` – Similar to `LightCombineHint`, but for how texture states to be combined.

Integrated Bounding Volume Hierarchy

As mentioned, each spatial in the scene graph has an associated world bounding volume and the graph forms a bounding volume hierarchy. The primary use for this hierarchy is for frustum culling during the drawing pass. Since a node's world bounding encompasses all of its children, if that bounding volume does not pass the test against the camera's frustum then the node and its entire branch can be thrown out without further checks. Consequently, each spatial remembers whether it was contained by the camera frustum, intersected, or was completely outside. So if a node is completely contained inside the frustum, no further checks are needed on the children since they are guaranteed to be completely contained also. This allows for a reduction in the number of frustum tests to be performed, as well as prevent unnecessary draw calls to the graphics card for geometry that will not be visible, allowing for more objects in the scene. A more detailed discussion of frustum culling and its impact on performance is presented in Section 5.

In addition to frustum culling, the bounding volume hierarchy forms the basis for collision determinism and 3D picking. The bounding volume hierarchy serves as a broad pass, where the scene graphs world bounding volumes are checked first for collision and picking queries. If there is no intersection with a node's world bounding, that entire branch can be effectively culled. A narrow pass then can be initiated, which could be at the triangle level. Another possibility is a second bounding volume pass, where the volumes are defined as collision volumes. Collision volumes are special use bounding volumes that an artist creates to approximate the mesh and are exported with the model. The engine does provide support for collision volumes; however it does not implement checking them for collisions or picking. They were used by the senior design project for non-engine classes, as they required extensions to the scene graph. Collision volumes are discussed further in detail in Section 4.6.

All bounding volumes support intersection and merge methods with one another, as well as methods for testing pick ray intersections and frustum plane intersections. The engine currently supports three types of bounding volumes, which inherit from an abstract super class:

- **Bounding sphere** – Simplest bounding volume has a radius and a center. It is one of the quickest to check.
- **Axis-aligned bounding box** – A box where its edges are aligned to the coordinate axes.
- **Oriented bounding box** – An arbitrary box that encapsulates the mesh.

Figure 4-2 shows these three bounding volume types as they appear in the engine. The white bounding volumes that surround the orange cube are drawn by the engine's debugger. The images provide a good visual for examining the strengths and weaknesses of the bounding types. Spheres are the fastest to check in collision and intersection tests, but they usually do not approximate an arbitrary mesh that well. This is because of false positives – an object may intersect the bounding volume, but not the mesh that it encapsulates. The Axis-aligned bounding box is not as simple as the sphere, but it is a widely used volume. However, it can have drawbacks when the object that it encapsulates is rotated. This requires the volume to be computed again, and since it is aligned to the coordinate axes, it does not rotate with the object. When the cube in Figure 3-2 is rotating, the axis-aligned bounding box does not contain it very well.

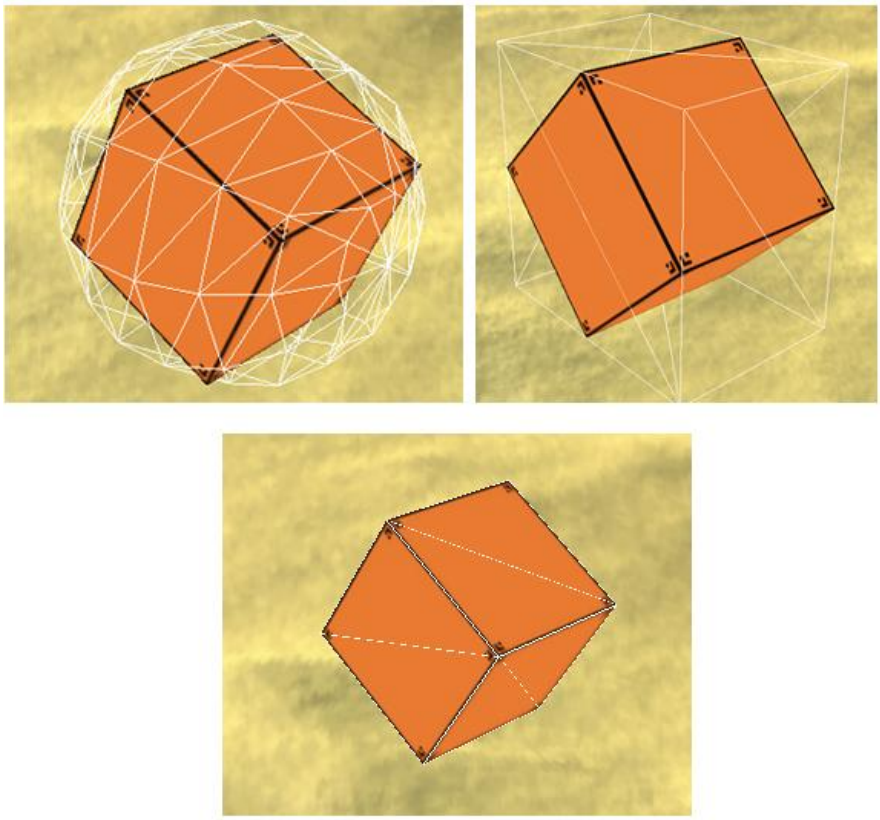

Figure 4-2 Upper left: Bounding sphere. Upper right: Axis-aligned bounding box. Lower middle: Oriented bounding box.

A special version of the box volume is an oriented bounding box. It's essentially an axis-aligned bounding box that is aligned to an arbitrary axis. This allows the box to generated, and then rotated with the object it encapsulates. This often produces a much tighter fit, but oriented bounding boxes require expensive computations in collision and intersection tests.

Collision and Picking Testing

Currently the engine provides utility methods for identifying if a collision has occurred between two spatials, and for testing a ray against the scene. As described in the previous section, these are broad pass queries using the world bounding volumes of scene objects. The engine also supports triangle accuracy ray intersection testing for individual meshes. Collision and picking testing is handled separately from the scene graph data structure, in the static `PickingUtil` class. The

scene graph root or parts of the scene graph are passed to the utility class for processing. This allows for custom implementations to easily be developed to replace the engine's default algorithms.

In addition to bounding volumes, there is support for artist created collision volumes, although they are not included in the queries by default. These utilize the same bounding volume types as the bounding volume hierarchy, but are used to approximate geometry. The engine only supports the importing and processing of collision volumes, as the senior design project utilized them. Although completely incorporating them into the collision and picking queries is a future goal. See Section 4.6 for further details on importing collision volumes.

Dirty Marks

Before a frame is rendered, the scene graph undergoes an update pass that ensures that all controllers, transformations, render states, and bounding volumes are updated. However, in certain cases such as static geometry, time can be wasted updating this data when it has not changed since the last update call. The engine automatically keeps track, for each spatial, what data has been changed with the use of a "dirty mark" bit flags. When certain spatial properties are changed directly, such as a change in the local transformation, render states, or bounding volumes the respective property is flagged. These flags are then propagated up and down the scene graph in order to notify ancestors and descendents that their world properties will require an update. Then during the update pass, if a property is marked dirty either directly or indirectly, the proper calculations are done and the flag is reset. If no flag is present, then those calculations are skipped.

The use of dirty marks is the key to reducing redundant computations and speeding up the time to update a scene. This is an improvement over Eberly's Wild Magic Engine, as it automatically locks parts of a static scene from updating. In Wild Magic, parts of the scene graph that were static would have to be locked by the developer manually. Dirty marks are more flexible and reduce errors if the developer forgets to lock or unlock parts of the scene graph.

4.1.2. Extensions

The scene graph is intended to be extended to allow for new types of scene objects. Such extensions are skyboxes and specialty nodes for lights or cameras. Some of these extensions are practical and are provided by the engine already. More ambitious examples of extensions can be for scene graph management. The engine does not have any spatial partitioning or scene graph management built in, such as an oct-tree system. However, the scene graph is by design malleable enough to be extended to be used in such systems. How the scene graph is organized is entirely up to the user and the scene objects are not aware nor specify any organizational requirements other than parent-child relationships.

A useful extension provided by the engine is the shapes library. Although the `Mesh` class is general enough to represent any geometry whether they are points, lines, or triangles, shapes provide convenience access to common primitives. Currently the shape library provides the following primitives:

- Boxes
- Lines
- Quads
- Spheres
- Utah Teapot

Of particular interest are boxes, lines, and spheres as they used actively by the engine in a debugger tool. The engine's debugger visually draws world bounding volumes and vertex normals. Throughout this paper, many of the engine screen captures have objects from this shape library.

4.1.3. The Update Pass

Before the scene is drawn, the scene graph is traversed and updated. This is initiated by calling the `Update()` method on the root of the scene graph. A GameTime XNA object is passed to each spatial to allow for determining the time between frames for interpolation purposes. The following summarizes the entire update process:

1. Update controllers
2. Update world transform (If flagged)
    a. Combine local transform with parent if any
    b. Cache world matrix
3. Call `Update()` for children (if `Node`)
4. Update world bounding (If flagged)
    a. Merge children world bounds (or transform model bound if `Mesh`)
    b. Propagate world bound to root (updates parent's world bound)

4.1.4. The Draw Pass

After the scene graph is updated, the drawing pass begins. Like that of the update pass, the drawing pass is initiated by calling the `OnDraw()` method for the root of the scene graph. During the draw pass, each spatial is passed a reference to the renderer object that will handle the rendering of the scene. The scene renderer and render queues are discussed in further detail in the next section and frustum culling is discussed in Section 5. The following summarizes the draw pass and the associated method calls:

1. `OnDraw()`
    a. Frustum cull check
    b. If inside or intersects, call `Draw()`
2. `Draw()`
    a. If `Node`, call `OnDraw()` for children
    b. If `Mesh`, add to render queue to be processed
        i. If set to skip, call `Render()` immediately
3. `Render()` (`Mesh` only)
    a. Apply render states (checked against renderer's enforced states)
    b. Apply material
        i. Update material definitions (sets shader constants)
        ii. Evoke device draw call

4.2. Rendering System

The engine's graphics system makes up a significant portion of the engine as it provides access to the XNA `GraphicsDevice` object, which in turn handles the low level DirectX API device calls. Two major components make up the engine graphics API – the scene renderer and render states.

