TEAM 12: TERMANATOR

PROJECT PROPOSAL

TEAM MEMBERS:

Donald Eng

Rodrigo Ipince

Kevin Luu

1. INTRODUCTION:

This project involves the design and implementation of a unique, first-person shooting game. The system is divided into four primary components: Devices, Inputs, Game Engine, and Graphics.

The basic structure of the game is that creatures will be coming towards the user from a pseudo 3-D perspective. If the creature touches the user, the game is over, and the high score is recorded. To stay in the game, the user must destroy each creature with a special gun (the user's input).

The gun consists of two devices that will be used as the user's input to the game. In the game, the gun will generate streams of blocks, targeted at the creatures coming at the user. That stream of blocks is controlled by two devices: the pointer and the shaker. The pointer directs the stream of blocks to where the user is aiming. The second device, the shaker, provides the corresponding power level of the gun; the higher the power level is, the more intense the weapon will be.

With the weapon, the user has the ability to move the direction that the gun is pointing, so that he/she has the capability to shoot other creatures. In addition, the user can change the angle that the gun is pointing based on its vertical position. Through this, the user can shoot over a particular creature if he/she is targeting a creature in far sight. One note is that in order for the gun to operate at its full potential, the shaker needs to be constantly in motion, so that the weapon is powered, and streams of blocks are targeting specific creatures.