Scene Renderer

The `SceneRenderer` class is the rendering vehicle for the engine. It is the object that is passed to each spatial during the draw pass and provides the functionality in actually rendering an object to the screen. The scene renderer was designed to be completely stand-alone. Each renderer owns a camera, and therefore a viewport – which can be the entire screen, or only a small portion of it. This allows for multiple renderers to be active at once in different contexts – e.g. a split screen game with two players with their own cameras that render the same scene graph. The scene graph data is entirely independent of the renderer.

Aside from containing the functions to interact with the graphics device and initiate draw calls; the renderer provides some additional functionality to increase rendering performance. The renderer accomplishes this in two ways:

1. **Render state caching** – The renderer keeps track of the last render state applied to the graphics device to reduce state switching. Switching states can be expensive, so reducing redundancies is a useful tactic the renderer employs.

2. **Render queue** – The renderer owns a render queue that manages a collection of buckets. This allows the developer to control the order of which objects are rendered. The engine supports by default four buckets: Pre-Bucket, Opaque, Transparent, and Post-Bucket which are rendered in that order. Control over the order of rendering geometry is a useful, as it allows for transparent objects to be drawn properly. Figure 4-3 demonstrates this, with a transparent and opaque cube. The picture on the left has the cubes ordered properly; whereas the picture on the right does not.

These two features allow the renderer to be intelligent in how it processes the scene graph during the draw pass. Aside from potential transparency issues, it can be inefficient drawing the scene graph in the order that it was organized. The scene graph may be optimally organized, but that is entirely up to how the developer creates their scene hierarchy as the engine does not provide that guarantee. Once the scene graph has been processed, the render queue then sorts all of the objects in each of its buckets accordingly. Then the contents of each bucket are rendered in the order mentioned previously.

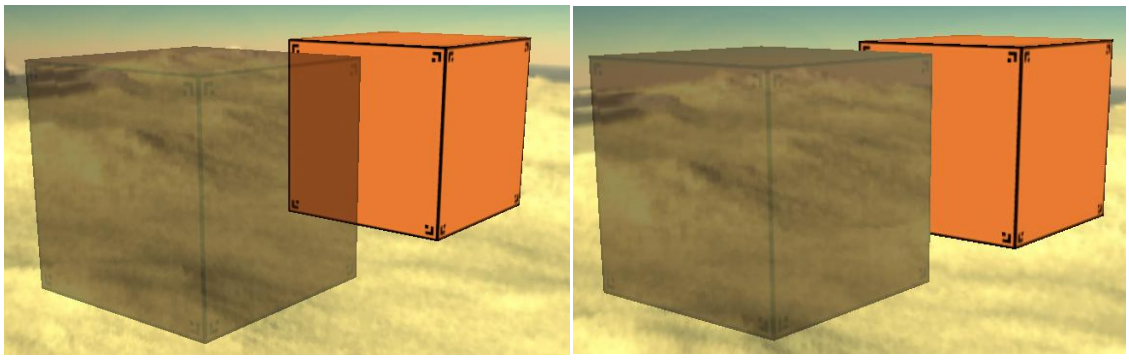

Figure 4-3 Left: Transparent object is correctly rendered in scene. Right: Same scene incorrectly rendered.

To illustrate why this is an efficient means of rendering, we will use an example scene from an engine test that had a thousand scene objects. Half share texture “a”, where as the other half share texture “b”. These objects can be arranged in any number of ways – the worst arrangement would be each object is rendered alternating between the two textures. The best arrangement would be to render the first 500 objects that share texture “a” and then the second 500 objects that share texture “b”.

However, without state caching this still can be slow since the texture for each object would be applied when it is rendered – even if it’s a redundant state change. With state caching, redundant states are prevented from being set, therefore the texture state is only set once for the 500 objects. Sorting the objects and employing state caching reduces the number of state switches from a thousand, to just two. This is perhaps one of the most significant advantages of using the engine over pure XNA code, as XNA does not have built in functionality for processing geometry for efficient rendering.

Since the engine provides interfaces for the render queue functionality, the type of sorting can be easily reconfigured by developers for their own needs. For example, sorting can be expanded to include sort by shader and material. All the buckets in the default implementation, save for the transparent bucket, sort their meshes by their texture state. If texture states are not present on meshes, then they are simply sorted front to back. The transparent bucket sorts meshes back to front, regardless of common states.

Render States

Render states affect the behavior of how geometry is processed as vertices and pixels flow through the rendering pipeline. In XNA, states are defined as enumerations which the engine uses. Unlike XNA, the engine groups these enumerations into individual classes that can be attached directly to the `Spatial` class. As mentioned in Section 4.1, the scene graph allows render states to be inherited and combined. The engine also takes a slightly different approach in the defining render states. Engine render states refer to all the information that is associated with geometric data, including material color information, textures, and lighting. This splits the classification of render states into two main categories: global states and data states.

Global render states contain the enumerations from XNA and are analogous to the typical definition of a render state – such as alpha blending, triangle culling, depth buffering, and so forth. These states are defined as global since their information is independent of any property of the `Spatial` class (Eberly 2007). Therefore these states are essentially nothing more than wrappers that provide a componentized form of interfacing with the render state enumerations in XNA.

The data render states are special use classes that provide the coloring, texture, and lighting information for the engine's material system. Separating data from the materials and shaders allows for flexibility and reuse – the textures and light objects held by data render states do not correspond to any one shader. This design is a departure from traditional XNA examples such as the `BasicEffect` class that relies on a single shader and has a one hard coded texture property. Meanwhile, the engine's `TextureState` can hold any number of textures that can be interpreted by different materials. For example, some shaders may only use the first texture as a diffuse color, others may use the second and third for normal and specular maps. The engine has a terrain texture

splatting shader that uses four textures and an alpha map for instance. How these data render states interact with the material system is discussed in detail in Section 4.3.

All engine render states inherit from an abstract class called `AbstractRenderState`. The following are the render states that the engine currently supports:

- `BlendState` - Controls alpha and color blending for transparency options.
- `CullState` – Controls triangle culling, e.g. cull counter-clockwise, clockwise, or none.
- `FillState` – Controls how the object should be filled, whether wireframe, solid, or point.
- `FogState` – Controls fixed-function DirectX9.0c fog functions.
- `ZBufferState` – Controls depth buffering.
- `MaterialState` – Data render state for controlling object's uniform material colors, such as diffuse, ambient reflectance, emissive, and specular reflectance colors.
- `TextureState` – Data render state for textures applied to an object. Also manages sampler states for texture filtering and wrap modes.
- `LightState` – Data render state for light objects, keeps track of a list of lights that are attached to the spatial.

Future Considerations

Although planned, the engine graphics API was to include a render pass system. This would allow for post-processing effects such as bloom, or multi-pass effects such as shadow mapping. In lieu of shadow mapping, the engine does support light mapping for static geometry. Such a pass system would allow those effects to easily be chained together without having to modify data in the scene graph. The renderer already provides some support for this, such as enforcing render states that override the render states of scene objects that are about to be rendered. This area is a candidate for future work, as all modern engines provide some sort of support for post-processing and shadows.

4.3. Material System

A material defines the properties of how an object should be colored and shaded. Every object that is rendered is required to have an associated material. This is directly tied to XNA's `Effect` API, since every object rendered in XNA must be done so via a shader. The material system is a simple but very flexible system that was designed to address several short comings of using the `Effect` API directly. Typically usage of an Effect in a draw call is as follows:

1. Load the shader effect file
2. Set shader constants
3. Call `Effect.Begin()`
4. For each `EffectPass` do
    a. Call `EffectPass.Begin()`
    b. Draw geometry
    c. Call `EffectPass.End()`
5. Call `Effect.End()`

This is the process for drawing geometry in the engine, as well as for any XNA application. The main problem with this process is setting shader constants – identifying the effect parameters and binding data to the effect are very specific to what the shader expects, and having this code directly inside the `Mesh` object's rendering call is unacceptable. Typically, for specific shaders, a subclass effect would be created. An example of this is XNA's `BasicEffect` class, which supports all the functionality offered by the basic effect shader. Many XNA applications and libraries have a similar setup for their custom shaders, but this lacks generality since the object using the shader needs to know what constants to set. If the engine implemented effect classes for the entire shader library like `BasicEffect` then engine classes would have to be re-written and extended for custom shaders. Therefore, the two defining goals for the material system were:

1. Allow generality when dealing with shaders and the objects they are attached to
2. Encourage reuse of engine objects with custom shaders

In addition to preventing the need to rewrite engine classes, this system was designed to eliminate having to hardcode shader parameters in effect classes and to adapt to changing data. The

desire to build a data driven material system grew from frustration with XNA libraries that often made it difficult using custom shaders. Since the creation of a shader library was a significant portion of this honors project, getting the material system's design right was crucial to the overall success of this honor's project.

The eventual design ultimately was simple and straight forward and consisted of three components that comprise a material:

1. HLSL effect shader (*.fx* file)
2. Material definitions
3. Data render states

As discussed in the previous section, the `MaterialState`, `TextureState`, and `LightState` are defined as the data render states. These states are independent from the shader, which allows these objects to be reusable. Shaders may interpret these states differently from other shaders. For example, a terrain texture splatting shader requires different textures than those found in a normal map shader.

The `MaterialDefinition` object acts as the glue between the data states and the shader's effect parameters. Therefore `Material` class is nothing more than a container that consists of the shader effect and a collection of material definitions. This is a significant departure from the design of XNA's `BasicEffect` class that has properties for setting the color, lighting, and texture parameters of the shader hard coded. A `MaterialDefinition` object does two things: scans the effect for specific effect parameters and sets shader constants. Normally the constants in the shader are identified via an associated semantic (such as the semantic "WORLD" for the world matrix or "DIFFUSE" for the diffuse texture), and once found are cached.