2. MODULE SPECIFICATIONS:

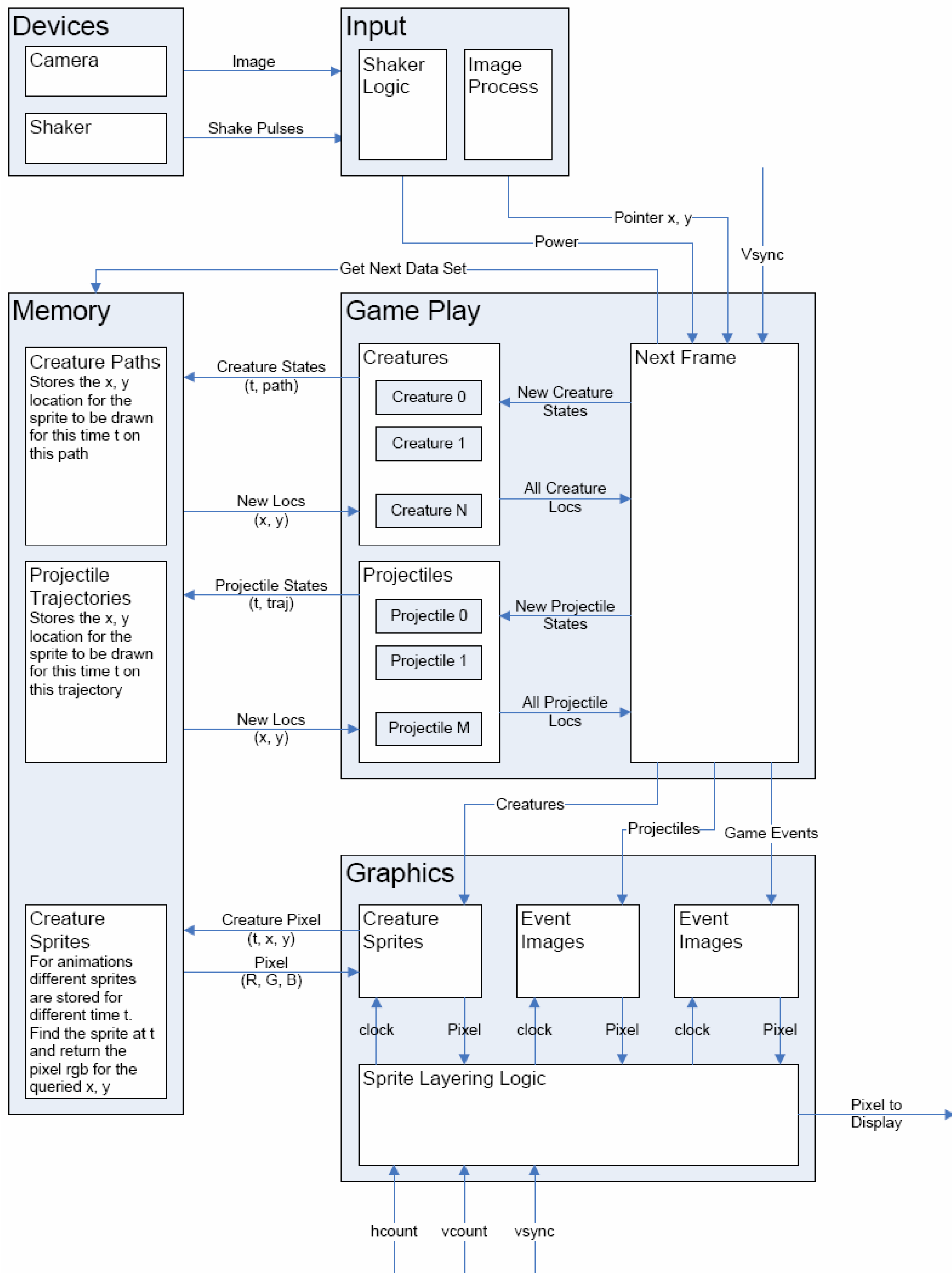

Figure 1: High level block diagram

The implementation of the project is divided into four major sections: the devices, the inputs, the Game Engine, and the Graphics. Each block handles different sets of functionality and, ultimately, contributes to the overall system. Below is a description of each block.

2.1. DEVICES COMPONENT: (Implementation by Kevin Luu)

2.1.1. Camera

A camera will be used to continuously output camera images. The camera's quality needs to allow the input handling modules to easily distinguish the location of the marker on the player's point devices through difference in color intensity. This device will be borrowed from the lab.

2.1.2. Shaker

The shaker device for the user would be a customized tube with two buttons on both ends, which contains a ball that will make contact with the buttons as the user shakes the tube in a vertical motion. Since the press of a button will generate a single pulse, the module will send continuous pulses with intervals depending on the speed of the shaker. The interval between pulses would be proportional to the intensity of the stream that would be used at that instance in the game. For example, if the device is shook vigorously, the time elapsed between the click of the buttons will be small, which correspond to more pulses occurring in a given time interval.

2.2. INPUTS COMPONENT: (Implementation by Kevin Luu)

2.2.1. Video Process

This module takes in a stream of pixels from the video camera in the YCrCb color space and store the information into memory. From the image in the memory, the module will detect the location of the marker, and display it accordingly onto the screen. The use of YCrCb color space enables a more precise means of recognizing the marker. Aside from detecting the traditional RGB pattern, YCrCb utilizes the measures of luminance and chrominance to accurately distinguish differences in color intensity. Through this, the user will be able to use the marker as an interactive pointing device to direct the stream in the game.

During the start of the game, the player will be asked to locate their marker onto a designated rectangular area on the screen as a mean of initializing the input to the game. From this, this module will try to locate the marker through detecting differences in color intensity between the marker and the background. From the initialization process, the module will store the exact color specification of the marker, and will be using these values as thresholds in detecting the location of the marker at every frame.

During game play, each pixel will be compared to the stored pixel ranges that resemble the color of the marker. Through comparisons of all pixels in the image, the module will be able to obtain a set of *x*, *y*-coordinates that translate to the location of the desired pixels on the VGA screen. However, noise can be introduced through numerous devices, including the quality of the camera and the lighting condition of the game area.

2.2.2. Medium Filter

In response to possible noises that can influence the overall marker detection, a medium filter (incorporated into Image Process Module) will be used as means of eliminating unwanted noise. The medium filter operates by checking four successive pixels to any pixels that might resemble the marker. This notion is based on the fact that the desired object to be detected is a solid figure that is much larger than a single pixel, meaning that the object should have a continuous number of pixels adjacent to one another. The medium filter checks to see if there are adjacent pixels that are also selected. With this filter, it is believed that all extraneous pixels that are mistakenly detected from the camera will be eliminated, resulting in a more precise representation of the marker's location.