The `Material` class is responsible for updating definitions when the shader changes as well as providing access to the `Mesh` that owns the `Material` during rendering. This allows for the definitions to access the render state information of a `Mesh`, and bind that data to the shader constants before drawing. Since the definitions provide the functionality for setting shader constants, a `Material` object can shrink or expand to include new definitions. The `Mesh` and

`Material` objects are entirely independent, which allows for a wide range of materials to be applied to geometry without requiring any change in the geometric data.

When a custom shader is written, a new subclass of `MaterialDefinition` is typically the only new class that is required for the new material to be applied to an object. During the development of the shader library, this proved to be a very efficient and streamlined system. For example, when normal mapping was introduced to the engine, only a definition for the normal map was required. The definitions for material colors, world information, and lighting were unaffected and reused by the normal map shader. The only pitfall involved in the system is that consistency is needed – the definitions provided by the engine identify constants via semantics. A good example of this is if a custom shader changes the lighting model, but wishes to reuse the lighting system, then the shader must use the semantics that the engine is scanning for. The same is true for world properties such as the camera (eye) position as well as the world, view, and projection matrices. These are fed to shaders via a world definition, which can be extended or replaced entirely depending on the needs of the developer (e.g. in addition to the world-view-projection matrix, the inverse matrix may be needed).

While the material system allows for the `Material` class to be used directly with custom material definitions, often it is convenient to extend the class. The engine provides subclasses that correspond to the *BasicMaterial.fx*, *NormalMapMaterial.fx*, and *RimLightMaterial.fx* shaders. The first is a shader that has permutations for basic texturing, vertex coloring, material colors, and lighting. For convenience, the `BasicMaterial` class automatically adapts to the vertex declaration and render states of a `Mesh` object. If the `Mesh` does not have normals, then lighting is disabled or if it has vertex colors, then a vertex color enabled shader is selected. If it has a `TextureState`, then a diffuse texture enabled shader is selected. These checks are performed when the `Material` is updated before the draw call. The `BasicMaterial` class also supports the functionality to disable texturing, coloring, or lighting. Every `Mesh` by default has a `BasicMaterial` attached if no other `Material` instance is defined. The subclasses for the normal mapping and rim lighted materials provide properties for disabling textures, or for changing parameters such as rim color.

Future Considerations

The design of the material system was not just intended for ease of use in creating custom shaders, but in providing the means for material scripting in the future. This is very similar to that of OGRE 3D's material scripts. The ability to define such attributes of objects, through the use of scripting, is an important feature of a game engine. This allows for artists – with minimal programming experience – to create effects with ease, as well as reduces the amount of hard coding in a game. For example, new types of materials could be defined in a script and created by the engine programmatically – eliminating the need to create new definition classes or any C# code. These material scripts could then be applied to any model and be reused between applications. While the engine does not presently offer this feature, the material system was designed with scripting in mind and its flexibility can certainly support the extension.

4.3.1. Lighting

The engine's material system was largely conceived during the construction of the lighting system. All lights in the engine inherit from the `Light` class that contains common properties such as color and attenuation properties. Every light has ambient, specular, and diffuse color terms. The three lights in the system are as follows:

- `PointLight` – Omni directional light with a world position that attenuates. (Figure 4-4)
- `SpotLight` – A specialized point light that has a direction and angles that restrict the light into a cone. (Figure 4-5)
- `DirectionalLight` – A light that has a direction and is taken to be infinitely away from the origin. This models light sources such as the sun, and since it does not have a position, it does not attenuate. (Figure 4-6)

These light objects need to be attached to a `LightState` in order to be used by shaders. The engine's shader library supports lighting library functions that utilize a lighting model similar to the default built in Blinn-Phong lighting found in OpenGL and DirectX fixed-function lighting. The engine currently only supports per-pixel lighting (up to 8 lights per object) and requires Shader

Model 3.0. The maximum lights per object are determined by the light state, for combination purposes, and is adjustable. Future incarnations of the engine may have a Shader Model 2.0 compliant version of the engine's built-in lighted materials.

The lighting system was created to address many shortcomings with the basic lighting provided in the XNA Framework. XNA provides a single light class – a directional light, and allows up to 3 of these lights in its basic effect shader. It cannot be extended, and to add new lights such as point and spot lights requires a developer to write their own lighting system. Since design goals for the engine were to promote flexibility and reusability, it would have been counterintuitive to follow XNA's design. It was undesirable to hard code the type of light required by the shader, for instance. This meant that the light struct in the HLSL code had to be general enough, that any light – whether it is a point, spot, or directional could be used with the shader. An object can have a directional light and several point and spot lights affecting it at once, or any other combination. The engine provides a shader library file that contains the light struct, and lighting functions, which allows them to be used by any shader.

```
//Light struct that represents a Point, Spot, or Directional
//light. Point lights are with a 180 degree inner/outer angle,
//and directional lights have their position's w = 1.0.
struct Light {
            //Color properties
            float3 Ambient;
            float3 Diffuse;
            float3 Specular;

            //Attenuation properties
            bool Attenuate;
            float Constant;
            float Linear;
            float Quadratic;

            //Positional (in world space) properties
            float4 Position; //Note: w = 1 means direction is used
            float3 Direction;
            float InnerAngle;
            float OuterAngle;
};
```

Above is a snippet of HLSL code for the light struct. It contains all the properties of each type of light; even if that light doesn't use them (e.g. point lights do not have a direction). This is how the lighting functions identify and process the light data. For instance, directional lights do not have

positions, therefore the W component in the `float4 Position` is set to 1. This allows us to determine if the `Position` or `Direction` properties should be used in the lighting computation. This of course generates some slight overhead, as some properties need to be checked to identify the type of light that is being used.

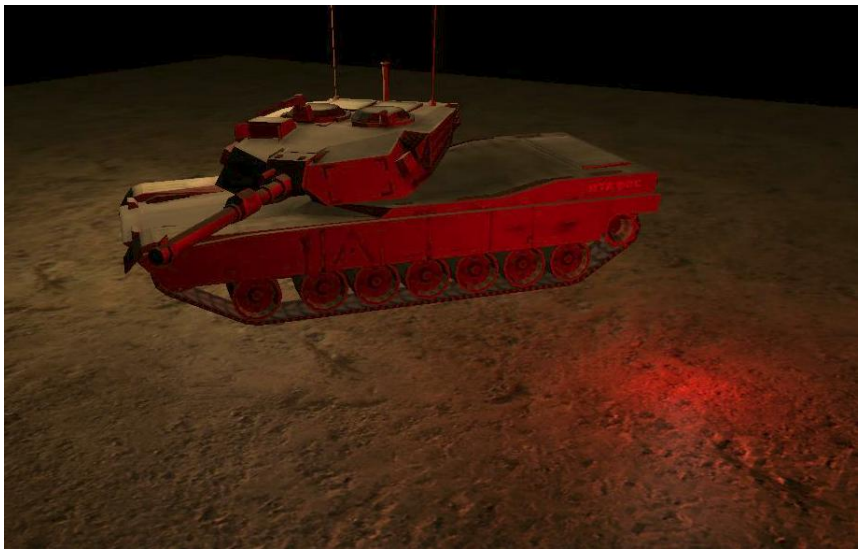

Figure 4-4 A scene with white and red point lights.

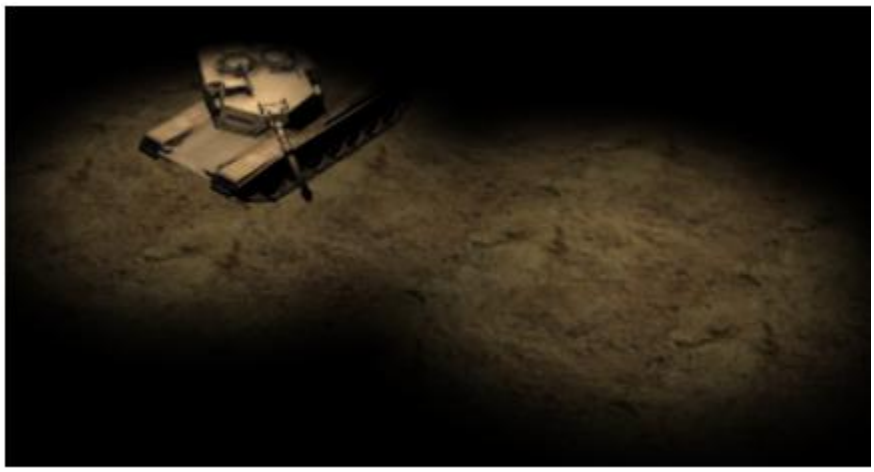

Figure 4-5 A scene with two white spot lights.

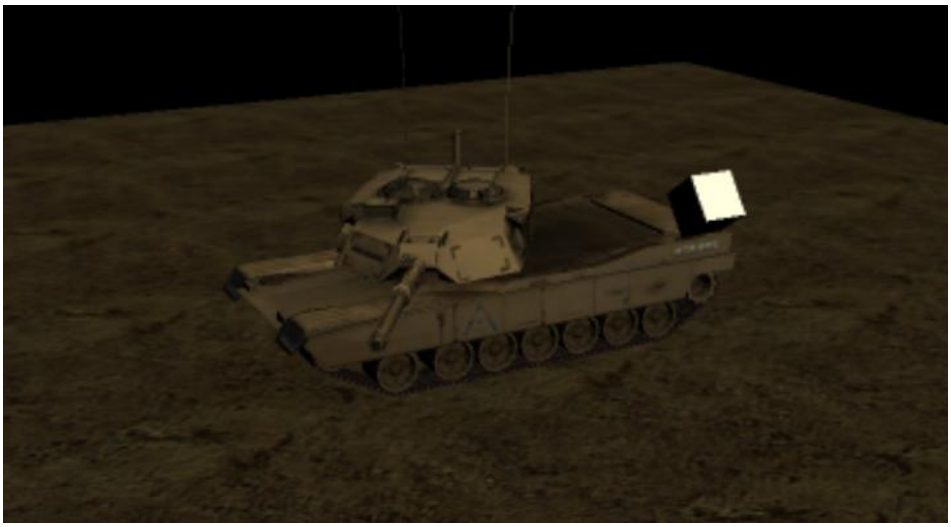

Figure 4-6 A scene with a directional light. The cube faces in the direction of the light rays.