As outputs from the medium filter, the *x*, *y*-coordinates of the resulting pixels will be used to compute the center location of the marker. This can be done through averaging the *x* and *y*-coordinates separately. These values, which represent the *x*, *y*-coordinates of the marker's center of mass on the VGA screen, will become outputs of this module.

2.2.3. Shaker Logic

This module is responsible for determining the intensity of the stream in the game. Inputs to the module are pulses that are translated from the speed of the shaker. Since the interval between consecutive pulses is proportional to the intensity of the stream in the game, the module will interpret the shake pulses and output the power level of the stream corresponding to how fast the device is being shaken. For example, more pulses occurring in a given time interval indicates that a higher power will be generated for the stream. Once the power value is determined from the shaker pulses, the module will send the determined power level to the game engine.

2.3. GAME ENGINE COMPONENT: (Implementation by Rodrigo Ipince)

The Game Engine Component is responsible for generating all the digital signals needed to represent the game abstractly. That is, it must be able to determine when the game starts, when it ends, and how the game evolves through time. The functionality of the component can be broken down into three main parts: game evolution logic, projectile trajectory generation, and creature path generation. The main task of this component amounts to creating a good model for the generation of creature and projectile positions. Once this is done, the game evolution logic should be fairly simple to implement. Currently, two different approaches are being explored: a Lookup method and a Real Time position generation method.

The Lookup method involves a discretization of the game, which means that only a set number of creature paths and projectile trajectories will be used. These can be stored in memory as sequences of *x* and *y* coordinates and accessed whenever they are needed. This approach is useful because it gets rid of the complexities introduced by trying to model 3-dimensional space on a flat screen. However, since the amount of paths that can be stored in memory is limited, it would take away from the continuous flow of the game.

The Real Time approach involves calculating all the trajectories in real-time. Therefore, each position is represented with three coordinates, and the task of transforming this 3-dimensional model into a 2-dimensional one with the desired perspective is left to the graphics component. This methodology would yield the desired continuous game, but since the transformation will presumably be computationally intensive, the lookup table method has not been discarded.

2.3.1. Creatures

To simplify the control of the creature movements on the screen, there will only be four allowable paths for the creatures to walk down through. Once a creature starts off in a path, it will stay in the same path, getting closer and closer to the front of the screen as time progresses. Also, to simplify the problem of spawning and destroying creatures, a set number of creatures will be used at all times. Each of these creatures can be in an active or inactive mode, which will determine whether they are displayed on the screen or not. Therefore, simply making these creatures loop around the paths (i.e., relocating each creature to the beginning of the path once it reaches the end) will achieve the desired effect. To avoid making the game monotonous, every time a creature reaches the end of the path, the path it will take during its next iteration will be determined randomly.

Basically, this module maintains an array with each creature's current path, position, frame (determines how far along the path is the creature at), and mode (active or inactive). This array gets updated according to the new creature state information coming from the game evolution module (Next Frame module in Figure 1). The module then passes out the new locations for all creatures back to the game evolution module, so it can use them to determine how the game evolves thereafter. For each creature, the game evolution module provides a new path, mode, and frame. Notice that a creature's path should only change when the creature is being relocated (looping back), and that its mode should only change when it has been hit by a projectile or when it's being relocated.

Now, depending on which approach is being used, the way new positions are assigned to each creature will be different. If the Lookup method is used, then the new position of the creature is simply fetched from memory by specifying the path and frame. This is depicted in Figure 1. If the Real Time generation method is used, the new location can also be easily obtained. For any creature, its *x* and *y* coordinates (length and height on the screen, with the origin in the bottom left corner) are completely determined by the path in which it's in, and its *z* coordinate (depth) is simply a linear function of the current frame.

2.3.2. Projectiles

The projectiles will be handled in the same general fashion as the creatures. There will be a fixed number of projectiles, each of which can be active or inactive at any point in time. Again, this module will maintain information about each projectile's trajectory, position, and mode, and will update it according to the new state information coming from the game evolution module. The main difference is that projectiles have more than four possible trajectories. In particular, the number of trajectories depends on which approach is being used. But in any case, each possible trajectory will be uniquely determined by specifying an initial angle, initial position, and power (which would physically correspond to an initial velocity).

If the Lookup method is used, then determining the new *x* and *y* coordinates for each projectile is trivial; the position just has to be fetched from memory. If the Real Time generation method is used, then some calculations have to be made to determine the three coordinates needed. First, notice that the *x* coordinate will always remain the same, since it is assumed that the projectiles only travel in one plane (that which is parallel to the plane spanned by the *y* and *z* axes). Next, the *y* component can computed from the current frame with a simple kinematics equation. The same can be done with the *z* component, as the projectile travels with constant velocity in this direction. A possible issue with this method is that multiplication and addition of fairly large numbers might be needed, so timing might cause a problem.

A detail worth pointing out is that when referring to the Lookup method, the *x* and *y* coordinates of an object represent the position of the object *in the screen*, whereas when referring to the Real Time generation method, the *x*, *y*, and *z* coordinates of an object represent the object's position in a 3-dimensional system of coordinates, whose origin lies in the bottom left corner of the screen. Thus, the *x* and *y* coordinates will be the same for both methods *only* if in the 3-D model, *z* equals zero.

2.3.3. Game Evolution

This module is the Next Frame module as depicted in Figure 1. It is responsible for generating all the game event signals, as well as supplying the graphics component with the positions of all the creatures and projectiles in play. Game event signals provide information about the game itself, such as whether it is starting, ending, or paused, as well as about happenings within the game, such as creatures being hit by projectiles. The module will receive the location of the pointer, the power of the stream, and some control signals from the inputs component, in addition to the path/trajectory, position, frame, and mode of all objects, coming from the creatures and projectiles modules. This

module basically implements the core of the game functionality. This involves dealing with many different subproblems, which are individually described below.

2.3.3.1. Game starts or pauses

This module determines whether the game is starting or paused directly from the signals coming from the inputs component. A timer will be kept throughout each game; it starts when the game starts, and pauses when the game is paused.

2.3.3.2. Game ends

Determining when the game ends is very simple. If the Lookup method is being used, then the module just checks if an *active* creature's position is too close to the bottom of the screen. Else, the module just needs to check for active creatures whose z coordinate is small enough. When the game ends, the timer is stopped and the time is recorded as a score.

2.3.3.3. Object motion

Given an object's path/trajectory, frame, and position, the module can get the next object position by simply feeding in the same path/trajectory into the creatures and projectiles modules, but with a different (presumably higher) frame. Notice that this is the only reason why the module needs each object's path/trajectory as an input, and it could be easily modified to make the design more modular. If the object is a projectile, then the frame must increase by a constant amount. If the object is a creature, then the frame increases by some number that depends on the timer. This is done so that creatures move faster and faster as the game progresses, to increase the game's difficulty.

2.3.3.4. Collisions

Collisions between active creatures and projectiles are determined differently depending on which approach is taken. For the Lookup method, the module would need to compare the position of a projectile with both the x coordinate of the creature and its height, which would depend on its y coordinate. For the Real Time generation method, the module just needs to compare if the projectile ever 'goes through' one of the creatures. This is easy to do, because it only needs to compare the positions coordinate by coordinate. Once a collision occurs, both objects involved must become inactive, and the appropriate game event signal needs to be generated.

2.3.3.5. New creatures

A 'new' creature must be generated periodically. This is very simple; every time a creature ends a path (its position is too close to the bottom of the screen or its frame reached a max), then its path needs to be replaced by a randomly chosen one, and its frame needs to be set to zero. Notice that this only applies to inactive creatures, because otherwise the game would end. To select the new path, pseudo-random numbers generated through an independent counter will be used.