4.3.2. Shader Library

In addition to the lighting library functions, the engine hosts a shader library of basic and advanced effects. The section details each shader that the engine provides as well as a discussion on their advantages and disadvantages. Engine development for this area was significant because it harks back to the original honors project concept for researching and implementing shader algorithms. Therefore every shader in this library was researched and written from scratch to support the rendering capabilities of the engine to meet those original project goals.

Quite a bit of experience was gained from the work done in implementing this library. For instance, while shaders have gained much popularity over fixed-function rendering due to their flexibility, they can be problematic when a large number of settings are supported. This can lead to an exponential explosion of shader permutations. Shader permutations became a limiting factor in the library's development due to time constraints; not every shader effect has a wide number of permutations, save for the basic material effect. This is also because the library is only intended to be a starting point for other developers and for general use, to be used for many different applications. This is a reasonable trade off, since the material system was designed to be easily extended according to a developer's requirements.

A different approach from shader permutations is to create “uber” shaders, which are very large and contain all the possible settings and effects for a complex material. While uber shaders are used in many games, but they can cause problems with older hardware due to instruction count limitations or frequent use of conditional statements to support the number of settings. The engine also cannot possibly predict all the needs of every developer that works with it either, which is another reason to favor creating shader permutations for general use.

As mentioned due to time constraints, not every shader written for the library has all possible permutations covered. For example, not every shader supports vertex coloring. Of all the engine shaders, there are approximately 30 possible permutations with 17 implemented. This is only considering several options for each shader (vertex coloring, material colors, texturing, and lighting). Some of these options are common between most of the shaders, but not all (e.g. normal mapping always requires lighting). This also does not account for other options. If the engine were to have its own implementation of fog as an option, this would add an additional 17 permutations. Fog in the engine is a fixed-function controlled by `FogState` and not a shader solution. It was decided to implement the permutations that would be used the most for each shader and then move on to other engine details that required more immediate attention.

Some shaders are also only a basic implementation and do not fully support what may be considered standard features – such as normal mapping on terrain and skinned materials. A shader library can quickly grow to include hundreds of individual shaders, which often have dedicated programmers in their creation and maintenance – a luxury that could not be afforded since the entire engine had only a single dedicated programmer.

From this firsthand experience working with shaders and observing the drawbacks of the library, it would be pertinent to describe how the library could be made better in the future. Ideally, this would be to have the shaders’ functionality split into global library functions that can be used by any shader such as how lighting is done. This would lead to the easy creation of new shaders that simply chain these effects together, effectively reducing the amount of hard coded shader permutations and eliminating the need for uber shaders. Shaders could be programmatically generated during runtime, compiled, and cached based on the needs of the geometry (e.g. if vertex coloring is needed, a new permutation is created without a programmer having to rewrite shader code). This can make the existing engine shaders more efficient and support older harder. For

instance, the engine's lighting uses up to 8 lights and will always loop 8 times. A model that uses only 2 lights would have a shader generated for this case thereby eliminating the instructions for 6 unused lights. While such a system would be extremely powerful, it also can be extremely difficult to build, although this could be a worthy area for future development.

Basic Material

If the user does not specify a `Mesh` object's material, the engine's basic material shader is applied using the `BasicMaterial` class. The basic material shader consists of a range of shader permutations that form the engine's basic rendering capability. This was the first set of shaders completed for the engine and closely resembles that of XNA's basic effect shader. There are four options for the material:

- Material coloring (ambient, diffuse, emissive, specular)
- Diffuse texturing
- Vertex coloring
- Per-Pixel lighting

This results in 8 different shader permutations for these options. An example of one of these permutations is geometry that has vertex coloring, a texture, and is lit. For every permutation that is affected by lighting, there is a sibling permutation that is unlit. It should be noted that the basic material is designed to closely resemble the functionality of a fixed-function graphics pipeline. This is the reason why all possible shader permutations are implemented for this shader.

Figure 4-7 is representative of several permutations of the basic material shader. The shader mimics the fixed-function pipeline by offering lighting, texturing, and vertex coloring for a `Mesh`. While these effects may not render geometry in the most exciting manner, they are absolutely necessary to have in the shader library as they form the backbone of the engine's rendering capabilities.

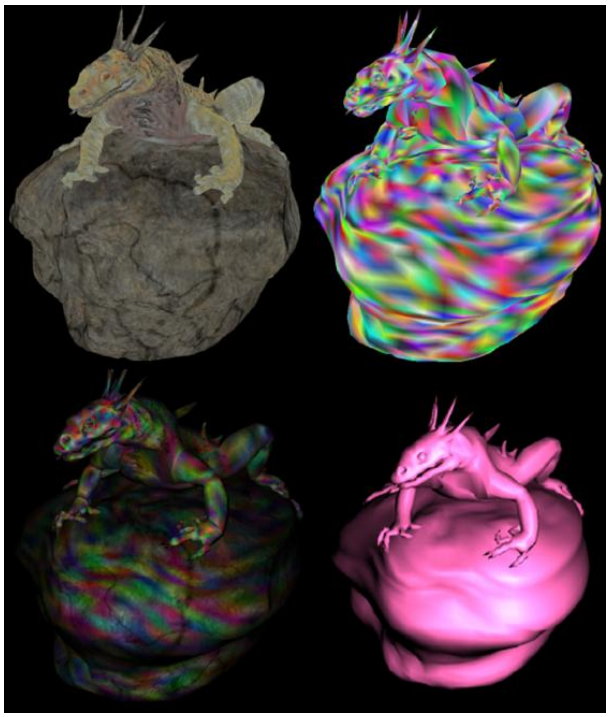

Figure 4-7 Upper left: Diffuse texturing. Upper right: Vertex coloring. Lower left: Diffuse texturing, vertex coloring, and per-pixel lighting combined. Lower right: Material colors and per-pixel lighting.

Normal Mapping

As we leave the basic material and move onto advanced shaders in the library, it would be sensible to begin with normal mapping first. Normal mapping is a technique to fake the lighting of rough surfaces without added geometric complexity. This is done by encoding normals in a texture, which is used instead of the geometry's vertex normals for per-pixel lighting calculations. The technique is also known as bump mapping, which was developed by James Blinn (Blinn 1978). This effect has become a common feature in games because it can give detail to models with a low number of polygons. Sometimes the normal map texture is created from a very high polygon version of the model (such as a character mesh of a million triangles), that would be applied to a very low polygon version (a character mesh with a few thousand triangles) and the two rendered models would virtually appear the same. Figure 4-8 shows this effect in the engine, on a Utah Teapot and a sphere with three point lights (red, green, and white). The sphere in particular was the first normal mapping test for the engine and is an example of how the effect can add detail to

the geometry. Both meshes are only several hundred triangles, yet appear to have indentions and bumps that a real rough surface would have.

Figure 4-8 Normal mapping examples in Spark Engine.

Implementation for normal mapping was based on the discussion for bump mapping in Nvidia's *The Cg Tutorial* (Fernando and Kilgard 2003). The shader was perhaps the hardest to implement than any other, not just because of the math involved, but also it required the most changes to existing engine classes. For example, the `MeshData` class needed to be expanded to include binormals and tangents and the engine's vertex formats had to also reflect this change. There were also some issues with space consistency – e.g. moving between tangent and world space. The implementation in the end transforms everything to world space for the calculations.

Figure 4-9 is a good example of the features supported by the shader. The normal map is often accompanied with a specular map (either in its alpha channel or as a separate texture) to control the specularity over the surface of the geometry. By default, the shader uses the model's material colors (if any) for specular highlighting, but a separate specular map can optionally be provided. An example of the differences can be striking, in Figure 4-9 the top left image of a lizard has a specular map that keeps the rock shiny, but not the lizard. The top right image has the lizard using the material's specular property, which makes the lizard appear too shiny and fake. However, both

images provide a strong contrast to the bottom image of the lizard rendered with the basic material shader.