2.3.3.6. New projectiles

'New' projectiles are generated periodically by simply assigning a new trajectory to currently inactive projectiles. This where the user inputs finally come in, as they determine which trajectory to pick. In the Lookup method case, the screen would be divided into several quadrants. Each quadrant corresponds to a path and an initial angle. The trajectory's initial angle and position is determined by the corresponding angle and path, respectively. In the Real Time generation case, the initial angle would be determined as a function of the height of the pointer. The initial position would always be at the bottom of the screen (where both y and z are zero), and the x coordinate is determined by the x coordinate of the pointer. And in either case, the power is simply given by the power signal coming from the inputs components.

2.4. GRAPHICS COMPONENT: (Implementation by Donald Eng)

2.4.1. Look up vs. 3-D Graphics Engine

Both the Lookup method and the Real Time generation method are being considered for game engine implementations. While the Lookup method requires a simple graphics engine for layering game sprites, the Real Time generation method would require a graphics engine capable of transforming objects located in 3-D space to objects on the 2-D space of the screen. It should be noted that the Lookup engine's reduced complexity is offset by the quality of the graphics produced. A Lookup engine would be a victim of the discretization of the all possible positions of the objects being displayed and the size of the memory storing these locations. Alternatively, the Real Time generation engine could produce graphics at a higher resolution without sacrificing memory. The disadvantage is generating the 2-D transformation of the 3-D coordinate system which could be computationally expensive on a pixel by pixel basis. Possible pipelining to produce graphics in real time with some latency only adds to the difficulty of implementing a Real Time graphics engine. However, the lookup table needs to be generated for both graphics and game play reducing the overall modularity of the graphics and game engine components. A primary advantage of 3-D space is the delocalization of the graphics engine and game play engine from any shared lookup table.

2.4.2. Lookup Graphics Engine

The Lookup engine is simple. Because trajectories and paths can be predetermined, no calculation is necessary for finding the location of object in the next frame. Hence x, y coordinates of the sprites can be parameterized in the memory by their PATH and FRAME or TRAJECTORY and FRAME as in a 2-D array of {x, y} coordinates. The PATH is the sequence of x, y coordinates that the sprite will assume in its lifetime, and the FRAME is how many frames the sprite has been alive for. Therefore, for a given sprite the graphics engine only requires its FRAME and which PATH or TRAJECTORY it lives on to create a full animation of the sprite. To create special animations using the FRAME could select a different sprite in the sequence of animations.

Game events are handled as a layering of sprites and backgrounds. Screens for losing and initializing values will be in the foreground of the game and 'opaque'. A game event bus will indicate which events need to be displayed from the game play module.

2.4.3. Real Time Graphics Engine

The Real Time engine would need the X, Y, Z, and FRAME fields of each object in the game. The game exists in a square rectangular prism. Looking into the square, the X spans the horizontal axis and the Y spans the vertical axis. This square would be the screen. The Z dimension shrinks back into the screen and from a 2-D perspective can be thought of as scaled concentric squares. Creatures and projectiles move into the screen and out of the background along the Z dimension. Hence, projectiles have a constant X coordinate, a constant velocity in Z (both determined when fired), and fall in Y according to a parabolic path. Because FRAMEs can be considered as units of time, each consecutive FRAME will propagate the object forward in time. Therefore, at every new FRAME (at every new Z coordinate of an object), determine which scaled concentric square of Z the object is in. Then scale the X and Y coordinates of the object at that instant in time by the scale factor associated with that Z and normalize the X, Y coordinate to the center of the screen. The new normalized X, Y = nX, nY are then the new sprite's location on the screen. Now traditional, sprite displaying can be used with lookup tables for sequences of sprites at specific frames and implementing sprite layering for 'depth'.