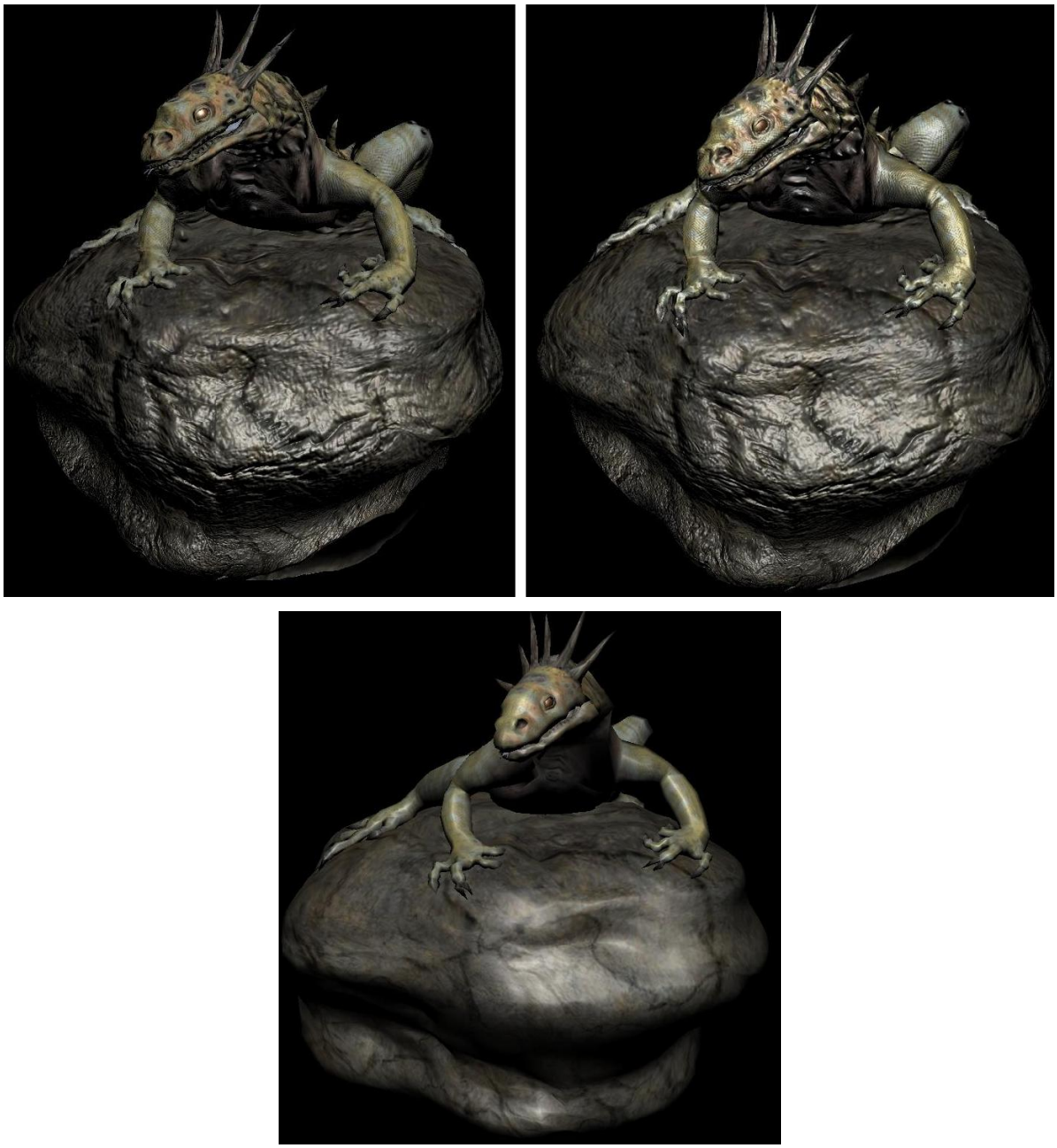

Figure 4-9 Top Left: Normal mapped lizard with a specular map. Top Right: Normal mapped lizard without a specular map. Bottom: The lizard rendered without a normal map.

Rim Lighting

Rim lighting is a method used to highlight objects and contribute to a unique style. Some usages of the technique in recent games have been to create a stylized cartoon effect. The inspiration for the shader implementation in the engine comes from one such game, Valve Corporation's *Team Fortress 2*. The game is a first person shooter that utilizes many non-photorealistic shading techniques that create a unique style uncommon to most games of its genre, which typically place a major emphasis on realism. Rim lighting in *Team Fortress 2* uses an impressionistic approach, where the rim lighting appears on game characters whether they are in indirect or direct light. This is because rim lighting is used to help players quickly and easily identify characters and objects during the fast paced gameplay typical of a first person shooter. In indirect lighting situations, the lighting effect only affects upward facing normals. This was an aesthetic choice to exploit the human instinct to assume that lighting tends to come from above (Mitchell, Francke and Dhabih 2007).

The engine's implementation of rim lighting follows closely to the method described for *Team Fortress 2*. It has the indirect rim lighting feature described above, as well as other options to adjust rim color and thickness. Figure 4-10 shows an example of the rim lighting effect in the engine. The image on the left uses the technique described above to get a rim effect that appears as if the light is coming from above. Meanwhile the image on the right uses the standard shader, where the entire model's outline is lit. Rim lighting could for styling purposes, such as in *Team Fortress 2* or for other purposes. For instance, the standard rim lighting effect is a good example of how the shader emphasizes the outlines of an object, which can be used by an application to identify objects selected by the user.

Figure 4-10 Left: Rim lighting for upward facing normals. Right: Default rim lighting. Both rendered with a back light, in order to create an outline effect.

Environmental Mapping

Environmental mapping is a simple yet effective shading technique. Figure 4-11 shows several examples of spheres and a Utah Teapot rendered in two different environments. The technique is also known as reflection mapping, and is a cheap way to approximate surface reflections with a pre-computed texture. This texture is typically a cube map of the surrounding environment, or skybox. However, this has drawbacks as the environmentally mapped object is required to be convex, since the effect does not allow for self-reflections. This is also true for nearby objects that are not included in the cube map, such as dynamic objects. The environmental mapping shader implemented for the engine is based on the technique described in *The Cg Tutorial* (Fernando and Kilgard 2003).

Generally for games it is not absolutely required to have a perfectly realistic effect, such as reflections. In most cases, an approximation that is good enough to fool an audience is all that is required, especially when it is used in tandem with other effects to create a complex material. For example, a surface that has uses both normal and environmental mapping to get a material that is

detailed, yet reflective such as brushed aluminum. The material may only reflect certain colors (such as the ground or sky) or light sources (such as the sun), but those reflections combined with other effects can distract the eye enough to give an impression that the geometry truly is reflective.

This is the current disadvantage of the engine's implementation of environmental mapping however. The effect is only purely reflective, giving a model the look of chrome. While this may result in attractive images, it generally is not very usable in practice since few objects are purely reflective. It would be desirable to someday revisit this shader to add an option for a base texture, with parameters such as how reflective the material is with specular and Fresnel terms. Another area of development would be this effect in tandem with other effects, such as normal mapping discussed earlier.

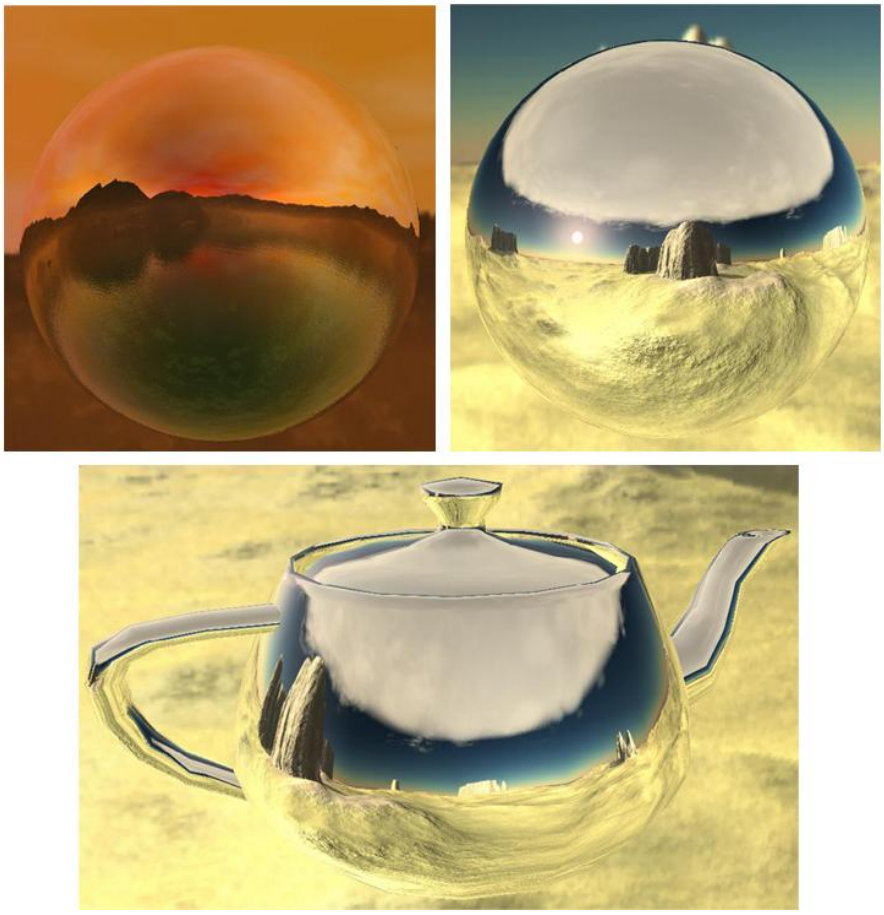

Figure 4-11 Examples of environmental mapping in the engine.

Terrain Texture Splatting

Texture splatting is the technique where several textures are blended together as dictated by an alpha map. The name was coined by Charles Bloom and the shader implementation for the engine was inspired by a description of his splatting techniques (Bloom 2000). The alpha map determines how the intensity of the texture's color over the entire surface, whether a texture should be completely or partially transparent. This often is used for terrain rendering, as a terrain's surface is seldom a single texture. Instead, a terrain has combinations of dirt, grass, rock, or other details, so texture splatting can mimic the details of a terrain quite well.

The engine's implementation uses a shader with four texture layers, an alpha map, and a detail layer. Each of the texture layers have their intensity dictated by a color channel in the alpha map – the RGBA channels. This could easily be extended to eight texture layers using two alpha maps. Alpha maps cover the entire terrain, which can lead to resolution issues. Generally it is a good idea to split a large terrain into a series of smaller sections, each with its own alpha map with a sufficient resolution. The four texture layers are not stretched over the entire terrain, but instead are repeated with a scaling factor. This can lead to tiling issues which breaks the authenticity of a terrain surface, because real life terrains do not tile. The detail layer helps break up the tiling effect by combining a small amount of noise in the final color of the terrain. A detail layer is a single texture that is stretched and repeated over the entire terrain using a scaling factor. It can repeat only once, meaning it is stretched over the entire terrain, or several times depending on the amount of scaling used.

Figure 4-12 is an image of a very high resolution terrain block rendered by the engine with a grass, dirt, rock, and sand textures layers (grass and dirt being the most visible). While a detail map was used, tiling is still noticeable due to the scale of the repeated base textures. Often this can be a challenging material to get just right for large terrains, and usually requires terrain modeling/generation tools.

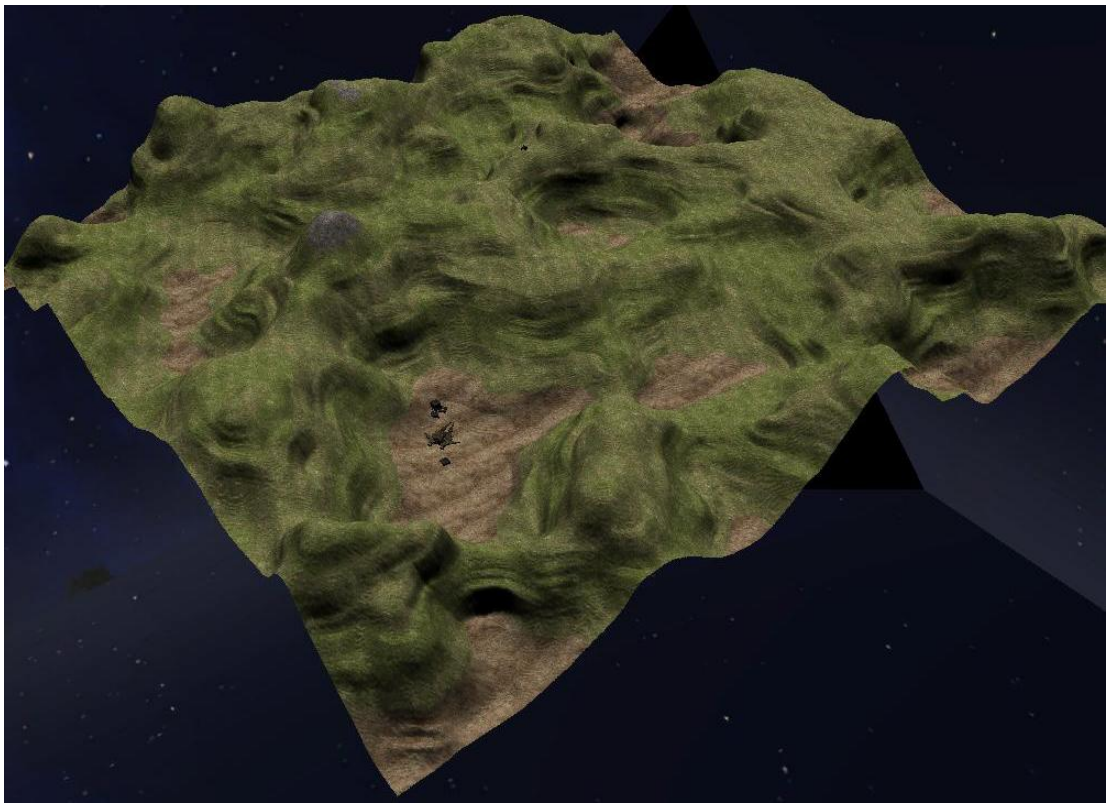

Figure 4-12 Terrain with grass, dirt, rock, and sand texture layers.

Light Mapping

Light mapping (Figure 4-13) is a technique where the lighting effects on geometry is not calculated in real time. Lighting is precomputed and stored in a texture, which is later combined with the geometry's base texture during real time rendering. The light map texture must be generated by a separate tool, such as the modeling software used to create the geometry. Therefore, the shader has two texture layers, the base color texture and the light map. This effect was developed late in the project to give the engine basic shadowing capabilities with the intent to develop real time shadowing in the future. However, light mapping is a common technique employed in real time rendering and is still common in today's games. While this is a useful effect to have as a feature in the engine's shader library, it does have limitations. The technique is only for static objects, and cannot be used on dynamic objects. Dynamic objects that are entering shadowed

areas of a static level for instance would obviously not have shadowing applied, which can be problematic for a scene that relies on lighting and shadows.

Figure 4-13 Example of a light mapped room with a single point light source. The character is lit in real time, as it is a dynamic object.

4.4. Input System

The last few sections focused on the rendering capabilities of the engine, now we shall shift focus to a minor, but important system that was developed for the engine: user input. Spark Engine's input system is built on top of XNA's input classes for the keyboard, mouse, and Xbox 360 gamepad. The state of these devices can be queried to determine if buttons are pressed or which keys are held down and so forth. However, XNA requires the developer to create their own logic for higher level functionality – such as disallowing repeats if a key is pressed or determine if a key has been released since the last update, which requires device states to be kept. The engine's input

system remedies this lack of high level functionality and provides an easy to use API to organize inputs and associate actions with those inputs.

In the engine, an input is designated as an `InputTrigger`, which is composed of an `InputAction` and an `InputCondition`. During an update, when the trigger is processed, it first checks if the condition is true, and if so performs the action. This simple and straightforward system allows for quick creation of user input, because these classes can be reused. For example, the engine provides condition classes for testing if a key is pressed and if repeats are allowed, or if a key is released.

Furthermore, the condition and action classes are designed to be used with delegates. Delegates in .NET are the basis for event driven programming and resemble that of function pointers in C/C++. They can also be used to create anonymous methods. Both condition and action classes can be used directly by passing a delegate to their constructor, which is called when the condition and action are processed. Often times a condition may be a simple if-statement and would not require the developer to extend the `InputCondition` class. This helps reduce code and remove the need for many small classes. The use of anonymous methods is a direct inspiration from experiences working with Java event listeners, which often are created as anonymous classes.

In order to organize and process input triggers, the engine provides the `InputLayer` class, which manages and updates a collection of input triggers. As its namesake implies, this allows for input to be layered. For example, common input triggers can be grouped together that allow the engine to enable or disable them at the same time. Such as during a game's cut-scene, movement controls should be disabled but the player should be able to still access menu options or quit the game. Different input layers can correspond to different players also, such as in a split screen game. Or perhaps certain input triggers should only be active when a certain menu screen is active. This is useful if there are multiple screens layered over one another, when only the topmost menu screen should receive any input (such as a menu that binds keys to commands). The input system is designed to be flexible enough to support all these possibilities while allowing code re-use.

4.5. Animation System

Spark Engine's animation system is a shader-based implementation that has its roots in one of the original goals for the honors project, as hardware accelerated skinning was a potent shader topic. Hardware accelerated skinning was resurrected as a needed feature by the spring semester's design project as an alternative to using a third party library. The goals for the animation system were twofold:

1. To be an introduction to hardware accelerated skinning.
2. A system that is integrated and elegantly designed to fit inside the engine, opposed to relying on a third party library.

The animation system's design was fairly basic and straight forwarded. It included classes for key frame data, bones, and animation clips. An animated mesh's skeleton is similar to that of the `MeshData` object, as it is designed to be shareable between different animated entities. The current pose of the skeleton is kept separately, which is updated for each new key frame and is responsible for creating the matrix palette sent to the shader. Extensions to the existing `Node` and `Mesh` classes hold a reference to both the skeleton and the pose. These are the `SkinNode` and `SkinnedMesh` classes respectively.

A `SkinnedMesh` is always attached to a `SkinNode` object, which acts as its controller. `SkinNode` holds a reference to an `AnimationManager` object that manages `AnimationClip` objects associated with the skeleton, as well as provide the functionality in playing and stopping those clips. The animation manager contains `AnimationClipState` objects for each clip. These state objects track which key frame is currently being played, as well as manage the clip playback, speed and looping properties. By separating the state of an `AnimationClip`, the key frame data is able to be re-used between multiple skinned characters.

Since the system uses a shader approach, all skinned meshes require a skinned material definition to communicate the matrix palette to the shader, which the engine provides. The shader implementation is based upon the description of vertex skinning from *The Cg Tutorial* (Fernando and Kilgard 2003). The shader allows a skeleton to have up to 59 bones and supports some of the rendering options offered in the basic material shaders, as can be seen in Figure 4-14. This includes

per-pixel lighting using the engine's lighting library and diffuse texturing. However, the effect does not support any other advanced effects such normal mapping. This could be an area of future development.

It should be noted that the XNA Framework does not directly support animation. However, the FBX Importer in the framework's content pipeline does parse animation data, such as bone and key frame content. In order to use this data, a custom processor needs to be created. Commonly, XNA animation libraries are used if developers do not wish to implement their own solution. Although one of the goals of this project was to implement features such as animation from scratch, some of these XNA animation libraries were reviewed for potential use in the design project.

Figure 4-14 An animated character walking.

Future Considerations

In its current form, the animation system is still fairly basic. The system does not support blending between animations or procedurally created animations. Blending capabilities is an important mechanism for smoothly transitioning between animations, such as from a walk cycle to

a running cycle. In addition, extensions to the idea of clip states can be taken to the next level. That would be the ability to define animation clips as states, and be able to link states together. For example, idle, walk, and running animations can be linked together and the system automatically transitions between them depending on how fast the character is translating. This would allow for the system to easily mesh with input controls and free the developer from writing the logic to make these transitions.

4.6. Content Pipeline

XNA provides an easy to use content pipeline, and is one of the few areas of the XNA Framework that the engine uses significantly. Content is built as a separate project and the pipeline is only for the Windows platform. Every form of content – models, textures, fonts, etc are compiled down to a portable binary format (the *.xnb* file). The XNB binary file can be loaded on any of the supported XNA platforms (Windows, Xbox 360, and Zune). Since system memory is at a premium for the Xbox 360 and Zune platforms, content is an offline process in order to prepare content to be in an efficient runtime format and perform all computations and processing before the asset is loaded. This speeds up loading and provides a unified architecture for all art assets.

The XNA pipeline is divided up into two sections:

1. **Importers** – Asset importers that convert data from file formats into an intermediate format. Such as the *.fbx*, *.dxs*, and *.x* formats which are converted into XNA's `NodeContent` intermediate form.
2. **Processors** – Asset processors that convert the intermediate formats to run-time friendly formats.

This is a powerful and extensible architecture, as it allows many importers for different model formats that are fed into a single processor. Since the content pipeline classes for the engine are their own project, they are a separate assembly from that of the engine assembly. Content cannot be built directly into engine runtime classes due to circular dependency, thus this required an implementation of intermediate content classes that mirrored the engine runtime classes. Since the content and runtime classes are different, this requires the use of XNA binary writers and readers which write and read from XNB files. Binary writers point to their reader counterparts, to identify how content should be imported at runtime.

Spark Engine uses a single pair of writers and readers that read in content that implement the `Savable` interface. All scene graph objects implement this interface. The engine provides a `BinaryImporter` singleton class that manages a collection of `BinaryContentModule` subclasses that the reader uses to create the necessary runtime class. There are modules for the `Node`, `Mesh`, `SkinNode`, and `SkinnedMesh` classes. These modules also parse the necessary material and animation information needed by their respective runtime classes.