2.5. DEBUGGING/TESTING:

2.5.1. Devices/Inputs:

The devices are fairly straight-forward in testing. The camera will be directly connected to the display, and its quality will be easily determined based on how well it is able to capture distinctions of objects based on color differences. With the camera images, the video processing module can be tested by highlighting the targeted marker with a defined color as an indication that the module is able to track the marker and follow its movement accurately. With the medium filter module incorporated, the tracking of the marker should eliminate majority of the noises and produce a clear capture of the marker.

The shaker will be linked to the hex display on the FPGA labkit. A counter will be used to determine the time interval between consecutive pulses. Ideally, if the shaker is shaken slowly, more time would elapse between pulses, which would translate to a high value on the hex display and vice versa. One important note is that the counter will not be at every positive edge of the internal clock; instead, the counter will be incrementing at every half-second. The hex display can also be used to show the corresponding power level of the stream based on the intensity of the shaker's movement.

2.5.2. Game Engine:

Testing for the Game Engine seems to be particularly hard, because there is no simple way of visualizing the outputs, as can be done in the other two components. However, testing the creatures and projectiles modules should be fairly easy, since the set of inputs can be manipulated in any way. A set of test cases would be constructed and simply verified using the FPGA's display or LEDs. Other small parts of the game engine can also be tested similarly, such as the timer and random number generator.

The game evolution module seems harder to test due to its complexity. The easiest way to test it would be to start with very simple cases. For example, restrict the possible paths to only one, use predetermined inputs for the pointer location and power so that the trajectories to be used are predefined, and easily verify the game events being generated by use of the LEDs or display in the FPGA. Next, a very simple module that just displays all the positions of the objects on a monitor would be extremely useful to see if the game makes sense physically. Later on, the graphics component (even if incomplete) should be used to carry out more tests. This would also be useful because it would allow the two components to be integrated early on.

2.5.2. Graphics:

For both the Lookup Graphics Engine and the Real Time Graphics Engine, the debugging should take place in 2 main stages. Initially, a sprite should displayed on the screen at an arbitrary location and maintain its perspective. This single sprite represents a single instant in time. The next stage would be propagating time and creating the animation sequence for an object. Now the object is a sequence of sprites that changes with time, and should move in the game world according to the game's perspective.

To debug the Lookup Graphics Engine, a set of X, Y, FRAME, PATH/TRAJECTORY would be needed to produce sprites at various locations around the screen. For sufficient debugging, a full lookup table would need to be generated to animate objects traversing trajectories. Then, creating a FRAME count could simulate an animation by looking up a new X, Y for the given FRAME and a set PATH/TRAJECTORY.

The Real Time Graphics Engine would need a set of X, Y, Z values to display an object on the screen. This single object for some set X, Y, Z represents the objects 2-D transform of its 3-D coordinates at some instant in time. Changing the X, Y, Z should change the location of the object according to the perspective governing the game. Then devise some equations to govern the trajectory of an object: a constant X coordinate, a constant Z velocity, acceleration in Y, and a time counted in FRAMEs. For each set of equations increment the FRAMEs to propagate time and check that the animation of the object in 2-D space matches the expected perspective.

Initially, sprites could be implemented as scalable rectangles that grow and shrink along their paths according to their FRAME (lifetime). After demonstrating a dependable operation, the sprites could be upgraded to bitmaps. The bitmaps would of course need to be parsed from an outside source and stored in the memory of the FPGA.

2.5.2. Interfacing:

Because the Lookup Graphics Engine is not independent from the Game Engine Component (both depend heavily on the generation of an adequate lookup table), it would be difficult to generate each module independently. Thus, interfacing becomes problematic in the absence of the actual lookup table. Alternatively, the Game Engine Component operates in 3-D space which is the same coordinate system that is input into the Real Time Graphics Engine. Therefore a rudimentary Real Time Graphics Engine can assist the Game Engine Component. Once a simple Real Time Graphics Engine is developed, the Game Engine Component can be developed using this platform, and the Real Time Graphics Engine can be enhanced separately. Otherwise, the rate limiting step on both the Game Engine and Graphics Components would be the generation of lookup tables.