This is still an area of the engine that requires more design and development. Ideally, serialization would be preferable, to reduce the redundant intermediate content classes and work with the run time classes directly. This would also allow those classes to be saved back into a XNB file. However, this poses a problem since serialization relies on reflection which is slow and the compact .NET Framework that is used by the Xbox 360 and Zune platforms does not support the entire reflection API. The current design for content importation is a compromise to forego the use of reflection but still be able to maintain flexibility since the scene graph is intended to be extended. Developers can create new types of scene objects, and use them in conjunction with existing scene objects, since each content module only reads in the data their runtime classes require. For an example, a node content module reads in transformation data, its number of children, loops over that number, and identifies and calls the content module that is associated with the child type.

This is a divergence from the model processing and model format that come standard with the XNA Framework. XNA's `Model` class is sealed and cannot be extended nor is it meant to be. Everything for the `Model` class is computed up front by the pipeline and cannot change at runtime, which poses problems for real time procedurally generated geometry. To address this shortcoming, the engine's implementation of the `MeshData` object was specifically designed to allow geometric information to be changed at runtime.

Importers and Processors

The engine provides support for two custom model processors: `MeshProcessor` and `SkinnedMeshProcessor`. The former returns a scene graph (where the root is a `Node`) that represents the geometry, as it was organized in the input file. The latter returns the geometry as a

`SkinnedMesh` attached to a `SkinNode` with the pertinent skeleton and key frame data. For all other content, such as textures and fonts, the default XNA importers and processors are used. The XNA Framework supports Autodesk's FBX and Microsoft's X model formats. An additional format, DeleD's DXS format, is supported by the engine with a custom importer. Spark Engine supports animation for the FBX format only.

Each of the model processors support the concept of artist created collision volumes to approximate geometry. This system supports all three bounding volumes and follows a strict naming convention:

cv_type_meshName_xxx

Where type is either "obb" for oriented bounding box, "aabb" for axis aligned bounding box, or "sphere" for bounding sphere. The convention is case sensitive, and the mesh name must correspond to the name of the geometry that the collision volume represents. Appended at the end of the name should be numbering, to ensure the name is unique. Geometry that is named with this convention are automatically extracted and processed into a XML physics file. This file is saved alongside with the XNB file for future use.

Figure 4-15 shows an image of a set of collision volumes for a room rendered by the engine's debugger. While normal bounding volumes would encapsulate the entire mesh, the collision volumes clearly approximate the room geometry. This requires more volumes to check, but is faster than checking all the triangles of a piece of geometry. This is especially true for geometry with a large number of triangles. The concept for collision volumes is based on a common practice employed by many game engines. First person shooter games frequently use them to not just speed up collision checks, but to make movement smoother for players. These types of game have fast paced gameplay and becoming snagged on geometry is simply unacceptable. This also lends itself to easily allowing players to slide against geometry during collision, since collision volumes are simpler shapes.

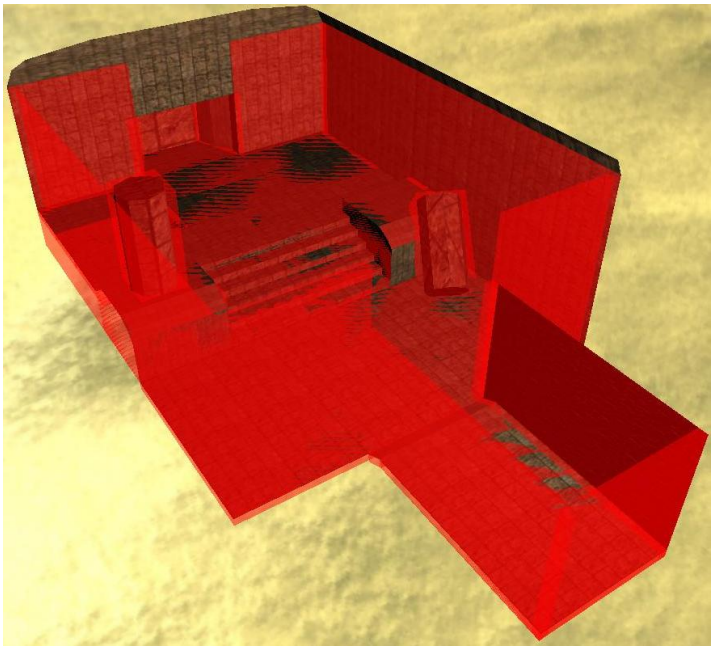

Figure 4-15 Collision volumes rendered in the engine.

Future Considerations

In order to reduce the amount of redundant content classes (currently each runtime class for scene graph objects, bounding volumes, and materials have mirrored content classes in the pipeline) it may be more efficient dealing with the runtime classes directly. This would require the engine to be split up among multiple assemblies. Also, the ability to save the scene graph back into a XNB file, or some other format is future development worth investigating. However the heavy use of reflection may pose difficulties with porting the engine to the Xbox 360 platform, which is an eventual goal. Currently the engine was developed for the Windows platform and is only tested for that, but theoretically could work on Xbox 360.

4.7. Application Usage

A typical application that uses Spark Engine follows closely to that of how an XNA game is created. The XNA Framework provides a Game class that manages the game loop. This serves as the

entry point to the application, and provides the means to update and draw the scene graph each frame. The reason why the engine uses the XNA Game class in lieu of its own implementation is because the XNA implementation is quite sophisticated – it will try to keep the number of frames rendered per second to a target value. For example, the XNA Game class by default targets 60 frames per second. The advantage for this is to prevent the game loop from running as fast as possible, as that is an unnecessary waste of resources due to the refresh rate of monitors.

The Game class also provides support for game services and components, which is a system that XNA employs for developing independent modules for games. A game service is an interface for an object that functions as a singleton. Any module can query the Game class for a service that implements the interface, without having to require an explicit reference to the service object. The engine utilizes this system for the concept of game screens, which is how engine systems such as the scene graph, scene renderer, and input system are combined with XNA components.

A game screen represents a portion of the screen – thus it requires a scene renderer and a camera. It also supports an input layer and holds a reference to a scene graph. Screens can be chained together in hierarchy – every game screen can have a number of child game screens, each of which can have their own scene data, input layer, and renderer. A typical usage of this feature would be to have the root screen represent the 3D world space, and children game screens would be the user interface and menu systems. This organization automatically allows layering of user interface elements, since child screens are drawn on top of their parent. Additionally, this can allow for easy compositing.

Since scene, input, and renderers are their own modules these objects can be shared between screens also. For example, a split screen game would have two game screens with two renderers (a camera for each player), two different input layers, but both screens would use the same scene data. Game screens are handled by a game screen service that is registered with the XNA Game class automatically by the engine at startup.

5. Performance and Optimization Work

In all real time graphics application, speed is certainly the key. Good performance is a must and simple optimizations can drastically influence an application's performance. Conversely, subtle details can adversely impact how well an application runs or even halt it completely. Attaining an idea of how well the engine functioned under stressed conditions, and attempting to optimize it became a large focus late in the project.

The discussion of the rendering system (Section 4.2.) touched on the subject of optimization with the use of render state caching and sorting geometry before drawing them. Sorting geometry and preventing redundant state switching can significantly increase the number of objects that the engine can render in a single frame. For example, the original culling testing described in Section 4.2 only had 1024 individual draw calls. When all the meshes were visible, frame rate dropped to 20-30 frames per second. After the introduction of the new renderer functionality, the cull test ran at a consistent 60 frames per second even when the entire scene was visible. This required the cull test to be expanded to an object count of over 3375 (each a unique draw call), which saw a similar performance drop when the entire scene was visible. This was a 3.3x increase in the number of draw calls the engine could perform until performance dropped. In addition, the new test used more textures and required more state switching than the old test. Although the number of different materials in a scene would be much higher than used in these tests, material sharing is a common optimization trick. A typical high end game may have upwards to 4000 draw calls, so these results were very promising.

While this is easily one of the more effective optimizations, it was still a significant feature that the engine was previously lacking. The rest of this section will discuss other such features that improved performance, as well as dealing with the subtle details of optimization such as memory allocation. All of this effort was in the pursuit of gaining as much performance out of the engine as possible so that it could be competitive with other similar software.

Frustum Culling

An optimization implemented in the engine was the ability to cull branches of the scene graph while the engine processes it for rendering. This feature was an early ambition for the engine, since XNA does not have built in methods for processing data that will be rendered. If all the meshes in a scene graph were allowed to be sent to the graphics card, the application's performance would drop off very quickly. Small scenes may not adversely impact performances, but large scenes certainly will. It is a tremendous waste of resources to draw something that ultimately will be clipped from view by the graphics card. For very large scenes, the application may freeze up completely. Most high end professional games today do not go beyond several thousand individual draw calls. Therefore aggressive culling is absolutely required for a scene that potentially has tens of thousands of objects in it.

Figure 5-1 presents screen captures of two iterations of the engine's frustum culling test, which used a large 3D array of cubes. The picture on the left is the original test that contained 3,375 cubes with 12 triangles each. This yielded a 40,500 triangle count for the scene, a number that a graphics card can easily handle. However, the focus in this test is not on triangle count, but on individual draw calls. This may not be the case for other situations, such as rendering large terrains containing millions of triangles. In that case, the terrain would be split up into a number of smaller pieces, each with its own bounding volume that allows those blocks to be culled when not in view. The frustum culling test ran at a consistent 60 frames per second when only a portion of the 3D array was visible. Performance dropped to 20-30 frames per second when all the cubes were in view.

The screen capture on the right is an interesting side note that arose from the original cull test, as a mistake during one execution of the test yielded in a 3D array with the dimensions 15 by 150 by 15. This meant 33,750 cubes were in the scene, with 405,000 triangles total. That many individual draw calls would slow down the application to the point where real time interaction would be non-existent. In fact, this was the case when the entire (or a reasonably large chunk) of the scene was in view. However, when only a fraction of the cube array was in view, the application still ran at a consistent 60 frames per second. The significance of this case is that it demonstrates that the engine is capable of managing and processing tens of thousands of objects in its scene graph. While care needs to be taken in such extreme cases, it was good to gain an understanding in how big of a scene the engine can handle while updating and rendering the scene.

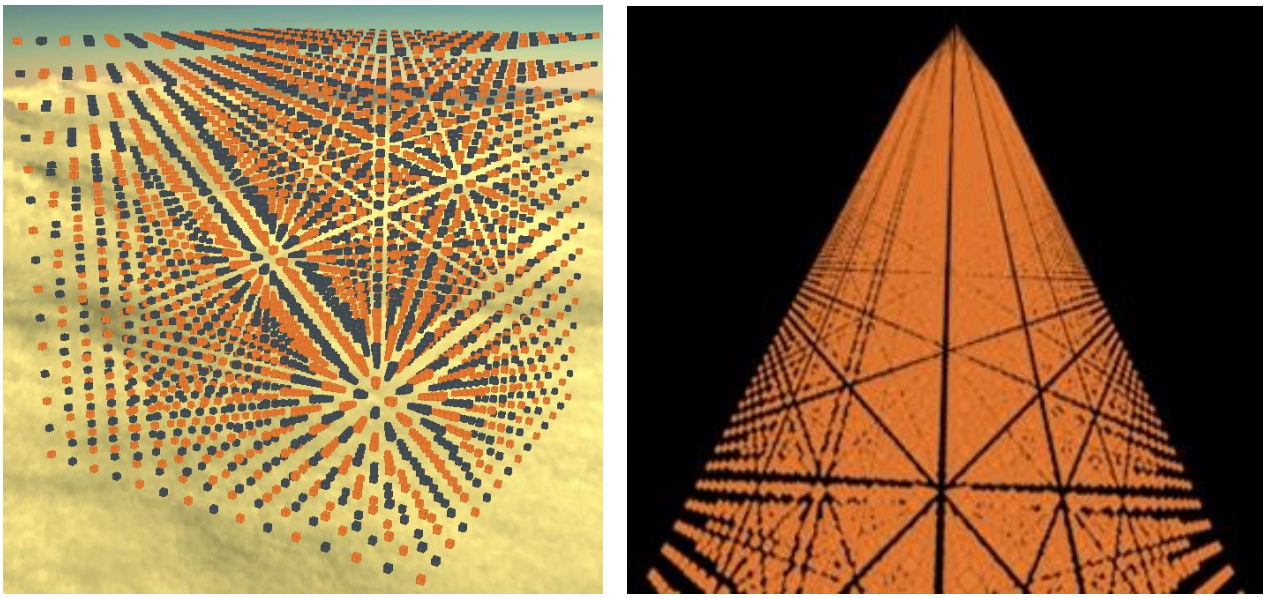

Figure 5-1 Left: A 15 by 15 by 15 array of cubes. Right: An erroneous 15 by 150 by 15 array of cubes.

Good Programming Practices

The saying "the devil is in the details" is a very good way to sum up the engine's development. Many lessons were learned the hard way in how little details can adversely impact a program significantly – either a reduction in the application's performance or outright crashing it. Most of these instances had a common issue – memory management. The C# language is managed, meaning it has a garbage collector and the developer does not have to explicitly de-allocate memory. However, this does not exclude C# developers from understanding memory – it still is one of the most important concepts that can make or break a graphics application.

In addition to reference types such as a class, C# has the concept of a value type. Value types are allocated on the stack and consist of two categories: structs and enumerations. Therefore they do not produce garbage, as they are not heap objects. Since structs support inheritance from interfaces and extend from the root Object class, they can be considered light weight classes. Small data types like vectors or matrices are typically implemented as structs. In fact, primitive types in C# such as *float* and *int* are structs.

The use of value types is very important to the engine and any XNA application. Most XNA applications run at 60 frames per second, which can be thought of as a tight loop. If garbage is created each time the scene graph is updated and rendered, then the application can face significant slowdowns. Small scenes that have several dozen or several hundred scene objects may never see a performance drop, but it is inevitable for large and complex scenes with thousands or tens of thousands of objects. A large scene over the span of several seconds can generate hundreds or thousands of pieces of garbage which would cause a significant drop in frame rate. Furthermore, the application may periodically become unresponsive, since the garbage collector is executed for too long or too often. On the Xbox 360, the garbage collector is run when 1 MB of heap memory is allocated. The engine had to be able to handle large scenes without suffering a performance drop due to garbage generation otherwise its development would not have been justifiable. Therefore this was an area of interest late in the development process.

The most obvious way of eliminating garbage is not creating it in the first place. It is better to allocate memory for everything up front and then reuse those objects. For example, the scene graph's world bounding volumes originally were recreated every time they were merged with another bounding volume, returning a completely new instance. This was changed to have a spatial's world bounding volume created the first time it would be updated, and then reuse that instance whenever it needed to be resized. For dynamic scenes that constantly had its bounding volumes updating each frame saw a tremendous increase in frames per second after this was changed. Reusing objects was also employed for other properties of the scene graph, such as transformations, render states, and so on.

Another example of optimization was how the engine sorts geometry in the render queue. Originally the implementation used .NET's default list sort. This was a quick sort algorithm that generated garbage, and often took 6-8 ms to perform. For 60 frames per second, there only is 16.6 ms for updating and rendering a single frame – therefore this was unacceptable performance. The engine implemented its own sorting, using the more stable merge sort. It also maintains a temporary scratch array for the merge sort algorithm as meshes are added to the render queue, only resizing it as needed. The sorting typically runs 1-2 ms, which is a tremendous difference from using .NET's implementation.

The above optimizations were found to have a significant positive increase on engine performance, but are only a part of the overall picture. In order to minimize garbage creation in each frame, the engine uses structs in favor of classes wherever practical. The XNA Framework provides all of its math objects, such as Vector3 as structs so it was only natural to extend this. However, structs can become pitfalls if care is not taken. Since value types can inherit from interfaces and do inherit from the root Object class there are cases where the value type is "boxed".

Boxing is the process of converting a value type to an object or interface, which generates garbage because the value type is "boxed" inside a heap object. This can be an often overlooked detail that can be fatal for a graphics application. For instance, if a method calls for type Object to be passed in its method signature, a value type that is passed is automatically boxed and garbage is created every time the method is evoked. Developers may unknowingly create garbage while trying to minimize its creation by using value types. Another good example of this is the use of enumerations as a key in a dictionary – they are boxed when the dictionary performs key comparisons. This actually posed a problem with how render states were organized in the scene graph, as they use an enumeration for identification. To prevent boxing, the dictionary can use integers as keys and the enumerations cast to an integer when accessing the dictionary. Another example would be to use *for* loops, instead of *foreach* loops to avoid the generation of enumerator objects for constantly changing lists of data. These practices were employed in every part of the engine in an effort to gain as much performance as possible, by reducing unnecessary heap allocations.

6. Future Work

Many more features could easily be added to the engine, or existing systems expanded and enhanced. A significant area to develop would be an increased focus on actual game systems – that is to create a platform that uses the engine to easily put together games. Currently the engine is more in line with that of a 3D graphics engine than a game engine since most of the focus of this project has been to develop the rendering capabilities. However, due to the needs of the design project that the engine was developed to be used for, Spark Engine does support some game centric concepts such as game entities and collision volumes. Game entities are specific to the senior design

project however, although they are provided by engine classes. Entities can have attributes added to them as well as triggers that are analogous to input triggers and spatial controllers. This allows for entities to respond and change their attributes when a condition calls for it. For example, if a character in a game enters a new area, a trap is triggered and interactive gameplay follows.

Future development for the engine can build upon these basic classes and expand Spark Engine to fully encompass the role of a game engine. In addition to more game-specific modules, a toolset would be a crucial requirement in the development of a game platform. Without quality tools, it is very difficult to build complex games with immersive worlds. Such a toolset would include tools for world building, mesh editing, entity creation, material editing, and so on. Although this honors project has concluded, it is the eventual goal to create dedicated tools for world editing to facilitate the creation of complex scenes which is currently a tedious and time consuming process. Tools can provide access to the engine for non-programmer users, such as artists who are interested in using the engine to render their content.

Ideally these additions would be a layer built on top of the engine. This would allow the engine to remain flexible – if a developer wants their own AI implementation or organization of backend systems for world building, they can plug their own implementation into the platform. Or the rendering and scene graph functionality described in this paper can be reused without any game specific modules if so desired.

7. Conclusion

As discussed in this paper, the development of Spark Engine has produced significant results. The engine is a fully functional graphics platform that can be used to build games or general graphics applications. It is feasible to obtain the engine source, compile it, and develop a quality application on top of it. The engine streamlines XNA development as it provides systems that would either have to be implemented by the developer or taken from third party libraries (e.g. one of the many XNA animation libraries). Such alternatives were the purpose of creating Spark Engine. Designing and implementing the majority of the engine's systems in order to be reused many times and in different contexts without rewriting code was the overarching goal, and was a success.

The hands on experience gained from the exploration of engine architecture and shader programming has proven to be invaluable. Due to the nature of the engine, many computer graphics topics were required to be explored and experience was gained working in those areas directly. Furthermore, valuable experience was gained from working with a complex and large piece of software. Many of the engine's systems were finely tuned to work in unison with one another, and the engine codebase grew well over twenty thousand lines of code. Typically an engine would be created by a team of developers. A single developer would not have to work with the entire piece of software in its entirety.

Therefore, with Spark Engine's core features developed and lessons learned, this honors project achieved what it originally set out to accomplish. While the engine can be developed further and numerous features can be added to it, the original design goals were satisfied and the necessary components implemented to create a functional game engine.

8. Works Cited

Blinn, James. "Simulation of Wrinkled Surfaces." *ACM SIGGRAPH Computer Graphics*, 1978: 268-292.

Bloom, Charles. *Terrain Texture Compositing.* November 2, 2000. http://www.cbloom.com/3d/techdocs/splatting.txt.

Eberly, David. *3D Game Engine Design: A Practical Approach to Real-Time Computer graphics.* Morgan Kaufmann Publishers, 2007.

Fernando, Randima, and Mark J. Kilgard. *The Cg Tutorial.* Addison-Wesley, 2003.

Mitchell, Jason, Moby Francke, and Eng Dhabih. "Illustrative Rendering in Team Fortress 2." *International Symposium on Non-Photorealistic Animation and Rendering*, 2007.